

Homework 4

Fisher Information Matrix, Independent Component Analysis

This notebook is arranged in cells. Texts are usually written in the markdown cells, and here you can use html tags (make it bold, italic, colored, etc). You can double click on this cell to see the formatting.

The ellipsis (...) are provided where you are expected to write your solution but feel free to change the template (not over much) in case this style is not to your taste.

Hit "Shift-Enter" on a code cell to evaluate it. Double click a Markdown cell to edit.

Imports

```
In [1]: import numpy as np
from scipy.integrate import quad
#For plotting
import matplotlib.pyplot as plt
%matplotlib inline
```

Mounting Google Drive locally

Mount your Google Drive on your runtime using an authorization code.

Note: When using the 'Mount Drive' button in the file browser, no authentication codes are necessary for notebooks that have only been edited by the current user.

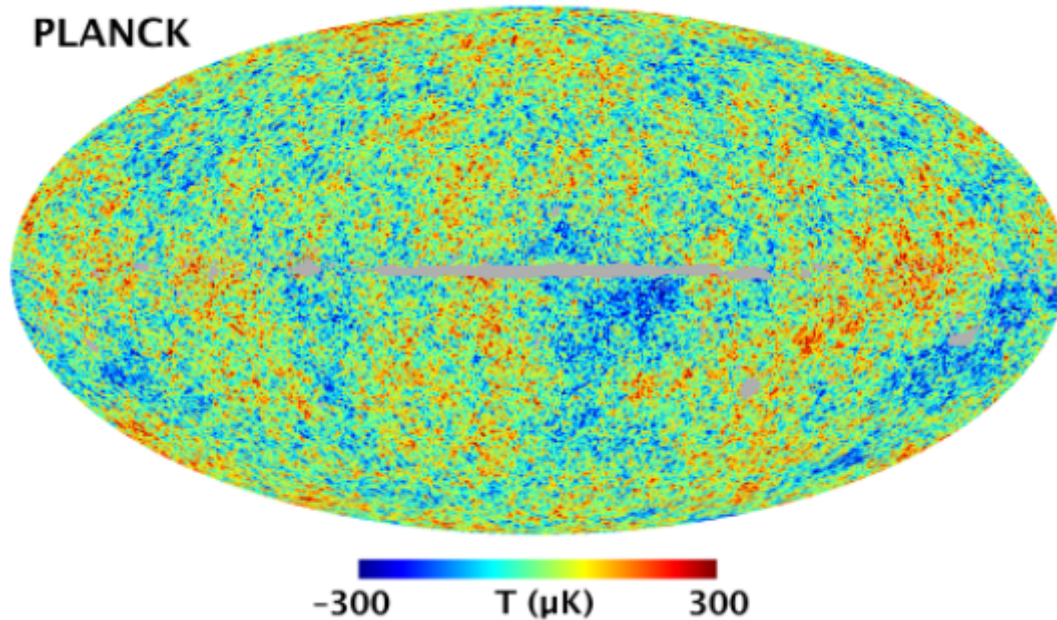
```
In [2]: from google.colab import drive
drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

Problem 1 - Constraining the cosmological parameters using the Planck power spectrum

Planck is the third-generation space telescope, following COBE and WMAP, and it aims to determine the geometry and content of the Universe by observing the cosmic microwave background radiation (CMB), emitted around 380,000 years after the Big Bang. Permeating the whole universe and containing information on the properties of the early Universe, the CMB is widely known as the strongest evidence for the Big Bang model.

Measuring the spectrum of the CMB, we confirm that it is very close to the radiation from an ideal blackbody, and fluctuations in the spectrum are very small. Averaging over all locations, its mean temperature is $2.725K$, and its root mean square temperature fluctuation is $\langle (\frac{\delta T}{T})^2 \rangle^{1/2} = 1.1 \times 10^{-5}$ (i.e. the temperature of the CMB varies by only $\sim 30 \mu K$ across the sky).



Suppose you observe the fluctuations $\delta T/T$. Since we are taking measurements on the surface of a sphere, it is useful to expand $\delta T/T$ in spherical harmonics (because they form a complete set of orthogonal functions on the sphere):

$$\frac{\delta T}{T}(\theta, \phi) = \sum_{l=0}^{\infty} \sum_{m=-l}^l a_{lm} Y_{lm}(\theta, \phi)$$

In flat space, we can do a Fourier transform of a function $f(x)$ as $\sum_k a_k e^{ikx}$ where k is the wavenumber, and $|a_k|$ determines the amplitude of the mode. For spherical harmonics, instead of k , we have l , the number of the modes along a meridian, and m , the number of modes along the equator. So l and m determine the wavelength ($\lambda = 2\pi/l$) and shape of the mode, respectively.

In cosmology, we are mostly interested in learning the statistical properties of this map and how different physical effects influence different physical scales, so it is useful to define the correlation function $C(\theta)$ and split the CMB map into different scales.

Suppose that we observe $\delta T/T$ at two different points on the sky. Relative to an observer, they are in direction \hat{n} and \hat{n}' and are separated by an angle θ given by $\cos\theta = \hat{n} \cdot \hat{n}'$. Then,

we can find the correlation function by multiplying together the values of $\delta T/T$ at the two points and average the product over all points separated by the angle θ .

$$C(\theta)^{TT} = \left\langle \frac{\delta T}{T}(\hat{n}) \frac{\delta T}{T}(\hat{n}') \right\rangle_{\hat{n} \cdot \hat{n}' = \cos \theta}$$

The above expression is specific to the temperature fluctuations, but we can also do a similar analysis for the polarization map of the CMB. (The CMB is polarized because it was scattered off of free electrons during decoupling.) We decompose the polarization pattern in the sky into a curl-free "E-mode" and grad-free "B-mode."

However, the CMB measurements (limited by the experiment resolution and the patch of sky examined) tell us about $C(\theta)$ over only a limited range of angular scales. (i.e. the precise values of $C(\theta)$ for all angles from $\theta = 0$ to $\theta = 180^\circ$ is not known.) Hence, using the expansion of $\delta T/T$ in spherical harmonics, we write the correlation function as:

$$C(\theta) = \frac{1}{4\pi} \sum_{l=0}^{\infty} (2l+1) C_l P_l(\cos \theta)$$

where P_l are the Legendre polynomials.

So we break down the correlation function into its multipole moments C_l , which is the angular power spectrum of the CMB. https://drive.google.com/uc?id=1zyduy-DE7U8zRJYW8ZMcMPO54STq_3M1

Remember that $\lambda = 2\pi/l$. So C_l measures the amplitude as a function of wavelength. ($C_l = \frac{1}{2l+1} \sum_{m=-l}^l |a_{lm}|^2$). In this problem, we will consider the temperature power spectrum C_l^{TT} , the E-mode power spectrum $C_l^{EE} = \frac{1}{2l+1} \sum_{m=-l}^l |a_{lm}^E|^2$, and the temperature-polarization cross-correlation $C_l^{TE} = \frac{1}{2l+1} \sum_{m=-l}^l a_{lm}^{T*} a_{lm}^E$.

The CMB angular power spectrum is usually expressed in terms of $D_l = l(l+1)C_l/2\pi$ (in unit of μK^2) because this better shows the contribution toward the variance of the temperature fluctuations.

Cosmologists built a software called "cosmological boltzmann code" which computes the theoretical power spectrum given cosmological parameters, such as the Hubble constant and the baryon density. Therefore, we can fit the theory power spectrum to the measured one in order to obtain the best-fit parameters.

Here, we consider six selected cosmological parameters,

$\vec{\theta} = [\theta_1, \theta_2, \dots, \theta_6] = [H_0, \Omega_b h^2, \Omega_c h^2, n_s, A_s, \tau]$. (H_0 = Hubble constant, $\Omega_b h^2$ = physical baryon density parameter, $\Omega_c h^2$ = physical cold dark matter density parameter, n_s

= scalar spectral index, A_s = curvature fluctuation amplitude, τ = reionization optical depth.). We provide you with the measured CMB power spectra from Planck Data Release 2.

References :

http://carina.fcaglp.unlp.edu.ar/extragalactica/Bibliografia/Ryden_IntroCosmo.pdf (Chapter 9, Intro to Cosmology, Barbara Ryden)

Fisher prediction for future CMB surveys

In class, we learned that the Fisher information matrix is useful for designing an experiment; we can vary the experiment design and predict the level of the expected error on any given parameter. In this problem, we aim to determine how well a low-noise, high-resolution future CMB survey would do in constraining the cosmological parameters.

The Fisher matrix is defined as the ensemble average of the Hessian of the log-likelihood ($\ln \mathcal{L}$) with respect to the given parameters $\vec{\theta}$:

$$F_{ij} = - \left\langle \frac{\partial^2 \ln \mathcal{L}}{\partial \theta_i \partial \theta_j} \right\rangle$$

Here we take the model CMB power spectrum as our observables. (Here we consider the auto-correlations D_l^{TT} , D_l^{EE} and cross-correlation D_l^{TE} obtained from the boltzmann code using the best-fit cosmological parameters from Planck, <https://arxiv.org/pdf/1502.01589v3.pdf>.) Then, we can estimate the Fisher matrix between two parameters θ_i and θ_j as:

$$F_{ij} = \sum_l \sum_k \frac{1}{(\sigma_l^k)^2} \frac{\partial D_l^k}{\partial \theta_i} \frac{\partial D_l^k}{\partial \theta_j}$$

where we sum over the CMB auto- and cross-power spectra

$D_l^k = [D_l^0, D_l^1, D_l^2] = [D_l^{TT}, D_l^{EE}, D_l^{TE}]$, and we assume that there is no correlation between them. σ^2 is the variance of D_l and noise N_l :

$$(\sigma_l^k)^2 = \frac{2}{(2l+1) \cdot f_{sky} \cdot \Delta l} (D_l^k + N_l^k)^2$$

f_{sky} is the fraction of the sky covered by the survey. Assume that $f_{sky} = 1$ for the sake of simplicity. Δl is the size of l -bin. Here, we set $l_{min} = 2$, $l_{max} = 2000$, and we have 92 l -bins in this range (For $2 \leq l < 30$, the power spectra are not binned ($\Delta l = 1$), and for $30 \leq l < 2000$, they are binned, and the bin size is $\Delta l = 30$). We obtain the measured and model power spectrum in that 92 l -bins.

In Part 1 and 2, the following is given:

(1) model power spectra ($D_l^{TT}, D_l^{EE}, D_l^{TE}$) evaluated at the best-fit values from Planck (i.e. assuming $\vec{\theta}_{best-fit}$)

(2) ($\frac{\partial D_l^{TT}}{\partial H_0} \Big|_{\vec{\theta}=\vec{\theta}_{best-fit}}, \frac{\partial D_l^{TT}}{\partial \Omega_b h^2} \Big|_{\vec{\theta}=\vec{\theta}_{best-fit}}$, etc): its derivative with respect to the parameter θ

evaluated at the best-fit values from Planck (i.e. assuming $\vec{\theta}_{best-fit}$). These are the measurement errors on $D_l^{TT}, D_l^{EE}, D_l^{TE}$.

(3) the measurement error $\sigma_{l,Planck}^{TT}, \sigma_{l,Planck}^{EE}, \sigma_{l,Planck}^{TE}$ assuming the noise from the current Planck survey

(4) (effective) l values for which the spectra are measured. (i.e. these are the effective bin center values for each l bin.)

In Part 3 and 4, you assume a zero-noise futuristic survey, so you need to compute new measurement error σ_l assuming $N_l = 0$.

Finally, we can compute the Fisher matrix F and obtain the covariance matrix C by inverting F :

$$[C] = [F]^{-1}$$

References:

Fisher Matrix Forecasting Review, Nicholas Kern

<https://arxiv.org/pdf/0906.4123.pdf>

1. First, load the measurement errors ($\sigma_l^{TT}, \sigma_l^{EE}, \sigma_l^{TE}$), model power spectrum ($D_l^{TT}, D_l^{EE}, D_l^{TE}$) and their derivatives with respect to six cosmological parameters evaluated at the best-fit values from Planck ($\frac{\partial D_l^{TT}}{\partial H_0} \Big|_{\vec{\theta}=\vec{\theta}_{best-fit}}, \frac{\partial D_l^{TT}}{\partial \Omega_b h^2} \Big|_{\vec{\theta}=\vec{\theta}_{best-fit}}$, etc). With the measurement errors from Planck, construct the Fisher matrix and the covariance matrix (you can use the `numpy.linalg.inv` for the matrix inversion). Evaluate the constraints on six parameters $\sigma(H_0), \sigma(\Omega_b h^2), \dots, \sigma(\tau)$ (corresponding to the square root of the diagonal entries of the covariance matrix). Print the results.

```

In [3]: # Load data

# Best-fit values of the cosmological parameters from https://arxiv.org/pdf/150
H0      = 67.27
ombh2   = 0.02225
omch2   = 0.1198
ns      = 0.9645
As      = 2.2065e-9
tau     = 0.079

theta_best_Planck = np.array([H0, ombh2, omch2, ns, As, tau])

data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_EE_me
# l (same for all power spectrum)
l = data[:,0]

# Planck noise

# sigma_l for D_l^EE assuming Planck N_l
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_EE_me
sigma_l_Planck_EE = data[:,2]

# sigma_l for D_l^TT assuming Planck N_l
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_TT_me
sigma_l_Planck_TT = data[:,2]

# sigma_l for D_l^TE assuming Planck N_l
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_TE_me
sigma_l_Planck_TE = data[:,2]

# Model power spectra given theta_best_Planck (calculated at the same l bins as

# D_l^EE (model)
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_EE_me
EE_model_Planck = data[:,1]

# D_l^TT (model)
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_TT_me
TT_model_Planck = data[:,1]

# D_l^TE (model)
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_TE_me
TE_model_Planck = data[:,1]

# Derivative of the power spectrum given theta_best_Planck (calculated at the s

# Derivative of D_l^EE with respect to six parameters
# ([theta1, theta2, theta3, theta4, theta5, theta6] = [H_0, \Omega_b h^2, \Omega_m h^2, \Omega_c h^2, n_s, tau])
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_deriv
deriv_DLEE_theta1 = data[:,1]
deriv_DLEE_theta2 = data[:,2]
deriv_DLEE_theta3 = data[:,3]
deriv_DLEE_theta4 = data[:,4]

```

```

deriv_DLEE_theta5 = data[:,5]
deriv_DLEE_theta6 = data[:,6]
deriv_DLEE_theta = data[:, 1:]

# Derivative of D_l^TT with respect to six parameters
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_deriv
deriv_DlTT_theta1 = data[:,1]
deriv_DlTT_theta2 = data[:,2]
deriv_DlTT_theta3 = data[:,3]
deriv_DlTT_theta4 = data[:,4]
deriv_DlTT_theta5 = data[:,5]
deriv_DlTT_theta6 = data[:,6]
deriv_DlTT_theta = data[:, 1:]

# Derivative of D_l^TE with respect to six parameters
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_deriv
deriv_DlTE_theta1 = data[:,1]
deriv_DlTE_theta2 = data[:,2]
deriv_DlTE_theta3 = data[:,3]
deriv_DlTE_theta4 = data[:,4]
deriv_DlTE_theta5 = data[:,5]
deriv_DlTE_theta6 = data[:,6]
deriv_DlTE_theta = data[:, 1:]

```

Fischer Information matrix:

$$F_{ij} = \sum_l \sum_k \frac{1}{(\sigma_l^k)^2} \frac{\partial D_l^k}{\partial \theta_i} \frac{\partial D_l^k}{\partial \theta_j}$$

```

In [4]: """
F_{ij} = \sum_l \sum_k \frac{1}{(\sigma_l^k)^2} \frac{\partial D^k_{\ell}}{\partial \theta_i} \frac{\partial D^k_{\ell}}{\partial \theta_j}
"""

# compute error:
sigma_l_1 = np.zeros((len(sigma_l_Planck_EE), 3))
sigma_l_1[:, 0] = sigma_l_Planck_EE
sigma_l_1[:, 1] = sigma_l_Planck_TT
sigma_l_1[:, 2] = sigma_l_Planck_TE

FM1 = np.zeros((6, 6))
for i in range(6):
    for j in range(6):
        fm = (deriv_DLEE_theta[:, i] * deriv_DLEE_theta[:, j]) / sigma_l_1[:, 0]
        fm += (deriv_DlTT_theta[:, i] * deriv_DlTT_theta[:, j]) / sigma_l_1[:, 1]
        fm += (deriv_DlTE_theta[:, i] * deriv_DlTE_theta[:, j]) / sigma_l_1[:, 2]
        FM1[i, j] = sum(fm)

# compute covariance matrix
Cov_mat1 = np.linalg.inv(FM1)
parameter_std1 = Cov_mat1.diagonal() ** 0.5

print('covariance matrix: \n', Cov_mat1)

```



```
print('\nfor the best parameters: H0 = {}, ombh2 = {}, omch2 = {}, ns = {}, As = {}')
print('\nstandard deviation of best parameters: \n', parameter_std1)
```

covariance matrix:

```
[[ 3.49026539e-01  5.69078119e-05 -7.60282072e-04  1.68962625e-03
   4.57765879e-12  1.94976566e-03]
 [ 5.69078119e-05  1.88461558e-08 -1.00651116e-07  1.86572863e-07
   8.01480269e-16  2.80327764e-07]
 [-7.60282072e-04 -1.00651116e-07  1.77921606e-06 -3.91118432e-06
  -9.54268738e-15 -4.38523901e-06]
 [ 1.68962625e-03  1.86572863e-07 -3.91118432e-06  1.35033967e-05
   2.29472680e-14  1.10871387e-05]
 [ 4.57765879e-12  8.01480269e-16 -9.54268738e-15  2.29472680e-14
   2.44117438e-21  5.64225137e-13]
 [ 1.94976566e-03  2.80327764e-07 -4.38523901e-06  1.10871387e-05
   5.64225137e-13  1.33726831e-04]]
```

for the best parameters: $H_0 = 67.27$, $ombh2 = 0.02225$, $omch2 = 0.1198$, $ns = 0.9645$, $As = 2.2065e-09$, $\tau = 0.079$

standard deviation of best parameters:

```
[5.90784680e-01 1.37281302e-04 1.33387258e-03 3.67469682e-03
 4.94082420e-11 1.15640318e-02]
```

Now from the covariance matrix, we can plot 1-d and 2-d constraints on the parameters. (See Fig. 6 in Planck 2015 paper <https://arxiv.org/pdf/1502.01589v3.pdf>)

1-d constraint (corresponding to the plots along the diagonal in Fig. 6, Planck 2015 paper):

First, the i th diagonal element of the covariance matrix correspond to $\sigma(\theta_i)^2$. Then, we can plot 1-d constraints on the parameter θ_i assuming a normal distribution with mean = $(\vec{\theta}_{best-fit})_i$ and variance = $\sigma(\theta_i)^2$.

2-d constraint (off-diagonal plots in Fig. 6, Planck 2015 paper):

Consider two parameters θ_i and θ_j from $\vec{\theta}$. Now marginalize over other parameters - in order to marginalize over other parameters, you can simply remove those parameters' row and column from the full covariance matrix. (i.e. From the full covariance matrix, you know the variance of all six parameters and their covariances with each other. So build a smaller dimension - 2×2 - covariance matrix from this.) - and obtain a 2×2 covariance matrix:

$$C_{ij} = \begin{pmatrix} \sigma(\theta_i)^2 & \text{Cov}(\theta_i, \theta_j) \\ \text{Cov}(\theta_i, \theta_j) & \sigma(\theta_j)^2 \end{pmatrix}$$

Now, we can plot the 2-dimensional confidence region ellipses from this matrix. The lengths of the ellipse axes are the square root of the eigenvalues of the covariance matrix, and we can calculate the counter-clockwise rotation of the ellipse with the rotation angle:

$$\phi = \frac{1}{2} \arctan\left(\frac{2 \cdot \text{Cov}(\theta_i, \theta_j)}{\sigma(\theta_i)^2 - \sigma(\theta_j)^2}\right) = \arctan\left(\frac{\vec{v}_1(y)}{\vec{v}_1(x)}\right)$$

where \vec{v}_1 is the eigenvector with the largest eigenvalue. So we calculate the angle of the largest eigenvector towards the x-axis to obtain the orientation of the ellipse.

Then, we multiply the axis lengths by some factor depending on the confidence level we are interested in. For 68%, this scale factor is $\sqrt{\Delta\chi^2} \approx 1.52$. For 95%, it is $\sqrt{\Delta\chi^2} \approx 2.48$.

Hint: For plotting ellipses, see HW3 Problem 1 Part 7.

2. From the covariance matrix, plot 1-d and 2-d constraints on the parameters. Note that the best-fit values of six parameters are already given in Part 1 (we just use the values from the Planck paper). For 2-d plot, show 68% and 95% confidence ellipses for pairs of parameters. Arrange those subplots in a triangle shape, as in Fig. 6, Planck 2015 (<https://arxiv.org/pdf/1502.01589v3.pdf>).

```
In [5]: from matplotlib import patches

param_names = np.array([r'$H_0$',
                        r'$\Omega_{bh}^2$',
                        r'$\Omega_{ch}^2$',
                        r'$n_s$',
                        r'$10^9 A_s$',
                        r'$\tau$'])
params = [H0, ombh2, omch2, ns, As, tau]

def my_gaussian(x, a, mu, sigma):
    norm = (sigma * np.sqrt(2 * np.pi))
    return a * np.exp(-((np.array(x) - mu) / sigma) ** 2 / 2) / norm

# plotting
fs = 20
fig, axes = plt.subplots(6, 6, figsize=(10, 10))
for i in range(6):
    for j in range(6):
        # set plot
        ax = axes[i, j]

        # only plot lower triangular interactions
        if j > i:
            ax.axis('off')
            continue

        # plot diagonal distributions
        elif i == j:
            xarr = np.linspace(params[i] - 5 * parameter_std1[i], params[i] + 5 * parameter_std1[i], 100)
            ax.plot(xarr, my_gaussian(xarr, 1, params[i], parameter_std1[i]), 'b')
            ax.set_xticks([params[i] - 2 * parameter_std1[i],
                          params[i] - 1 * parameter_std1[i],
                          params[i],
                          params[i] + 1 * parameter_std1[i],
                          params[i] + 2 * parameter_std1[i]])
```

```

        params[i] + 1 * parameter_std1[i],
        params[i] + 2 * parameter_std1[i]])

ax.set_yticklabels([])
ax.set_xticklabels([])
ax.set_xlim(params[i] - 3 * parameter_std1[i],
             params[i] + 3 * parameter_std1[i])

# off-diagonal part
elif j < i:

    # compute SVD of covariance
    CovMM = np.array([[Cov_mat1[i, i], Cov_mat1[i, j]],
                      [Cov_mat1[j, i], Cov_mat1[j, j]]])

    eigvec, eigval, u = np.linalg.svd(CovMM)
    semi_major = 2 * np.sqrt(eigval[0])
    semi_minor = 2 * np.sqrt(eigval[1])

    # Rotation angle of the ellipse
    theta = np.arctan(eigvec[0][1] / eigvec[0][0])

    # Create ellipses
    l_1 = patches.Ellipse(xy=[params[i], params[j]],
                           width=1.52*semi_major,
                           height=1.52*semi_minor,
                           angle = theta*180/np.pi,
                           facecolor='crimson',
                           edgecolor='red',
                           label='68% confidence')

    l_2 = patches.Ellipse(xy=[params[i], params[j]],
                           width=2.48*semi_major,
                           height=2.48*semi_minor,
                           angle=theta*180/np.pi,
                           facecolor='rebeccapurple',
                           edgecolor='mediumpurple',
                           label='95% confidence')

ax.add_patch(l_2)
ax.add_patch(l_1)

ax.set_ylim(params[j] - 3 * parameter_std1[j],
            params[j] + 3 * parameter_std1[j])
ax.set_xlim(params[i] - 3 * parameter_std1[i],
            params[i] + 3 * parameter_std1[i])

ax.set_xticks([params[i] - 2 * parameter_std1[i],
               params[i] - 1 * parameter_std1[i],
               params[i],
               params[i] + 1 * parameter_std1[i],
               params[i] + 2 * parameter_std1[i]])

ax.set_yticks([params[j] - 2 * parameter_std1[j],
               params[j] - 1 * parameter_std1[j],
               params[j],
               params[j] + 1 * parameter_std1[j],

```

```

        params[j] + 2 * parameter_std1[j]])

    else:
        raise Exception('Issue while plotting')

    # edit individual frames
    yls = [params[i] - 2 * parameter_std1[i],
           params[i] - 1 * parameter_std1[i],
           params[i],
           params[i] + 1 * parameter_std1[i],
           params[i] + 2 * parameter_std1[i]]

    xls = [params[j] - 2 * parameter_std1[j],
           params[j] - 1 * parameter_std1[j],
           params[j],
           params[j] + 1 * parameter_std1[j],
           params[j] + 2 * parameter_std1[j]]

    if j != 0:
        ax.set_yticklabels([])
    if i != 5:
        ax.set_xticklabels([])

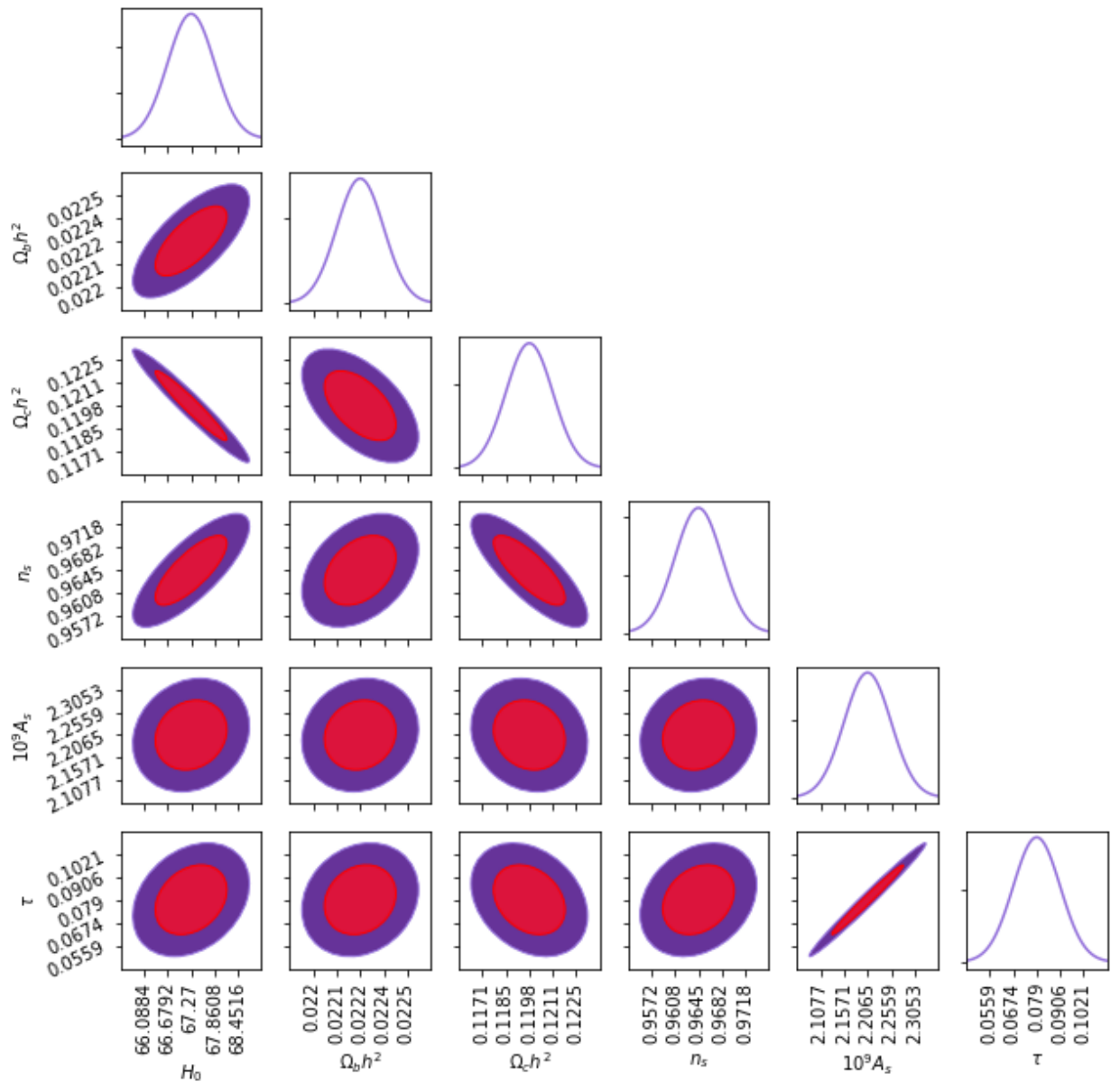
    if j == 0 and i != 0:
        ax.set_ylabel(param_names[i], fontsize=10)

        if i == 4:
            ax.set_yticklabels(np.around(np.array(yls) * 1e9, 4))
        else:
            ax.set_yticklabels(np.around(np.array(yls), 4))

        # ax.set_yticklabels(np.around(yls, 4))
        [tl.set_rotation(26) for tl in ax.get_yticklabels()]

    if i == 5:
        ax.set_xlabel(param_names[j], fontsize=10)
        if j == 4:
            ax.set_xticklabels(np.around(np.array(xls) * 1e9, 4))
        else:
            ax.set_xticklabels(np.around(np.array(xls), 4))
        # ax.set_xticklabels(np.around(xls, decimals = 4))
        [tl.set_rotation(90) for tl in ax.get_xticklabels()]

```



Now, assume that we have an ideal, zero-noise CMB survey with $N_l = 0$. However, we are still intrinsically limited on the number of independent modes we can measure (there are only $(2l+1)$ of them) - $C_l = \frac{1}{2l+1} \sum_{m=-l}^l \langle |a_{lm}|^2 \rangle$. This leads that we get an intrinsic error (called "cosmic variance") in our estimate of C_l . So we approximate that

$$(\sigma_l^{EE})^2 = \frac{2}{(2l+1) \cdot f_{sky} \cdot \Delta l} (D_l^{EE})^2, \quad (\sigma_l^{TT})^2 = \frac{2}{(2l+1) \cdot f_{sky} \cdot \Delta l} (D_l^{TT})^2,$$

$$(\sigma_l^{TE})^2 = \frac{2}{(2l+1) \cdot f_{sky} \cdot \Delta l} \frac{(D_l^{TE})^2 + D_l^{TT} D_l^{EE}}{2}$$

3. First compute σ_l for this zero-noise futuristic survey. (assuming $N_l^k = 0$) Repeat Part 1 and 2. (How well does a zero-noise CMB survey constrain the cosmological parameters?)

```
In [6]: data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4/Project1_EE_me
l_ = data[:,0]
D_l = np.ones(len(l_))
D_l[28:] *= 30

sig_l_2 = np.zeros((len(sigma_l_Planck_EE), 3))

sig_l_2[:, 0] = 2 * EE_model_Planck ** 2 / ((2 * l_ + 1) * D_l)
sig_l_2[:, 1] = 2 * TT_model_Planck ** 2 / ((2 * l_ + 1) * D_l)
sig_l_2[:, 2] = 2 * (TE_model_Planck ** 2 + EE_model_Planck * TT_model_Planck)
sig_l_2 **= 0.5

FM2 = np.zeros((6, 6))
for i in range(6):
    for j in range(6):
        fm = (deriv_DLEE_theta[:, i] * deriv_DLEE_theta[:, j]) / sig_l_2[:, 0]
        fm += (deriv_DLTT_theta[:, i] * deriv_DLTT_theta[:, j]) / sig_l_2[:, 1]
        fm += (deriv_DLTE_theta[:, i] * deriv_DLTE_theta[:, j]) / sig_l_2[:, 2]
        FM2[i, j] = sum(fm)

# compute covariance matrix
Cov_mat2 = np.linalg.inv(FM2)
parameter_std2 = Cov_mat2.diagonal() ** 0.5
print('covariance matrix: \n', Cov_mat2)
print('\nfor the best parameters: H0 = {}, ombh2 = {}, omch2 = {}, ns = {}, As
print('\nstandard deviation of best parameters: \n', parameter_std2)

# plotting
fs = 20
fig, axes = plt.subplots(6, 6, figsize=(10, 10))
for i in range(6):
    for j in range(6):
        # set plot
        ax = axes[i, j]

        # only plot lower triangular interactions
        if j > i:
            ax.axis('off')
            continue

        # plot diagonal distributions
        elif i == j:
            xarr = np.linspace(params[i] - 5 * parameter_std2[i], params[i] + 5
            ax.plot(xarr, my_gaussian(xarr, 1, params[i], parameter_std2[i]), 1
            ax.set_xticks([params[i] - 2 * parameter_std2[i],
                           params[i] - 1 * parameter_std2[i],
                           params[i],
                           params[i] + 1 * parameter_std2[i],
                           params[i] + 2 * parameter_std2[i]])

            ax.set_yticklabels([])
            ax.set_xticklabels([])
```

```

ax.set_xlim(params[i] - 3 * parameter_std2[i],
             params[i] + 3 * parameter_std2[i])

# off-diagonal part
elif j < i:

    # compute SVD of covariance
    CovMM = np.array([[Cov_mat2[i, i], Cov_mat2[i, j]],
                      [Cov_mat2[j, i], Cov_mat2[j, j]]])

    eigvec, eigval, u = np.linalg.svd(CovMM)
    semi_major = 2 * np.sqrt(eigval[0])
    semi_minor = 2 * np.sqrt(eigval[1])

    # Rotation angle of the ellipse
    theta = np.arctan(eigvec[0][1] / eigvec[0][0])

    # Create ellipses
    l_1 = patches.Ellipse(xy=[params[i], params[j]],
                           width=1.52*semi_major,
                           height=1.52*semi_minor,
                           angle = theta*180/np.pi,
                           facecolor='maroon',
                           edgecolor='salmon',
                           label='68% confidence')

    l_2 = patches.Ellipse(xy=[params[i], params[j]],
                           width=2.48*semi_major,
                           height=2.48*semi_minor,
                           angle=theta*180/np.pi,
                           facecolor='darkolivegreen',
                           edgecolor='olivedrab',
                           label='95% confidence')

    ax.add_patch(l_2)
    ax.add_patch(l_1)

    ax.set_ylim(params[j] - 3 * parameter_std2[j],
                 params[j] + 3 * parameter_std2[j])
    ax.set_xlim(params[i] - 3 * parameter_std2[i],
                 params[i] + 3 * parameter_std2[i])

    ax.set_xticks([params[i] - 2 * parameter_std2[i],
                   params[i] - 1 * parameter_std2[i],
                   params[i],
                   params[i] + 1 * parameter_std2[i],
                   params[i] + 2 * parameter_std2[i]])

    ax.set_yticks([params[j] - 2 * parameter_std2[j],
                   params[j] - 1 * parameter_std2[j],
                   params[j],
                   params[j] + 1 * parameter_std2[j],
                   params[j] + 2 * parameter_std2[j]])

else:
    raise Exception('Issue while plotting')

```

```

# edit individual frames
yls = [params[i] - 2 * parameter_std2[i],
       params[i] - 1 * parameter_std2[i],
       params[i],
       params[i] + 1 * parameter_std2[i],
       params[i] + 2 * parameter_std2[i]]

xls = [params[j] - 2 * parameter_std2[j],
       params[j] - 1 * parameter_std2[j],
       params[j],
       params[j] + 1 * parameter_std2[j],
       params[j] + 2 * parameter_std2[j]]

if j != 0:
    ax.set_yticklabels([])
if i != 5:
    ax.set_xticklabels([])

if j == 0 and i != 0:
    ax.set_ylabel(param_names[i], fontsize=10)

    if i == 4:
        ax.set_yticklabels(np.around(np.array(yls) * 1e9, 4))
    else:
        ax.set_yticklabels(np.around(np.array(yls), 4))

    # ax.set_yticklabels(np.around(yls, 4))
    [tl.set_rotation(26) for tl in ax.get_yticklabels()]

if i == 5:
    ax.set_xlabel(param_names[j], fontsize=10)
    if j == 4:
        ax.set_xticklabels(np.around(np.array(xls) * 1e9, 4))
    else:
        ax.set_xticklabels(np.around(np.array(xls), 4))
    # ax.set_xticklabels(np.around(xls, decimals = 4))
    [tl.set_rotation(90) for tl in ax.get_xticklabels()]

```

covariance matrix:

```

[[ 4.10562422e-02  4.39183415e-06 -1.00977342e-04  1.74206677e-04
   4.49686287e-13  2.22139540e-04]
 [ 4.39183415e-06  2.46193272e-09 -7.01129745e-09 -1.58888136e-08
   8.19869199e-17  1.78732051e-08]
 [-1.00977342e-04 -7.01129745e-09  2.60563315e-07 -4.94609574e-07
  -1.02776976e-15 -5.62327204e-07]
 [ 1.74206677e-04 -1.58888136e-08 -4.94609574e-07  2.50558004e-06
   9.07205472e-16  1.32271124e-06]
 [ 4.49686287e-13  8.19869199e-17 -1.02776976e-15  9.07205472e-16
   1.00817752e-22  2.36926561e-14]
 [ 2.22139540e-04  1.78732051e-08 -5.62327204e-07  1.32271124e-06
   2.36926561e-14  6.19873868e-06]]

```

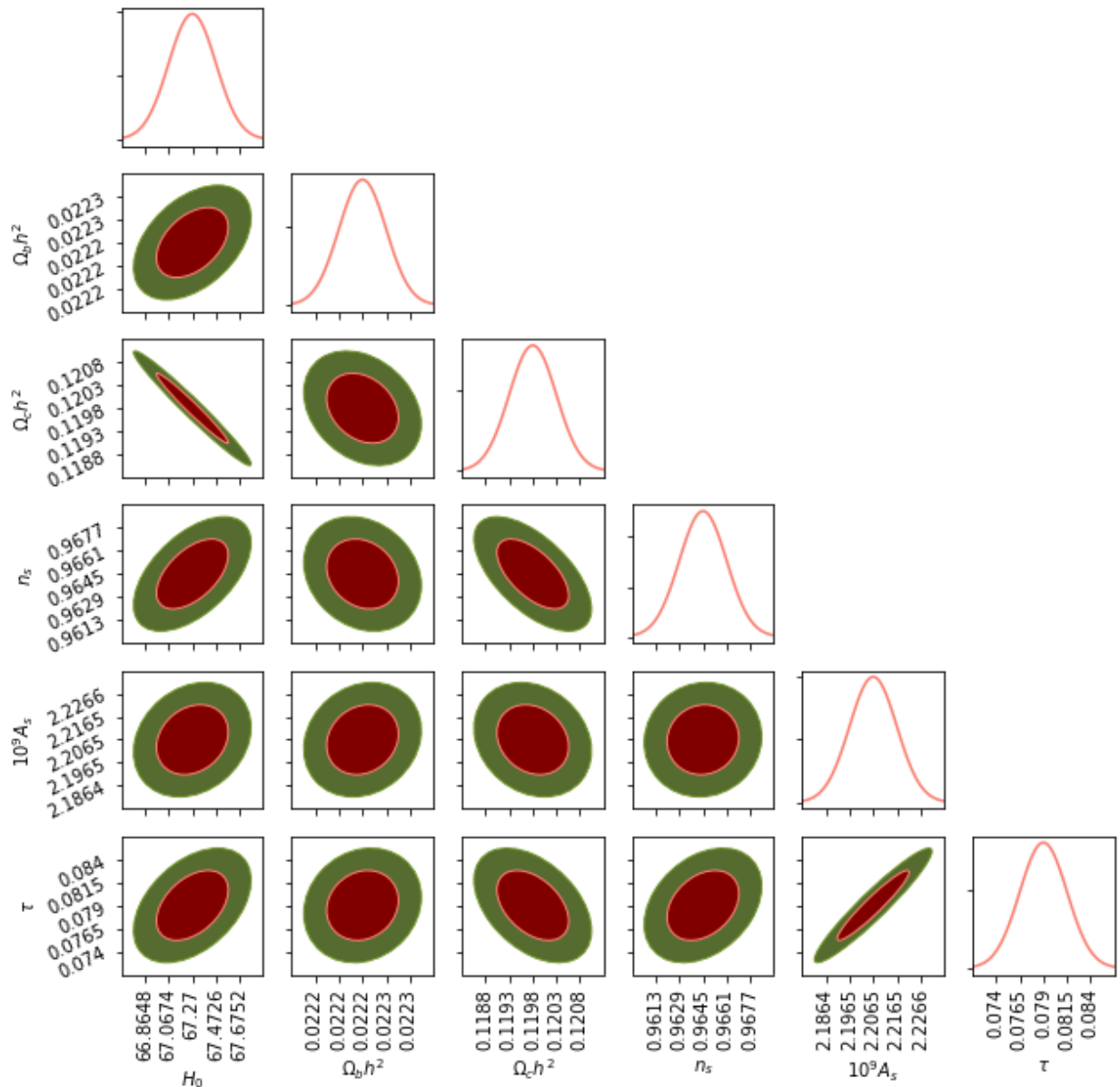
for the best parameters: $H_0 = 67.27$, $ombh^2 = 0.02225$, $omch^2 = 0.1198$, $ns = 0.9645$, $As = 2.2065e-09$, $\tau = 0.079$

standard deviation of best parameters:

```

[2.02623400e-01 4.96178669e-05 5.10454028e-04 1.58290241e-03
 1.00408044e-11 2.48972663e-03]

```

4. Combine Part 2 and Part 3 and compare. (First plot your results from Part 2 (1-d and 2-d constraints using the Planck power spectra and noise). Then, plot Part 3 results (assuming zero noise) on top with different colors. Note that your 1-d constraints in Part 3 are more sharply peaked Gaussians (with much smaller variances), so you can scale them so that its peak amplitudes match with your results from Part 2.)

```
In [7]: fig, axes = plt.subplots(6,6,figsize=(12,12))
fig.subplots_adjust(wspace=0, hspace=0)

for i in range(6):
    for j in range(6):
        ax = axes[i, j]
        # only plot lower triangular interactions
        if j > i:
            ax.axis('off')
            continue

        # plot diagonal distributions
        elif i == j:
```

```

x = np.linspace(params[i] - 5 * parameter_std1[i], params[i] + 5 *

ax.plot(x, my_gaussian(x, parameter_std1[i], params[i], parameter_s
ax.plot(x, my_gaussian(x, parameter_std2[i], params[i], parameter_s

ax.set_xticks([params[i] - 2 * parameter_std1[i],
               params[i] - 2 * parameter_std2[i],
               params[i] - 1 * parameter_std1[i],
               params[i] - 1 * parameter_std2[i],
               params[i],
               params[i]+1*parameter_std2[i],
               params[i]+1*parameter_std1[i],
               params[i]+2*parameter_std2[i],
               params[i]+2*parameter_std1[i]])

ax.set_yticklabels([])
ax.set_xticklabels([])

ax.set_xlim(params[i] - 3 * parameter_std1[i], params[i] + 3 * para

# off-diagonal part
elif j < i:

    # compute SVD of covariance
    CovMM1 = np.array([[Cov_mat1[i, i], Cov_mat1[i, j]],
                       [Cov_mat1[j, i], Cov_mat1[j, j]]])
    eigvec1, eigval1, u1 = np.linalg.svd(CovMM1)

    CovMM2 = np.array([[Cov_mat2[i, i], Cov_mat2[i, j]],
                       [Cov_mat2[j, i], Cov_mat2[j, j]]])
    eigvec2, eigval2, u2 = np.linalg.svd(CovMM2)

    # Get Semimajor/minor axes of the ellipse
    semi_major1 = 2 * np.sqrt(eigval1[0])
    semi_minor1 = 2 * np.sqrt(eigval1[1])

    semi_major2 = 2 * np.sqrt(eigval2[0])
    semi_minor2 = 2 * np.sqrt(eigval2[1])

    # rotate
    theta1 = np.arctan(eigvec1[0][1] / eigvec1[0][0])
    theta2 = np.arctan(eigvec2[0][1] / eigvec2[0][0])

    # Create ellipses
    ax.add_patch(patches.Ellipse(xy=[params[i], params[j]],
                                  width=2.48*semi_major1,
                                  height=2.48*semi_minor1,
                                  angle = theta1*180/np.pi,
                                  facecolor = 'honeydew',
                                  edgecolor = 'green',
                                  label = '95% conf.'))

    ax.add_patch(patches.Ellipse(xy=[params[i], params[j]],
                                  width=1.52*semi_major1,
                                  height=1.52*semi_minor1,
                                  angle=theta1*180/np.pi,

```

```

        facecolor='green',
        edgecolor='honeydew',
        label='68% conf.'))

    ax.add_patch(patches.Ellipse(xy=[params[i], params[j]],
                                width=2.48*semi_major2,
                                height=2.48*semi_minor2,
                                angle=theta2*180/np.pi,
                                facecolor='skyblue',
                                edgecolor='royalblue',
                                label='95% conf.: N=0'))

    ax.add_patch(patches.Ellipse(xy=[params[i], params[j]],
                                width=1.52*semi_major2,
                                height=1.52*semi_minor2,
                                angle=theta2*180/np.pi,
                                facecolor='dodgerblue',
                                edgecolor='royalblue',
                                label='68% conf.: N=0'))

    ax.set_ylim(params[j] - 3 * parameter_std1[j],
                params[j] + 3 * parameter_std1[j])

    ax.set_xlim(params[i] - 3 * parameter_std1[i],
                params[i] + 3 * parameter_std1[i])

    ax.set_xticks([params[i] - 2 * parameter_std1[i],
                    params[i] - 1 * parameter_std1[i],
                    params[i],
                    params[i] + 1 * parameter_std1[i],
                    params[i] + 2 * parameter_std1[i]])

    ax.set_yticks([params[j] - 2 * parameter_std1[j],
                    params[j] - 1 * parameter_std1[j],
                    params[j],
                    params[j] + 1 * parameter_std1[j],
                    params[j] + 2 * parameter_std1[j]])

    else:
        raise Exception('plotting error')

    yls = [params[i] - 2 * parameter_std1[i],
            params[i] - 1 * parameter_std1[i],
            params[i],
            params[i] + 1 * parameter_std1[i],
            params[i] + 2 * parameter_std1[i]]
    xls = [params[j] - 2 * parameter_std1[j],
            params[j] - 1 * parameter_std1[j],
            params[j],
            params[j] + 1 * parameter_std1[j],
            params[j] + 2 * parameter_std1[j]]

    # make graphs pretty
    if j != 0:

```

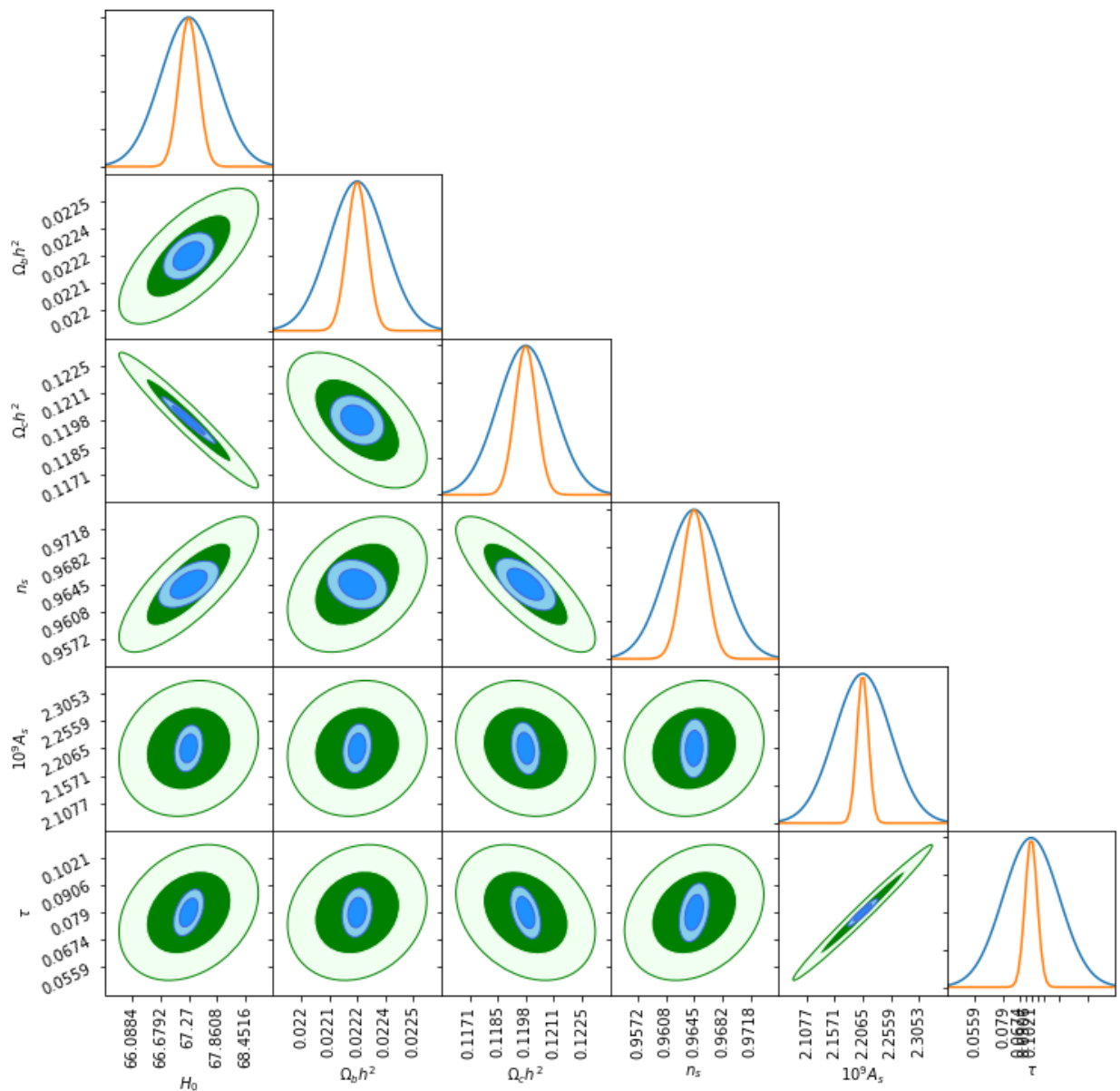
```
ax.set_yticklabels([])
if i != 5:
    ax.set_xticklabels([])

if j == 0 and i != 0:
    ax.set_ylabel(param_names[i], fontsize=10)

    if i == 4:
        ax.set_yticklabels(np.around(np.array(yls) * 1e9, 4))
    else:
        ax.set_yticklabels(np.around(np.array(yls), 4))

    # ax.set_yticklabels(np.around(yls, 4))
    [tl.set_rotation(26) for tl in ax.get_yticklabels()]

if i == 5:
    ax.set_xlabel(param_names[j], fontsize=10)
    if j == 4:
        ax.set_xticklabels(np.around(np.array(xls) * 1e9, 4))
    else:
        ax.set_xticklabels(np.around(np.array(xls), 4))
    # ax.set_xticklabels(np.around(xls, decimals = 4))
    [tl.set_rotation(90) for tl in ax.get_xticklabels()]
```



5. Starting from the best-fit values from the Planck 2015 paper, you constrained six cosmological parameters assuming that you have a zero-noise future CMB survey. Compare your results with Table 3 and Figure 6 in <https://arxiv.org/pdf/1502.01589v3.pdf>.

Answer:

According to the data reported in the Planck paper in Table3 (last column), the results computed in this assignment with a zero-noise assumption appear to be tighter than the reported constraints with noise. This is expected since noise will cause a greater variance.

Problem 2 - Solving the cocktail party problem with ICA

"Independent component analysis was originally developed to deal with problems that are closely related to the cocktail-party problem. The goal is to find a linear representation of

nongaussian data so that the components are statistically independent, or as independent as possible. Such a representation seems to capture the essential structure of the data in many applications, including feature extraction and signal separation." More details on ICA can be found in <https://www.cs.helsinki.fi/u/ahyvarin/papers/NN00new.pdf>

In this problem, we take the mixed sounds and images, and apply ICA to them to separate the sources.

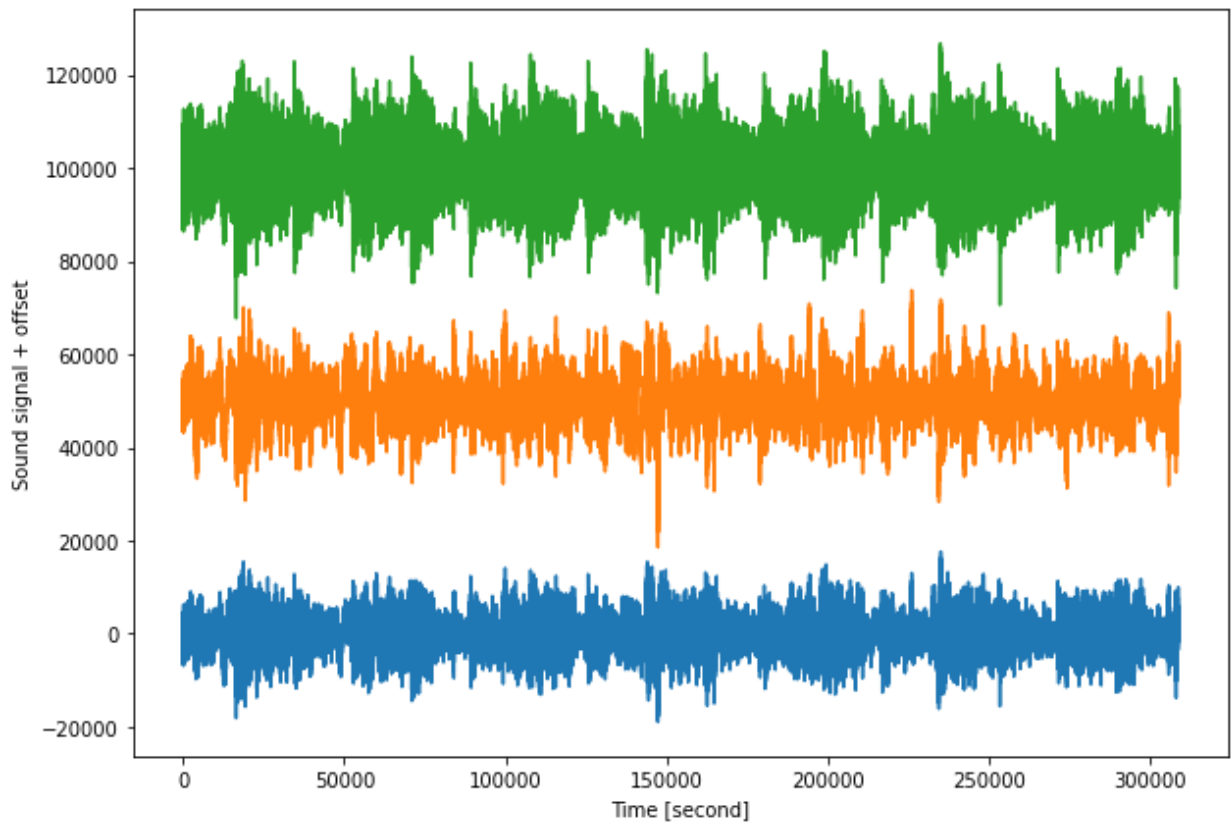
1. Read the 3 sound files, and plot them as a function of time. (In order to better see the features, you may plot them with some offsets.)

```
In [8]: from scipy.io import wavfile
```

```
In [9]: fs, data1 = wavfile.read('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sou
fs, data2 = wavfile.read('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sou
fs, data3 = wavfile.read('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sou
#fs is the sample rate, i.e., how many data points in one second.

data = np.float64(np.c_[data1, data2, data3])
```

```
In [10]: label = ['label_1', 'label_2', 'label_3']
offset = 5e4
plt.figure(figsize = (10,7))
for i in range(3):
    plt.plot(data[:, i] + i * offset, label=label[i])
plt.xlabel('Time [second]')
plt.ylabel('Sound signal + offset')
plt.show()
```



2. Now run the following cells and play the sounds.

```
In [11]: from IPython.display import Audio
```

```
In [12]: Audio('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sound1.wav')
```

Out[12]:

```
In [13]: Audio('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sound2.wav')
```

Out[13]:

```
In [14]: Audio('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sound3.wav')
```

Out[14]:

You can tell there are 3 signals mixed in these sounds. You can consider these sounds as recorded by 3 different recorders that have different distance to the 3 sources. In other words,

$$\mathbf{X} = \mathbf{A}\mathbf{S} + \boldsymbol{\mu} \quad (1)$$

where \mathbf{X} is a vector of these 3 sounds, \mathbf{S} is a vector of 3 signals, $\boldsymbol{\mu}$ is a vector of the mean of \mathbf{X} , and \mathbf{A} is the mixing matrix.

Next, using the sklearn's fastICA module, we will separate the signals from these sounds.

(1) Define the fastICA model:

```
ica = FastICA(algorithm='parallel')
```

(2) Using "fit_transform," fit the model with the data and obtain the signals

```
S = ica.fit_transform(data)
```

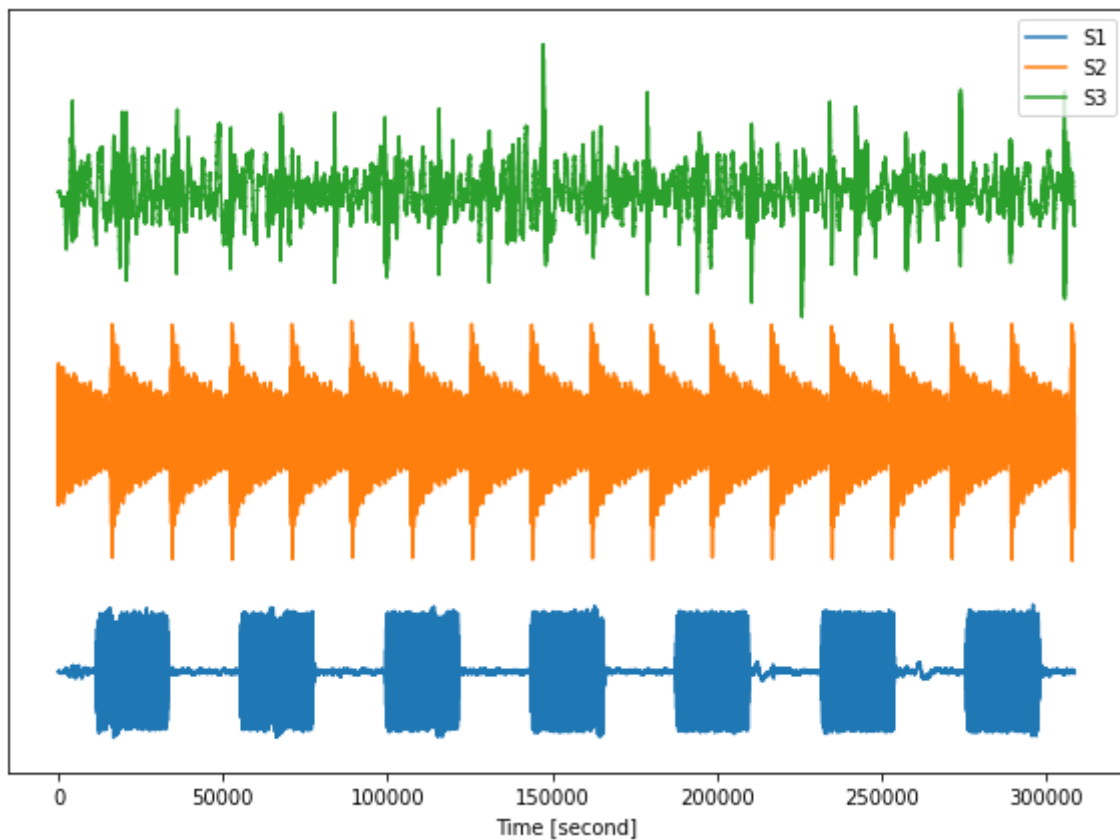
3. Find the 3 signals in the sound files. Plot the signals. (Again, you may plot them with some offsets.)

```
In [15]: from sklearn.decomposition import FastICA
```

```
In [16]: ica = FastICA()
S = ica.fit_transform(data)
S_label = ['S1', 'S2', 'S3']

# Plot S
S_offset = 0.02
plt.figure(figsize = (10,7))
for i in range(3):
    plt.plot(S[:, i] + i * S_offset, label=S_label[i])

plt.xlabel('Time [second]')
plt.yticks([])
plt.legend()
plt.show()
```



You will find the amplitude of the signals is very small. This is because fastICA whitens the data first before applying ICA, so that the covariance matrix of the signals is I . We can amplify the signals by multiplying them with a large number.

4. Now let's save the signals as wav files and play the sounds.

```
In [17]: Amp = 1e6

wavfile.write('signal_sound1.wav', fs, np.int16(Amp*S[:,0]))
wavfile.write('signal_sound2.wav', fs, np.int16(Amp*S[:,1]))
wavfile.write('signal_sound3.wav', fs, np.int16(Amp*S[:,2]))
```

Play the sounds.

```
In [18]: Audio('signal_sound1.wav')
```

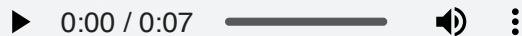
Out[18]:

```
In [19]: Audio('signal_sound2.wav')
```

Out[19]:

```
In [20]: Audio('signal_sound3.wav')
```

Out [20]:



Now we can reconstruct the mixed sounds with the signals. The mixing matrix is given by `ica.mixing_`, and the mean of the data is given by `ica.mean_`. Note that the X and S from equation (1) are matrices of shape (Nsignal, Nsample), but the data and the signal you get from FastICA are matrices of shape (Nsample, Nsignal).

5. Reconstruct the sounds from the source signal, and show that the reconstruct sounds is very close to the given sounds using `np.allclose`

```
In [21]: S_mix = ica.mixing_
S_mean = ica.mean_

data_ = S_mix @ S.T
data_[0, :] += S_mean[0]
data_[1, :] += S_mean[1]
data_[2, :] += S_mean[2]

print('Using np.allclose(), show that the reconstructed sounds are clos to the
```

Using `np.allclose()`, show that the reconstructed sounds are clos to the given sounds: True

```
In [22]: print(np.allclose(data_.T, data))

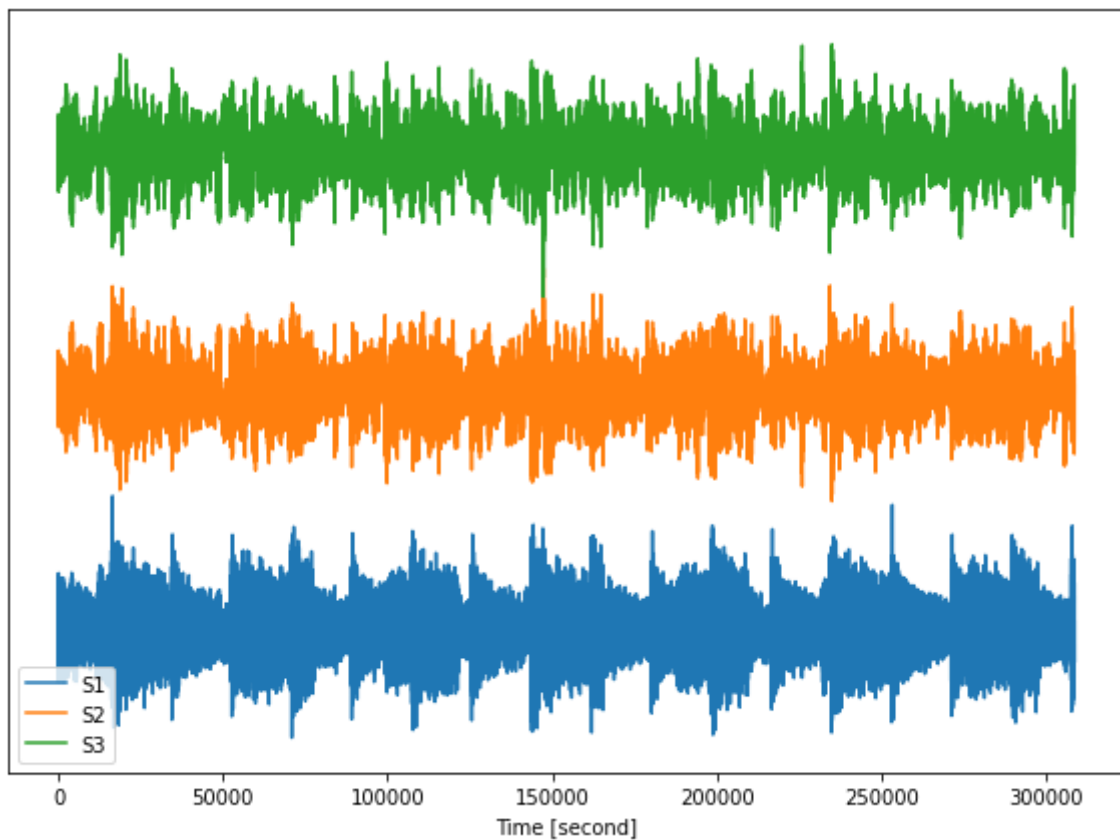
True
```

6. The ICA requires the data to be centered and whitened. The FastICA module from sklearn does the data centering and whitening automatically. Now let's disable the data preprocessing in FastICA by `ica = FastICA(whiten=False)`, and then redo the analysis in part 3 and 4. Plot and play the sounds. Does ICA work without data preprocessing?

```
In [23]: ica = FastICA(whiten=False)
S = ica.fit_transform(data)

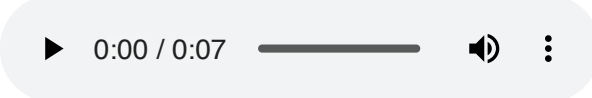
# plot
S_offset = 5e4
plt.figure(figsize=(10, 7))
for i in range(3):
    plt.plot(S[:, i] + i * S_offset, label=S_label[i])
plt.xlabel('Time [second]')
plt.yticks([])
plt.legend()
plt.show()

# rewrite files
Amp = 2e6
wavfile.write('signal_sound1.wav', fs, np.int16(Amp*S[:,0]))
wavfile.write('signal_sound2.wav', fs, np.int16(Amp*S[:,1]))
wavfile.write('signal_sound3.wav', fs, np.int16(Amp*S[:,2]))
```



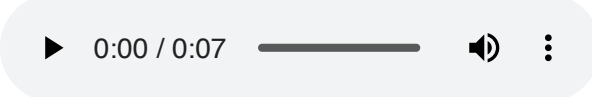
```
In [24]: Audio('signal_sound1.wav')
```

Out[24]:



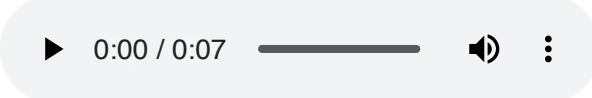
```
In [25]: Audio('signal_sound2.wav')
```

Out[25]:



```
In [26]: Audio('signal_sound3.wav')
```

Out[26]:



Answer:

No, FastICA does not work without data preprocessing beforehand.

7. The principal component analysis (PCA) also tries to interpret the underlying structure of the data by decomposing the data into linear combinations of the principal components. Now let's apply PCA to the sounds and see if the signals are cleanly separated in the

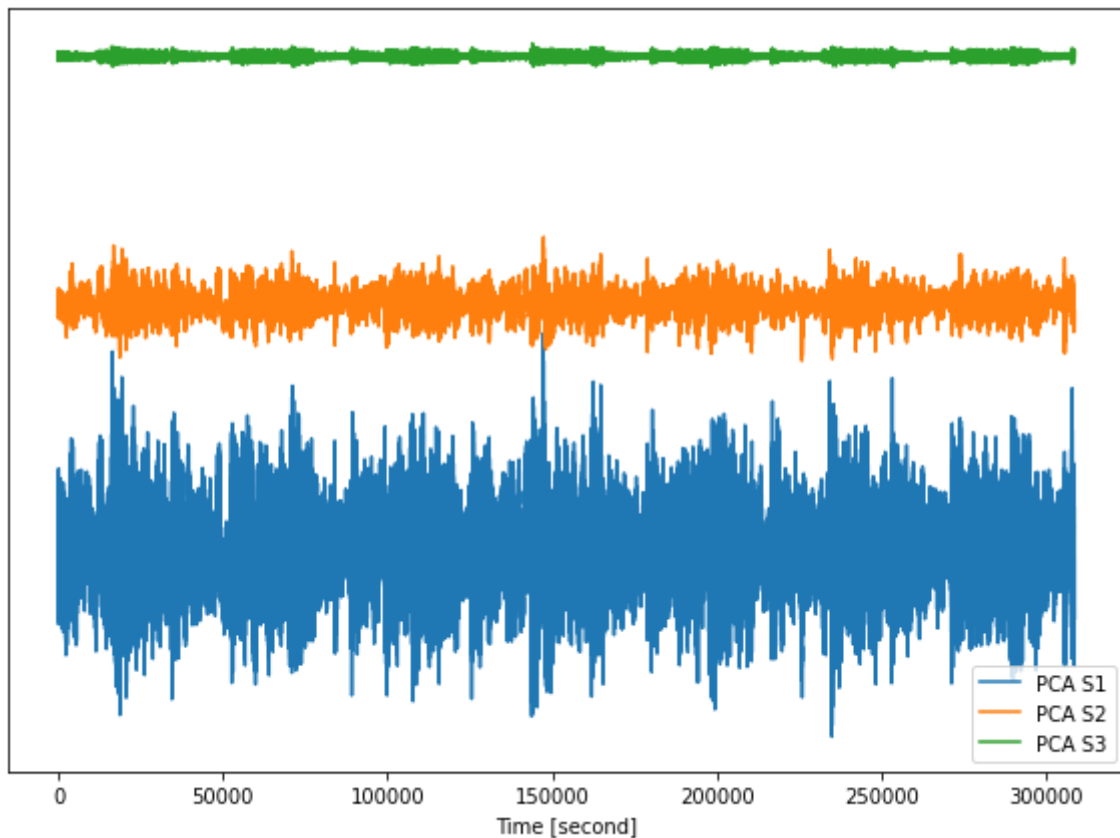
principal components. Plot the principal components, save them as wav files and play the sounds. How does it compares to Part 3 and 4?

```
In [27]: from sklearn.decomposition import PCA
```

```
In [28]: pca = PCA()

S_pca = pca.fit_transform(data)

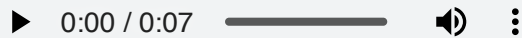
# plot
offset = 5e4
S_label = ['PCA S1', 'PCA S2', 'PCA S3']
plt.figure(figsize=(10, 7))
for i in range(3):
    plt.plot(S_pca[:, i] + i * offset, label=S_label[i])
plt.xlabel('Time [second]')
plt.yticks([])
plt.legend()
plt.show()
```



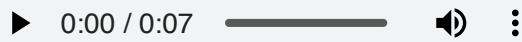
```
In [29]: # rewrite files
wavfile.write('signal_sound1.wav', fs, np.int16(Amp*S_pca[:,0]))
wavfile.write('signal_sound2.wav', fs, np.int16(Amp*S_pca[:,1]))
wavfile.write('signal_sound3.wav', fs, np.int16(Amp*S_pca[:,2]))

Audio('signal_sound1.wav')
```

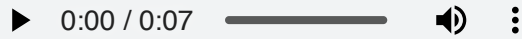
Out [29]:

In [30]: `Audio('signal_sound2.wav')`

Out [30]:

In [31]: `Audio('signal_sound3.wav')`

Out [31]:



Comparison

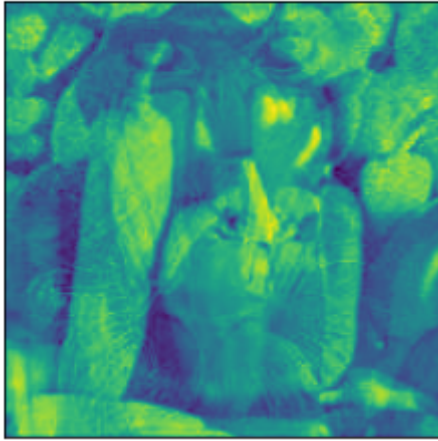
PCA is not able to reconstruct the audiofile, as with Part3, and sounds more like part 1.

Now let's take a look at another example. Suppose now we have some linearly mixed images, and we are going to find the original photos with ICA. (This example is from <https://github.com/vishwajeet97/Cocktail-Party-Problem>)

8. Load in photos, and plot them.

In [32]: `import matplotlib.image as mpimg`In [33]: `img1=mpimg.imread('/content/drive/My Drive/P188_288/P188_288_HW4/unos.jpg')
img2=mpimg.imread('/content/drive/My Drive/P188_288/P188_288_HW4/dos.jpg')
img3=mpimg.imread('/content/drive/My Drive/P188_288/P188_288_HW4/tres.jpg')`In [34]: `plt.figure()
plt.imshow(img1)
plt.xticks([])
plt.yticks([])
plt.title('unos.jpg')
plt.show()`

unos.jpg

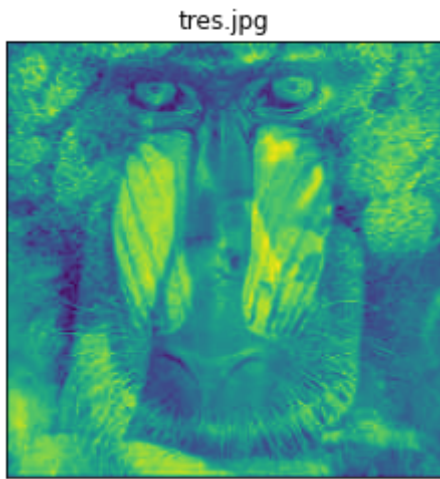


```
In [35]: plt.figure()
plt.imshow(img3)
plt.xticks([])
plt.yticks([])
plt.title('dos.jpg')
plt.show()
```

dos.jpg



```
In [36]: plt.figure()
plt.imshow(img2)
plt.xticks([])
plt.yticks([])
plt.title('tres.jpg')
plt.show()
```

The image is a 2D array. You will need to flatten the data for the following analysis.

9. Redo the analysis in part 3 and 4. Separate the original photos and plot them. Note that the sign of the signals recovered by ICA may not be correct, so you probably need to multiply the photos by -1.

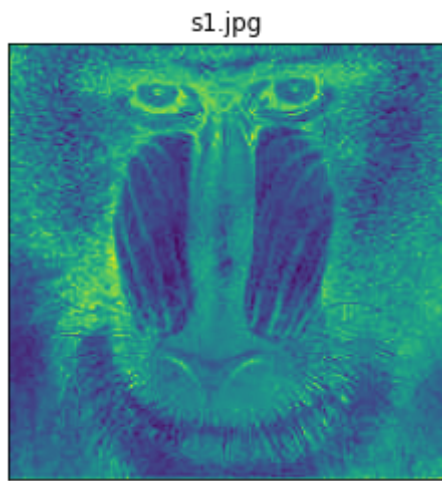
```
In [37]: img1_flat = np.ndarray.flatten(img1)
img2_flat = np.ndarray.flatten(img2)
img3_flat = np.ndarray.flatten(img3)

data = np.array([img1_flat, img2_flat, img3_flat]).T

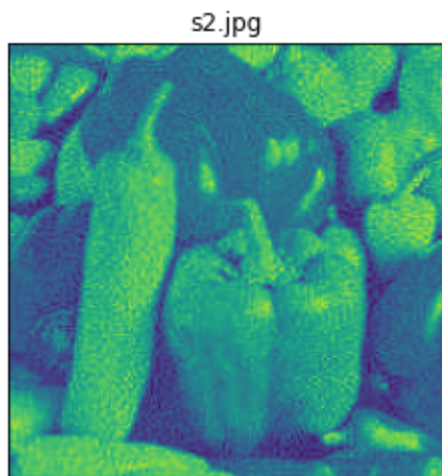
ica = FastICA()
s = ica.fit_transform(data)

s1 = s[:,0].reshape((200,200))
s2 = s[:,1].reshape((200,200))
s3 = s[:,2].reshape((200,200))
```

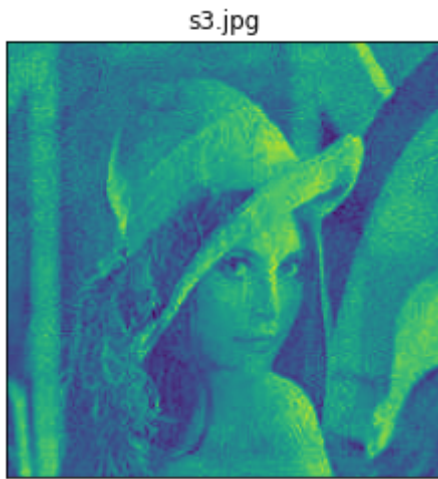
```
In [38]: plt.figure()
plt.imshow(s1 * -1)
plt.xticks([])
plt.yticks([])
plt.title('s1.jpg')
plt.show()
```



```
In [39]: plt.figure()
plt.imshow(s2)
plt.xticks([])
plt.yticks([])
plt.title('s2.jpg')
plt.show()
```



```
In [40]: plt.figure()
plt.imshow(-1 * s3)
plt.xticks([])
plt.yticks([])
plt.title('s3.jpg')
plt.show()
```



ICA algorithm tries to find the most non-Gaussian directions of given data. FastICA uses the KL divergence between the data and standard Gaussian (negentropy) to characterize the non-Gaussianity. Another way to measure non-Gaussianity is to use the Wasserstein distance between the data and standard Gaussian. In 1D, the Wasserstein distance, also called earth mover's distance, has a closed form solution using Cumulative Distribution Function (CDF). Below we provide you the code for doing ICA using Wasserstein distance. The code searches for the most non-Gaussian directions by maximizing the Wasserstein distance between the data and Gaussian.

10. Perform ICA on the mixed photos from Q9 (do the following steps)

1. Whitens the data with `sklearn.decomposition.PCA(whiten=True)`
2. Run ICA with Wasserstein distance: $A = \text{ICA_Wasserstein}(X_{\text{whiten}})$
3. Recover the Signal S from the whitened data and mixing matrix A . Note that $\mu = 0$ because of the whitening, and the shape of X of equation (1) is $(N_{\text{signal}}, N_{\text{sample}})$.
4. Plot the signals (original photos).

```
In [41]: import torch
import torch.optim as optim

def Percentile(input, percentiles):
    """
    Find the percentiles of a tensor along the last dimension.
    Adapted from https://github.com/aliutkus/torchpercentile/blob/master/torchpercentile.py
    """
    percentiles = percentiles.double()
    in_sorted, in_argsort = torch.sort(input, dim=-1)
    positions = percentiles * (input.shape[-1]-1) / 100
    floored = torch.floor(positions)
    ceiled = floored + 1
    ceiled[ceiled > input.shape[-1] - 1] = input.shape[-1] - 1
    weight_ceiled = positions - floored
    weight_floored = 1.0 - weight_ceiled
    d0 = in_sorted[..., floored.long()] * weight_floored
    d1 = in_sorted[..., ceiled.long()] * weight_ceiled
    result = d0+d1
```

```

    return result

def SlicedWassersteinDistanceG(x, pg, q, p, perdim=True):
    if q is None:
        px = torch.sort(x, dim=-1)[0]
    else:
        px = Percentile(x, q)

    if perdim:
        WD = torch.mean(torch.abs(px-pg) ** p)
    else:
        WD = torch.mean(torch.abs(px-pg) ** p, dim=-1)
    return WD

def SWD_prepare(Npercentile=100, device=torch.device("cuda:0"), gaussian=True):
    start = 50 / Npercentile
    end = 100-start
    q = torch.linspace(start, end, Npercentile, device=device)
    if gaussian:
        pg = 2**0.5 * torch.erfinv(2*q/100-1)
    return q, pg

def maxSWDdirection(x, x2='gaussian', n_component=None, maxiter=200, Npercentile=100):
    #if x2 is None, find the direction of max sliced Wasserstein distance between x and x2
    #if x2 is not None, find the direction of max sliced Wasserstein distance between x and x2

    if x2 != 'gaussian':
        assert x.shape[1] == x2.shape[1]
        if x2.shape[0] > x.shape[0]:
            x2 = x2[torch.randperm(x2.shape[0])][:x.shape[0]]
        elif x2.shape[0] < x.shape[0]:
            x = x[torch.randperm(x.shape[0])][:x2.shape[0]]

    ndim = x.shape[1]
    if n_component is None:
        n_component = ndim

    q = None
    if x2 == 'gaussian':
        if Npercentile is None:
            q, pg = SWD_prepare(len(x), device=x.device)
            q = None
        else:
            q, pg = SWD_prepare(Npercentile, device=x.device)
    elif Npercentile is not None:
        q = SWD_prepare(Npercentile, device=x.device, gaussian=False)

    #initialize w. algorithm from https://arxiv.org/pdf/math-ph/0609050.pdf
    wi = torch.randn(ndim, n_component, device=x.device)
    Q, R = torch.qr(wi)
    L = torch.sign(torch.diag(R))
    w = (Q * L).T

```

```

lr = 0.1
down_fac = 0.5
up_fac = 1.5
c = 0.5

#algorithm from http://noodle.med.yale.edu/~hdtag/notes/steifel_notes.pdf
#note that here w = X.T
#use backtracking line search
w1 = w.clone()
w.requires_grad_(True)
if x2 == 'gaussian':
    loss = -SlicedWassersteinDistanceG(w @ x.T, pg, q, p)
else:
    loss = -SlicedWassersteinDistance(w @ x.T, w @ x2.T, q, p)
loss1 = loss
for i in range(maxiter):
    GT = torch.autograd.grad(loss, w)[0]
    w.requires_grad_(False)
    WT = w.T @ GT - GT.T @ w
    e = - w @ WT #dw/dlr
    m = torch.sum(GT * e) #dloss/dlr

    lr /= down_fac
    while loss1 > loss + c*m*lr:
        lr *= down_fac
        if 2*n_component < ndim:
            UT = torch.cat((GT, w), dim=0).double()
            V = torch.cat((w.T, -GT.T), dim=1).double()
            w1 = (w.double() - lr * w.double() @ V @ torch.pinverse(torch.e
        else:
            w1 = (w.double() @ (torch.eye(ndim, dtype=torch.double, device=

    w1.requires_grad_(True)
    if x2 == 'gaussian':
        loss1 = -SlicedWassersteinDistanceG(w1 @ x.T, pg, q, p)
    else:
        loss1 = -SlicedWassersteinDistance(w1 @ x.T, w1 @ x2.T, q, p)

    if torch.max(torch.abs(w1-w)) < eps:
        w = w1
        break

    lr *= up_fac
    loss = loss1
    w = w1
if x2 == 'gaussian':
    WD = SlicedWassersteinDistanceG(w @ x.T, pg, q, p, perdim=False)
else:
    WD = SlicedWassersteinDistance(w @ x.T, w @ x2.T, q, p, perdim=False)
return w.T, WD**(1/p)

```

```

In [42]: def ICA_Wasserstein(x):

    A, WD = maxSWDdirection(torch.tensor(x, dtype=torch.float32))

    return A.detach().numpy()

```

1. Whitens the data with `sklearn.decomposition.PCA(whiten=True)`
2. Run ICA with Wasserstein distance: $A = \text{ICA_Wasserstein}(X_{\text{whiten}})$
3. Recover the Signal S from the whitened data and mixing matrix A . Note that $\mu = 0$ because of the whitening, and the shape of X of equation (1) is $(N_{\text{signal}}, N_{\text{sample}})$.
4. Plot the signals (original photos).

```
In [43]: data = np.array([img1_flat, img2_flat, img3_flat]).T
pca = PCA(whiten=True)
S = pca.fit_transform(data)
print('data: ', data.shape)
print('S: ', S.shape)
A = ICA_Wasserstein(S) # mixing matrix
print('A: ', S.shape)
data_ = (A @ S.T).T
print(data_.shape)
s1 = data_[ :,0].reshape((200, 200))
s2 = data_[ :,1].reshape((200, 200))
s3 = data_[ :,2].reshape((200, 200))
```

```
data: (40000, 3)
```

```
S: (40000, 3)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:74: UserWarning:
torch.qr is deprecated in favor of torch.linalg.qr and will be removed in a future PyTorch release.
```

```
The boolean parameter 'some' has been replaced with a string parameter 'mode'.
Q, R = torch.qr(A, some)
should be replaced with
Q, R = torch.linalg.qr(A, 'reduced' if some else 'complete') (Triggered internally at
../aten/src/ATen/native/BatchLinearAlgebra.cpp:2491.)
```

```
A: (40000, 3)
```

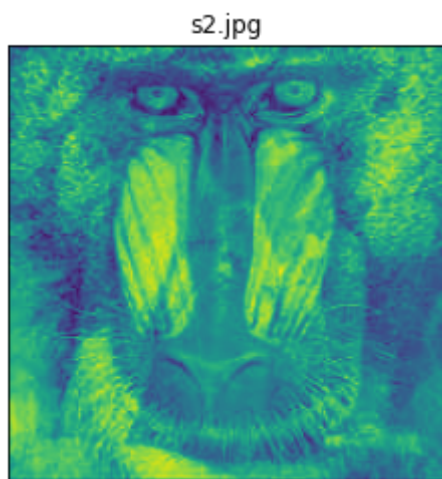
```
(40000, 3)
```

```
In [45]: plt.figure()
plt.imshow(s1)
plt.xticks([])
plt.yticks([])
plt.title('s1.jpg')
plt.show()
```

s1.jpg



```
In [46]: plt.figure()  
plt.imshow(s2)  
plt.xticks([])  
plt.yticks([])  
plt.title('s2.jpg')  
plt.show()
```



```
In [47]: plt.figure()  
plt.imshow(s3)  
plt.xticks([])  
plt.yticks([])  
plt.title('s3.jpg')  
plt.show()
```

