bayesian-analysis (/github/christianhbye/bayesian-analysis/tree/main)
/ homeworks (/github/christianhbye/bayesian-analysis/tree/main/homeworks)

Open in Colab
(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/homeworks/HW4_288.ipynb)

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.

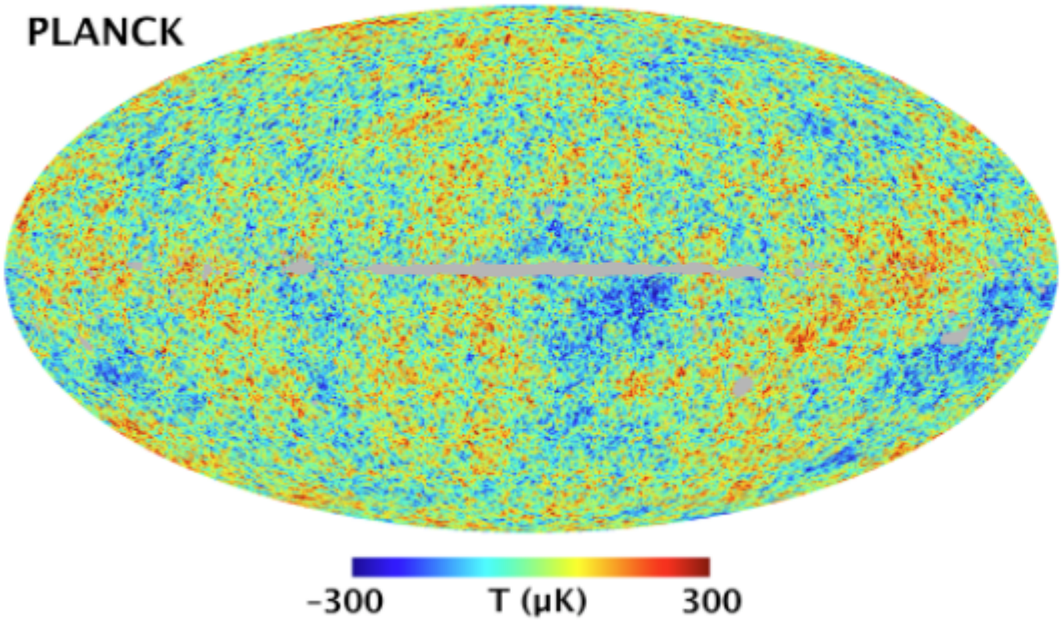
In [2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

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 flunctuations in the spectrum are very small. Averaging ocer 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 ~ 30 μK 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):

$$\frac{\delta T}{T}(\theta, \phi) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \mathrm{a}_{lm} \mathrm{Y}_{lm}(\theta, \phi)$$

In flat space, we can do a Fourier transform of a function $f(x)$ as $\sum_k \mathrm{a}_k \mathrm{e}^{ikx}$ where k is the wavenumber, and $|\mathrm{a}_k|$ 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.

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

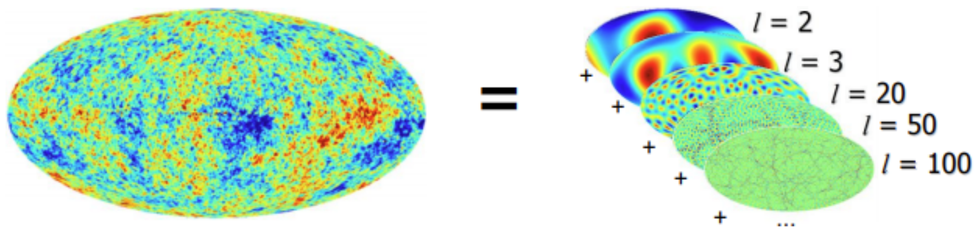
The above expression is specific to the temperature fluctuations, but we can also do a similar analysis for the polarization map of the CMB. (The CMB is polarized because it was scattered off of free electrons during decoupling.) We decompose the polarization pattern in the sky into a curl-free "E-mode"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



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

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, $\vec{\theta} = [\theta_1, \theta_2, \ldots, \theta_6] = [H_0, \Omega_b h^2, \Omega_c h^2, n_s, A_s, \tau]$. ($H_0$ = 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 (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 (https://arxiv.org/pdf/1502.01589v3.pdf).) Then, we can estimate the Fisher matrix between two parameters θ_i and θ_j as:

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

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 (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 $\sigma(H_0), \sigma(\Omega_b h^2), \ldots, \sigma(\tau)$ (corresponding to the square root of the diagonal entries of the covariance matrix). Print the results.

In [3]:

```
"""
NOTE: we change this code slightly to not unpack every spectrum
"""

all_derivs = {}
all_errs = {}
all_models = {}

# Load data

# Best-fit values of the cosmological parameters from https://ar
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
# 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
sigma_l_Planck_EE = data[:,2]
all_errs["EE"] = sigma_l_Planck_EE

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

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


# Model power spectra given theta_best_Planck (calculated at the

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

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

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

```
# Derivative of the power spectrum given theta_best_Planck (calc

# Derivative of D_l^EE with respect to six parameters
# ([theta1, theta2, theta3, theta4, theta5, theta6] = [H_0, \Ome
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4
all_derivs["EE"] = data[:, 1:7]
# 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]

# Derivative of D_l^TT with respect to six parameters
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4
all_derivs["TT"] = data[:, 1:7]
# 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]

# Derivative of D_l^TE with respect to six parameters
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW4
all_derivs["TE"] = data[:, 1:7]
# 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]
```

In [32]:
```
N_PARAMS = 6  # number of parameters in model

# Construct empty Fisher matrix before summing
fisher_summand = np.empty((N_PARAMS, N_PARAMS, 3, len(l)))

for i in range(N_PARAMS):  # i index of Fisher matrix
  for j in range(i+1):  # j index (don't have to do j>i by matri
    for k, mode in enumerate(["TT", "EE", "TE"]):  # TT, EE, TE
      prod = all_derivs[mode][:, i] * all_derivs[mode][:, j]
      prod /= all_errs[mode] **2
      fisher_summand[i, j, k] = prod
      fisher_summand[j, i, k] = prod

fisher = fisher_summand.sum(axis=(-1, -2))
cov = np.linalg.inv(fisher)

params = [
    "H_0", "Omega_b h^2", "Omega_c h^2", "n_s", "A_s", "tau"
]
print("Errors on parameters")
for i, p in enumerate(params):
  print(f"Error on {p}: {np.sqrt(cov[i, i]):.3g}")
```

```
Errors on parameters
Error on H_0: 0.591
Error on Omega_b h^2: 0.000137
Error on Omega_c h^2: 0.00133
Error on n_s: 0.00367
Error on A_s: 4.94e-11
Error on tau: 0.0116
```

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 (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 ith 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 x 2 - covariance matrix from this.) - and obtain a 2×2 covariance matrix:

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

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:

$$\phi = \frac{1}{2}\mathrm{arctan}\Big(\frac{2 \cdot \mathrm{Cov}(\theta_i, \theta_j)}{\sigma(\theta_i)^2 - \sigma(\theta_j)^2}\Big) = \mathrm{arctan}(\frac{\overrightarrow{v_1}(y)}{\overrightarrow{v_1}(x)})$$

where $\overrightarrow{v_1}$ 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 alrady 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).

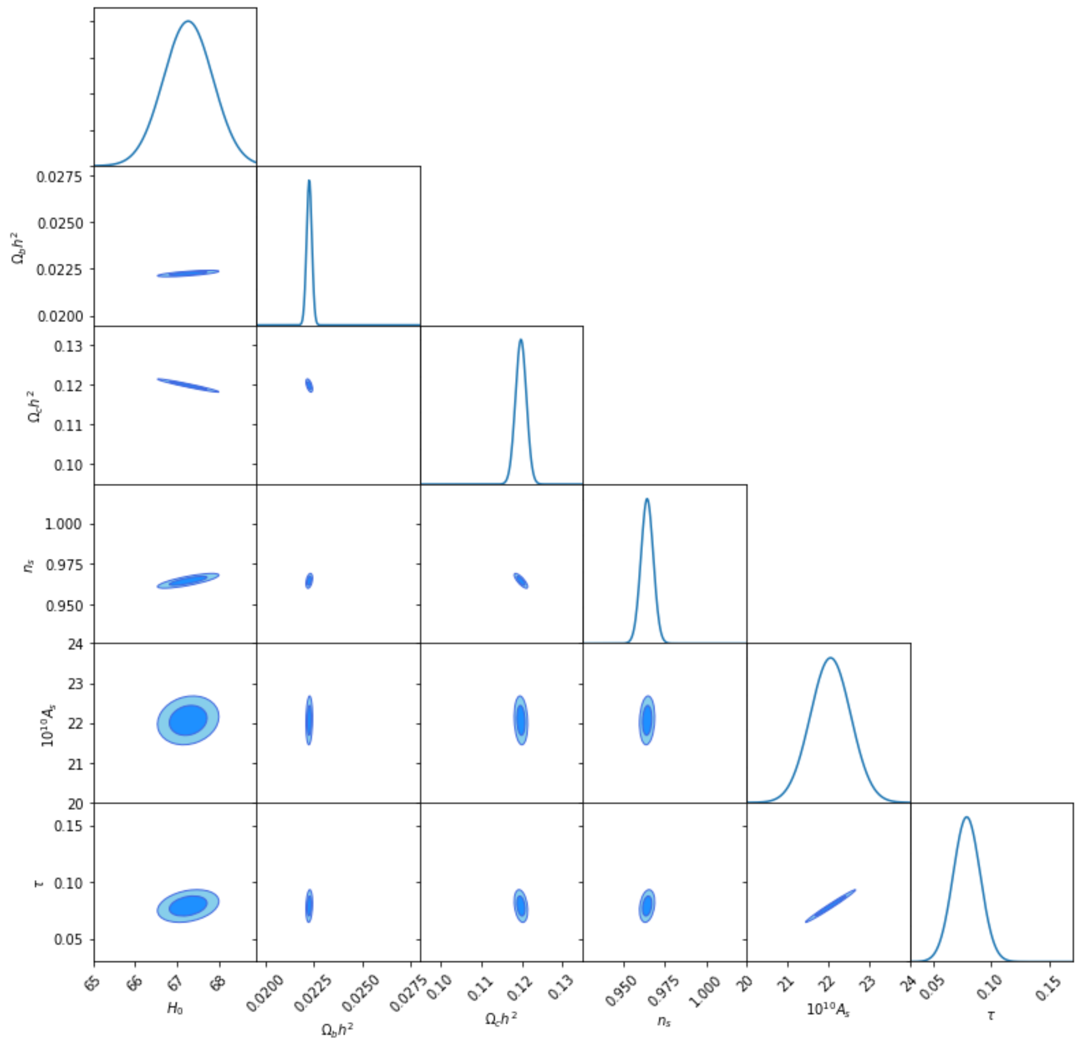
In [130]:

```
"""
Note: We rescale A_s by 10^10 since it has a very small value. T
plotting a bit nicer. We can do this since none of the probabili
shapes are changed (we've only done linear transformations to ar
probability distributions)
"""

AS_RESCALE = 1e10

from matplotlib.patches import Ellipse

def gauss(x, mean, variance):
  """
  Normal distribution given mean and variance
  """
  a = 1 / np.sqrt(2 * np.pi * variance)
  z = (x - mean)**2 / variance
  return  a * np.exp(-z / 2)

params_latex = [
    "$H_0$",
    "$\\Omega_b h^2$",
    "$\\Omega_c h^2$",
    "$n_s$",
    "$10^{10}A_s$",
    "$\\tau$",
]


def plot_1d(ax, mean, variance, xmin, xmax, color=None, label=No
  """
  Helper function to plot 1d normal distribution
  """
  xgrid = np.linspace(xmin, xmax, num=200)
  y = gauss(xgrid, mean, variance)
  y /= y.max()  # divide out amplitude in plot since units are n
  ax.plot(xgrid, y, color=color, label=label)
  ax.set_xlim(xmin, xmax)
  ax.set_ylim(0., 1.1)

def confidence(cov_matrix, interval):
  """
  Construct confidence ellipse from cov matrix and confidence in
  """
  eigvec, eigval, u = np.linalg.svd(cov_matrix)
  # Semimajor axis (diameter)
  semimaj = np.sqrt(eigval[0])
  # Semiminor axis (diameter)
  semimin = np.sqrt(eigval[1])
  # theta
  theta = np.arctan2(eigvec[0, 1], eigvec[0, 0])

  if interval == 0.68:
    factor = 1.52
  elif interval == 0.95:
    factor = 2.48

  semimaj *= factor
  semimin *= factor
  return semimaj, semimin, theta


def corner(best_params, cov_matrix, bounds, cov_matrix2=None):
  """
  Make corner plot given a covariance matrix. Optionally, constr
  a 2nd covariance matrix can be overplotted.

  Parameters
  ----------
  best_params : dict
    The best fit parameters (mean of normal distribution). Key i
    name, value is the best fit value.

  cov_matrix : np.ndarray
    Covariance matrix.

  cov_matrix2 : np.ndarray
    2nd covariance matrix.
  """
  NROWS = NCOLS = len(params) + 1
  fig = plt.figure(figsize=(15, 15))
  axs = []  # diagonal axes

  # plot the 1d constraints along the diagonal
  for i, par in enumerate(best_params):
    if i == 0:
      ax = fig.add_subplot(NROWS, NCOLS, i * (NCOLS + 1) + 1)
    else:
      ax = fig.add_subplot(NROWS, NCOLS, i * (NCOLS + 1) + 1, sh
    axs.append(ax)

    mean = best_params[par]
    var = cov_matrix[i, i]

    if par == "As":
      mean *= AS_RESCALE  # see note on top of cell
      var *= AS_RESCALE ** 2

    # parameter bounds
    xmin, xmax = bounds[par]
    plot_1d(ax, mean, var, xmin, xmax, color="C0", label="PLANCK

    if cov_matrix2 is not None:
      var2 = cov_matrix2[i, i]
      if par == "As":
        var2 *= AS_RESCALE ** 2
      plot_1d(ax, mean, var2, xmin, xmax, color="C3", label="No

      if i == 0:
        ax.legend(loc="upper right", bbox_to_anchor=(3., 1.), fo

    # labels and ticks
    ax.locator_params(axis="x", nbins=4, min_n_ticks=3)
    plt.setp(ax.get_yticklabels(), visible=False)
    if i == len(best_params)-1:
      ax.set_xlabel(params_latex[i])
      visible = True
    else:
      visible = False
    plt.setp(ax.get_xticklabels(), visible=visible, rotation=45)


  # 2d constraints
  covMs = [cov_matrix]
  if cov_matrix2 is not None:
    covMs.append(cov_matrix2)

  # ellipse colors
  ELL_FACE_COLORS = ["dodgerblue", "skyblue", "firebrick", "ligh
  ELL_EDGE_COLORS = ["royalblue", "red"]

  for i in range(NROWS-1):
    for j in range(i+1, NCOLS-1):

      ax = fig.add_subplot(NROWS, NCOLS, j * NCOLS + i + 1, shar
      # set limits on y axes to be same as x axis of the same pa
      ypar = list(best_params.keys())[j]
      ymin, ymax = bounds[ypar]
      ax.set_ylim(ymin, ymax)
      ax.locator_params(axis="y", nbins=4, min_n_ticks=3)

      # best parameter values
      p1 = list(best_params.values())[i]
      p2 = list(best_params.values())[j]

      if i == 4:  # As, see note on top of cell
        p1 *= AS_RESCALE
      if j == 4:
        p2 *= AS_RESCALE

      for k, cov in enumerate(covMs):  # loop over cov matrices
        # 2x2 cov matrix
        cov_22 = np.zeros((2, 2))
        cov_22[0, 0] = cov[i, i]
        cov_22[0, 1] = cov_22[1, 0] = cov[i, j]
        cov_22[1, 1] = cov[j, j]

        if i == 4:  # As, see note on top of cell
          cov_22[0, 0] *= AS_RESCALE ** 2
          cov_22[0, 1] *= AS_RESCALE
          cov_22[1, 0] *= AS_RESCALE

        if j == 4:
          cov_22[1, 1] *= AS_RESCALE ** 2
          cov_22[0, 1] *= AS_RESCALE
          cov_22[1, 0] *= AS_RESCALE

        semimaj, semimin, theta = confidence(cov_22, .68)
        fc = ELL_FACE_COLORS[2*k]
        ec = ELL_EDGE_COLORS[k]
        ell = Ellipse(
            xy=[p1, p2],
            width=semimaj,
            height=semimin,
            angle=theta*180/np.pi,
            facecolor=fc,
            edgecolor=ec,
        )
```

```
            semimaj, semimin, theta = confidence(cov_22, .95)
            fc = ELL_FACE_COLORS[2*k+1]
            ell2 = Ellipse(
                xy=[p1, p2],
                width=semimaj,
                height=semimin,
                angle=theta*180/np.pi,
                facecolor=fc,
                edgecolor=ec,
            )

            # add ellipses
            ax.add_patch(ell2)
            ax.add_patch(ell)

        if i == 0:
            ax.set_ylabel(params_latex[j])
            visible = True
        else:
            visible = False
        plt.setp(ax.get_yticklabels(), visible=visible)

        if j == NCOLS-2:
            ax.set_xlabel(params_latex[i])
            visible = True
        else:
            visible = False
        plt.setp(ax.get_xticklabels(), visible=visible, rotation=4

plt.subplots_adjust(wspace=0, hspace=0)
plt.show()
```

In [131]:

```
best_params = {
    "H0": H0,
    "ombh2": ombh2,
    "omch2": omch2,
    "ns": ns,
    "As": As,
    "tau": tau
}

# parameter bounds to plot, aiming to match fig6 in the paper
BOUNDS = {
    "H0": (65., 68.9),
    "ombh2": (0.0195, 0.028),
    "omch2": (0.095, 0.135),
    "ns": (0.926, 1.024),
    "As": (20, 24),  # the paper plots the log
    "tau": (0.03, 0.17),
}

corner(best_params, cov, BOUNDS)
```



Now, assume that we have an ideal, zero-noise CMB survey with $N_l = 0$. However, we are still instrinsically 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 instrinsic error (called "cosmic variance") in our estimate of C_l. So we approximate that

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

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

.

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 cosmologial parameters?)

The formula for the error is given as:

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

Now, we assume $f_{\text{sky}} = 1$ and $N_l^k = 0$. Thus:

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

In [132]:

```
def sigma(l, delta_l, Dlk):
  """
  Compute (sigma_kl)**2 according to the formula above
  """
  s = 2 * Dlk ** 2
  s /= (2 * l  + 1) * delta_l
  return s

# compute sigma kl
sigma_kl_sq = np.empty((3, len(l)))  # rows are k (mode), cols a
lbins = np.concatenate((np.ones(28), np.full(l.size-28, 30)))  #

for k, mode in enumerate(["TT", "EE", "TE"]):
  Dlk = all_models[mode]
  sigma_kl_sq[k] = sigma(l, lbins, Dlk)
```

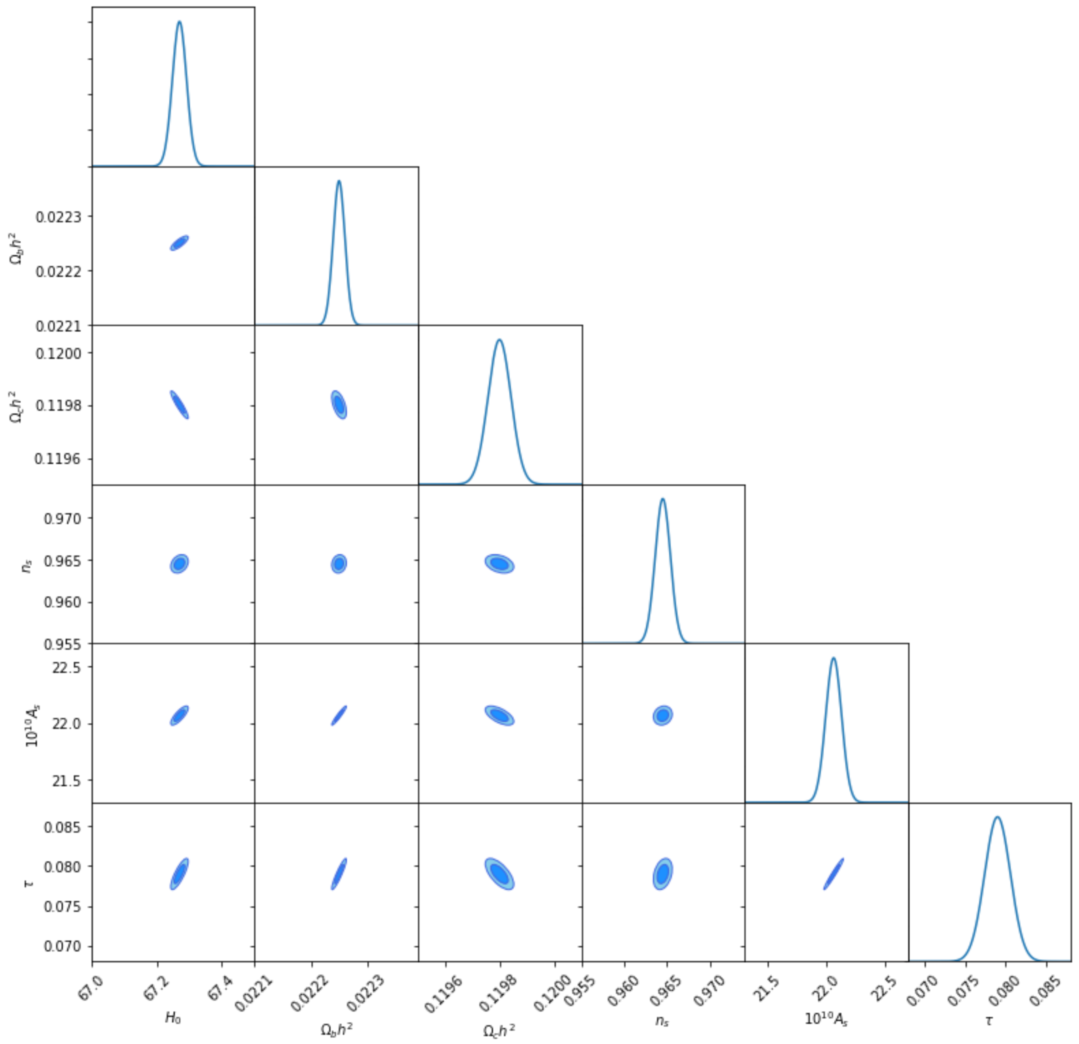
In [133]:

```
fisher_summand = np.empty((N_PARAMS, N_PARAMS, 3, len(l)))

for i in range(N_PARAMS):
  for j in range(i+1):
    for k, mode in enumerate(["TT", "EE", "TE"]):
      prod = all_derivs[mode][:, i] * all_derivs[mode][:, j]
      prod /= sigma_kl_sq[k]  # this changed from above
      fisher_summand[i, j, k] = prod
      fisher_summand[j, i, k] = prod

fisher = fisher_summand.sum(axis=(-1, -2))
cov_nonoise = np.linalg.inv(fisher)

print("Errors on parameters")
for i, p in enumerate(params):
  print(f"Error on {p}: {np.sqrt(cov_nonoise[i, i]):.3g}")
```

```
Errors on parameters
Error on H_0: 0.0216
Error on Omega_b h^2: 1.04e-05
Error on Omega_c h^2: 4.22e-05
Error on n_s: 0.000882
Error on A_s: 6.76e-12
Error on tau: 0.00158
```

In [134]:

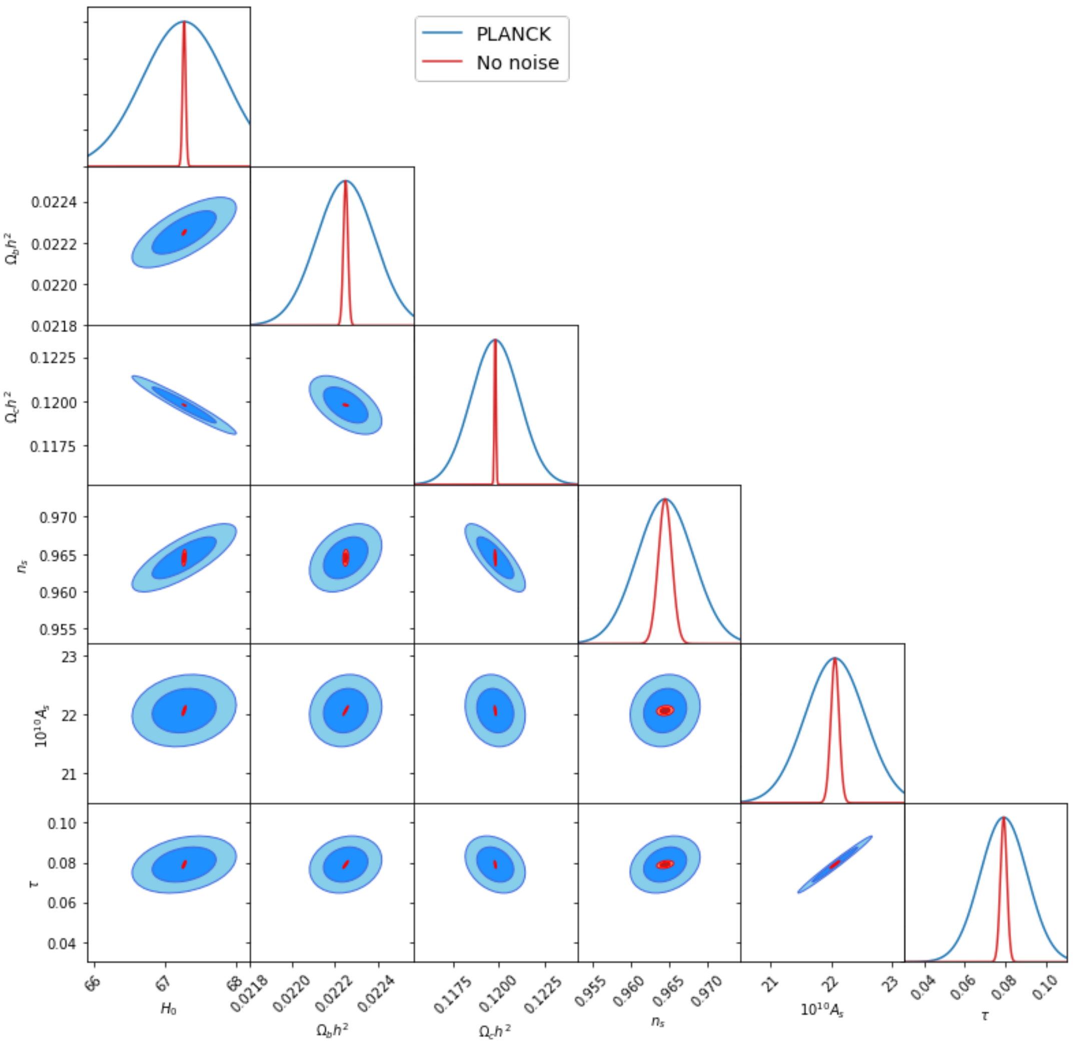
```
# smaller parameter bounds in this case since parameters are mor
bounds2 = {
    "H0": (67., 67.5),
    "ombh2": (0.0221, 0.02239),
    "omch2": (0.1195, 0.1201),
    "ns": (0.955, 0.974),
    "As": (21.3, 22.7),
    "tau": (0.068, 0.088),
}

corner(best_params, cov_nonoise, bounds2)
```



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 constrains 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 [135]:

```
# new bounds that allow us to see constraints from both experime
bounds3 = {
    "H0": (65.9, 68.2),
    "ombh2": (0.0218, 0.02257),
    "omch2": (0.1153, 0.1243),
    "ns": (0.953, 0.9743),
    "As": (20.5, 23.2),
    "tau": (0.03, 0.11),
}

corner(best_params, cov, bounds3, cov_nonoise)
```



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:

Clearly, with zero noise, we are able to constrain the parameters much more. This is evident in the 1d-plots (along the diagonal), where the curves are much narrower, and in the 2d-plots, where the ellipses are of smaller area.

Perhaps the most interesting result is that the no-noise survey sometimes achieve differently shaped 2d-constraints compared to Planck. For example in the H0-ns plot, the red ellipses seem different from the blue ones.

Compared to Figure 6, we expect our blue curves (with Planck noise) to be similar to the blue curves there since those incorporate both the autocorrelations and the cross-correlation. These are in good agreement; note however that they have $100\theta_{\mathrm{MC}}$ instead of H_0, that they have the order of n_s and A_s switched compared to us, and that they show the logarithm of A_s. Taking these differences into account, there are no major discrepancies between the results.

The blue curve results in Figure 6 correspond Colum 4 in Table 3. For our no-noise survey, the best fit parameters are the same (by construction), but the 1σ are significantly smaller.

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 (https://www.cs.helsinki.fi/u/ahyvarin/papers/NN00new.pdf)

In this problem, we take the mixed sounds and images, and apply ICA tn 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.)

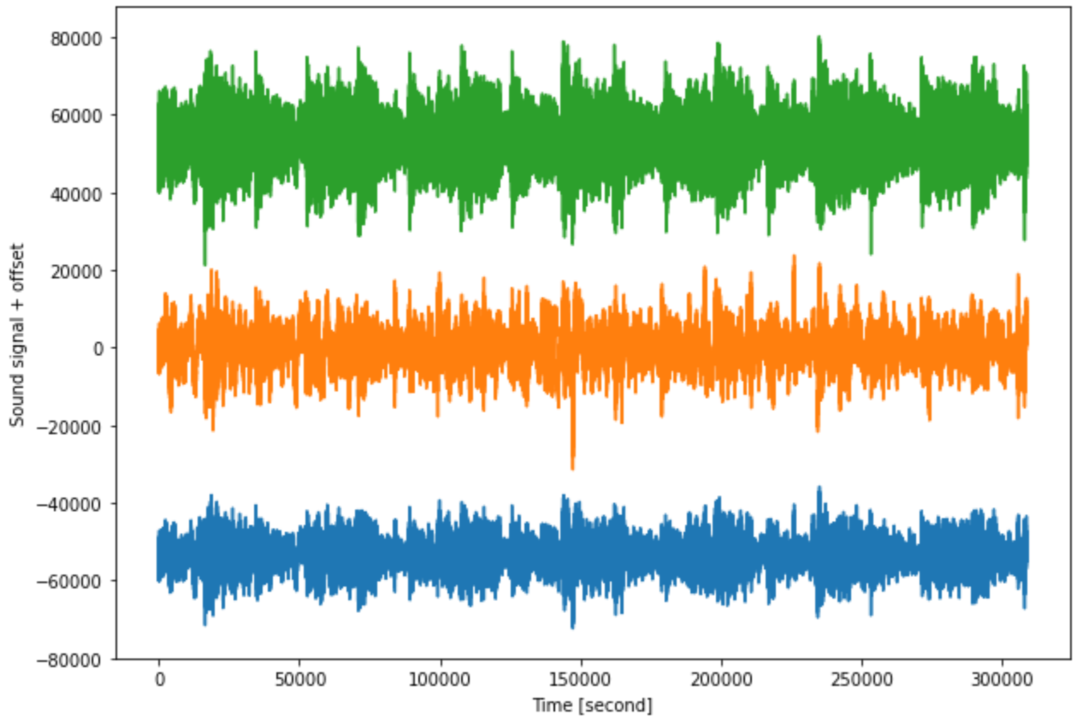
In [136]:

```
from scipy.io import wavfile
```

In [137]:

```
fs, data1 = wavfile.read('/content/drive/My Drive/P188_288/P188_
fs, data2 = wavfile.read('/content/drive/My Drive/P188_288/P188_
fs, data3 = wavfile.read('/content/drive/My Drive/P188_288/P188_
#fs is the sample rate, i.e., how many data points in one second

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

In [138]:
```
plt.figure(figsize = (10,7))

offset = 2*data.max()
plt.plot(data + offset * np.array([-1, 0, 1])[None])

plt.xlabel('Time [second]')
plt.ylabel('Sound signal + offset')
plt.show()
```



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

In [139]:
```
from IPython.display import Audio
```

In [140]:
```
Audio('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sound1.
```

Out[140]:

0:00 / 0:07

In [141]:
```
Audio('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sound2.
```

Out[141]:

0:00 / 0:07

In [142]:
```
Audio('/content/drive/My Drive/P188_288/P188_288_HW4/mix_sound3.
```

Out[142]:

0:00 / 0:07

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 distrance to the 3 sources. In orther words,

$$\mathbf{X} = \mathbf{AS} + \mu \tag{1}$$

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

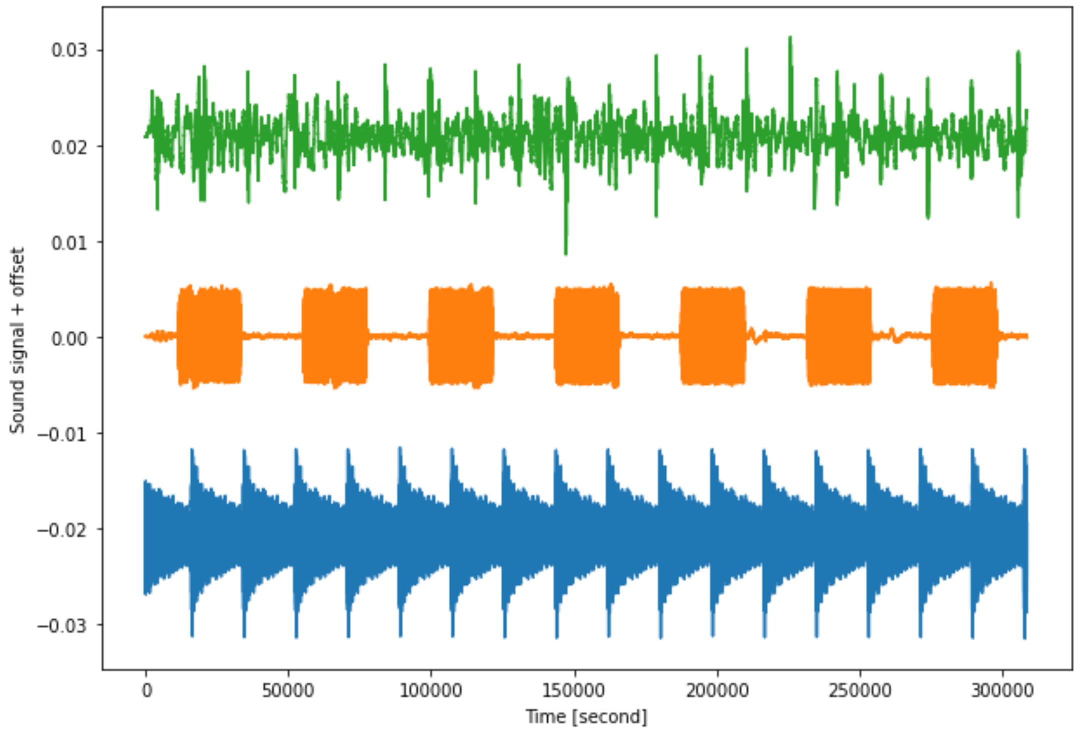
S = ica.fit_transform(data)

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

In [143]:
```
from sklearn.decomposition import FastICA
```

In [144]:
```
ica = FastICA(algorithm="parallel")
S = ica.fit_transform(data)
```

In [145]:
```
plt.figure(figsize = (10,7))

offset = 2*S.max()
plt.plot(S + offset * np.array([-1, 0, 1])[None])

plt.xlabel('Time [second]')
plt.ylabel('Sound signal + offset')
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 [146]:
```
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 [147]:
```
Audio('signal_sound1.wav')
```

Out[147]:

0:00 / 0:07

In [148]:
```
Audio('signal_sound2.wav')
```

Out[148]:

0:00 / 0:07

In [149]:
```
Audio('signal_sound3.wav')
```

Out[149]:

0:00 / 0:07

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

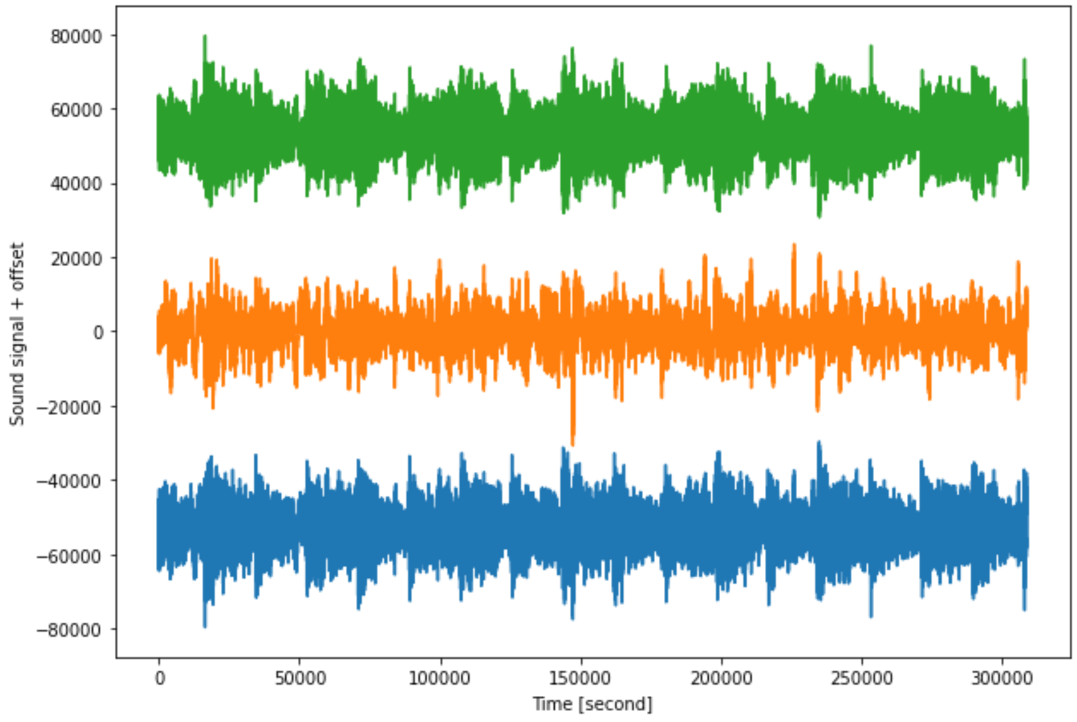
In [150]:
```
X = ica.mixing_ @ S.T + ica.mean_[:, None]
X = X.T  # transpose
```

In [151]:
```
print (np.allclose(X, 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 [152]:
```
ica = FastICA(whiten=False)
S = ica.fit_transform(data)

# plot
plt.figure(figsize = (10,7))
offset = 2*S.max()
plt.plot(S + offset * np.array([-1, 0, 1])[None])
plt.xlabel('Time [second]')
plt.ylabel('Sound signal + offset')
plt.show()


# save
wavfile.write('signal_sound1_unwhiten.wav', fs, np.int16(S[:,0])
wavfile.write('signal_sound2_unwhiten.wav', fs, np.int16(S[:,1])
wavfile.write('signal_sound3_unwhiten.wav', fs, np.int16(S[:,2])
```



In [153]:
```
Audio('signal_sound1_unwhiten.wav')
```
Out[153]:

0:00 / 0:07

In [154]:
```
Audio('signal_sound2_unwhiten.wav')
```
Out[154]:

0:00 / 0:07

In [155]:
```
Audio('signal_sound3_unwhiten.wav')
```
Out[155]:

0:00 / 0:07

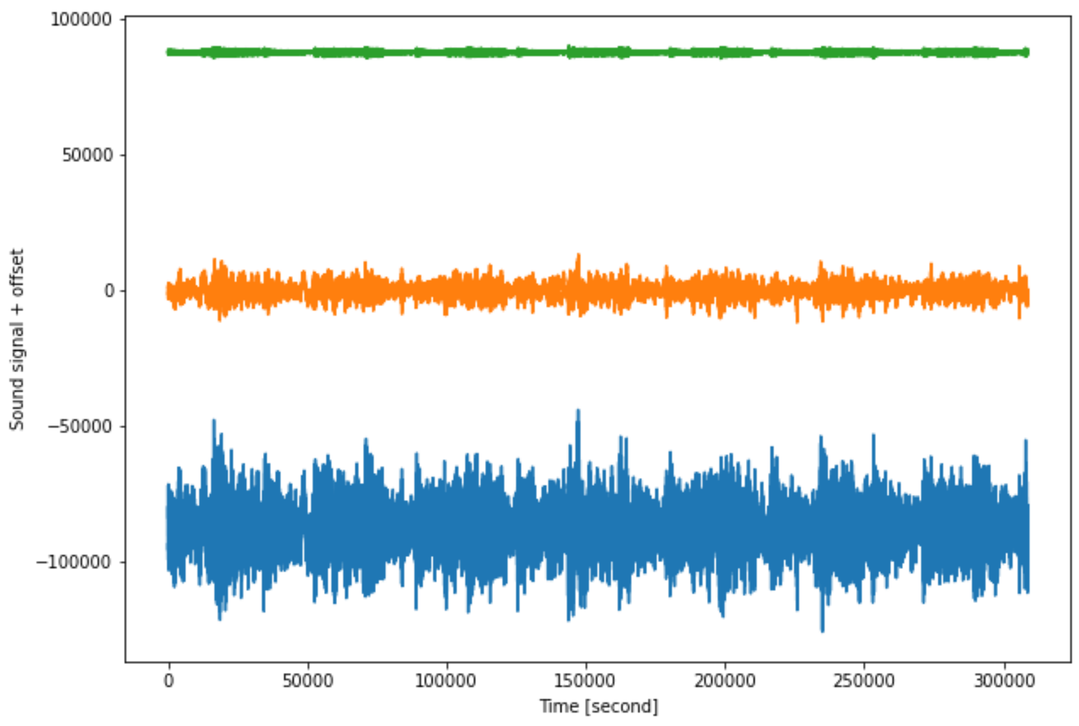
7. The principal companent analysis (PCA) also tries to interpret the underlying *structure of the data by decomposing the data into linear combinations of the principal components. Now let's apply PCA to the sounds and see if the signals are cleanly separated in the principal components. Plot the principal components, save them as wav files and play the sounds. How does it compares to Part 3 and 4?*

In [156]:

```
from sklearn.decomposition import PCA
```

In [157]:

```
pca = PCA(n_components=3)
S = pca.fit_transform(data)

# plot
plt.figure(figsize = (10,7))
offset = 2*S.max()
plt.plot(S + offset * np.array([-1, 0, 1])[None])
plt.xlabel('Time [second]')
plt.ylabel('Sound signal + offset')
plt.show()


# save
Amp = 1e6
wavfile.write('signal_sound1_pca.wav', fs, np.int16(Amp*S[:,0]))
wavfile.write('signal_sound2_pca.wav', fs, np.int16(Amp*S[:,1]))
wavfile.write('signal_sound3_pca.wav', fs, np.int16(Amp*S[:,2]))
```



In [158]:

```
Audio('signal_sound1_pca.wav')
```

Out[158]:

0:00 / 0:07

In [159]:

```
Audio('signal_sound2_pca.wav')
```

Out[159]:

0:00 / 0:07

In [160]:
```
Audio('signal_sound3_pca.wav')
```

Out[160]:

```
0:00 / 0:07
```

PCA is similar to unwhitened ICA. We cannot seperate the sgnals very well, and the resulting components are noisy.

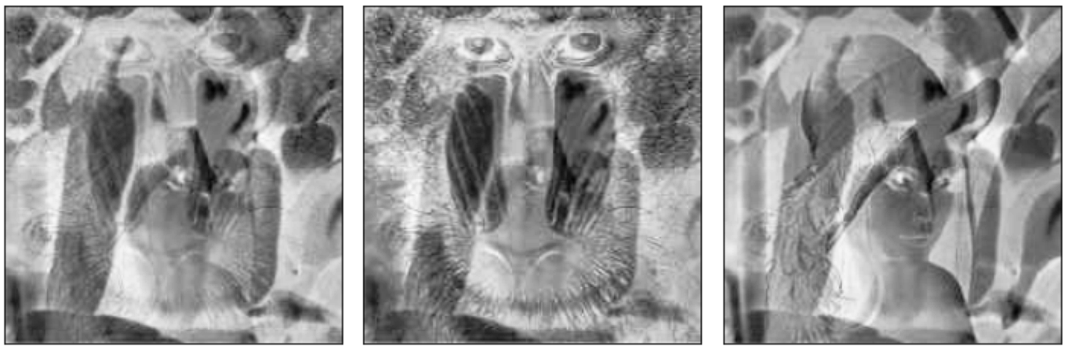
Now let's take a look at another example. Suppose now we have some linearly mixed images, and we are going to find the original photos with ICA. (This example is from https://github.com/vishwajeet97/Cocktail-Party-Problem (https://github.com/vishwajeet97/Cocktail-Party-Problem))

8. Load in photos, and plot them.

In [161]:
```
import matplotlib.image as mpimg
```

In [162]:
```
img1=mpimg.imread('/content/drive/My Drive/P188_288/P188_288_HW4
img2=mpimg.imread('/content/drive/My Drive/P188_288/P188_288_HW4
img3=mpimg.imread('/content/drive/My Drive/P188_288/P188_288_HW4
```

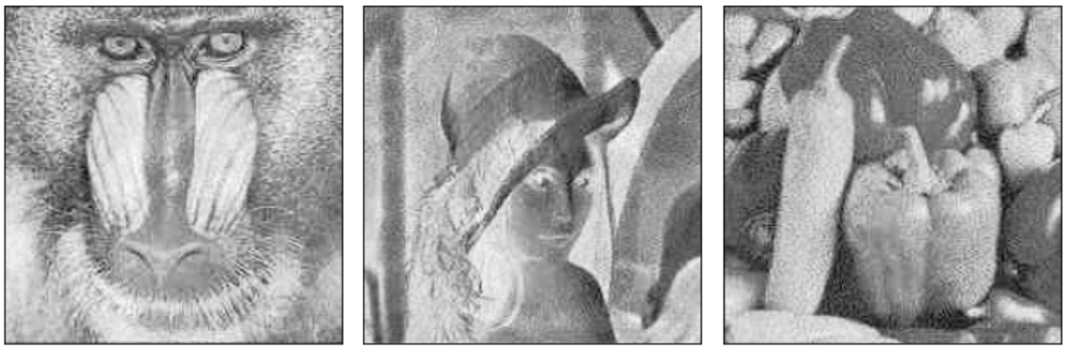
In [163]:
```
fig, axs = plt.subplots(figsize=(10, 10), ncols=3, sharex=True,
for i, img in enumerate([img1, img2, img3]):
  axs[i].imshow(img, cmap="Greys")
plt.setp(axs, xticks=[], yticks=[])
plt.tight_layout()
plt.show()
```



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 [164]:
```
img_flat = np.array([img1.ravel(), img2.ravel(), img3.ravel()]).

ica = FastICA(algorithm="parallel", whiten=True)
S = ica.fit_transform(img_flat) * -1

fig, axs = plt.subplots(figsize=(10, 10), ncols=3, sharex=True,
for i in range(3):
  axs[i].imshow(S[:, i].reshape(*img1.shape), cmap="Greys")
plt.setp(axs, xticks=[], yticks=[])
plt.tight_layout()
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 charactrize 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 = ICA*Wasserstein($X*{\mathrm{whiten}}$)
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 (Nsignal, Nsample).
4. Plot the signals (original photos).

In [165]:

```
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/blo
    """
    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"),
    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=

    #if x2 is None, find the direction of max sliced Wasserstein
    #if x2 is not None, find the direction of max sliced Wassers

    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=F


    #initialize w. algorithm from https://arxiv.org/pdf/math-ph/
    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/stei
    #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,
    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.p
            else:
                w1 = (w.double() @ (torch.eye(ndim, dtype=torch.

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

        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, perdi
    else:
        WD = SlicedWassersteinDistance(w @ x.T, w @ x2.T, q, p,
    return w.T, WD**(1/p)
```

In [166]:

```
def ICA_Wasserstein(x):

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

    return A.detach().numpy()
```

In [169]:

```
pca = PCA(whiten=True)
X_whiten = pca.fit_transform(img_flat)
A = ICA_Wasserstein(X_whiten)
S =  np.linalg.inv(A) @ X_whiten.T  # X = AS --> S = A^-1 X
S = S.T
```

In [170]:

```
fig, axs = plt.subplots(figsize=(10, 10), ncols=3, sharex=True,
for i in range(3):
  axs[i].imshow(S[:, i].reshape(*img1.shape), cmap="Greys")
plt.setp(axs, xticks=[], yticks=[])
plt.tight_layout()
plt.show()
```