bayesian-analysis (/github/christianhbye/bayesian-analysis/tree/main)
/ projects (/github/christianhbye/bayesian-analysis/tree/main/projects)
/ project2 (/github/christianhbye/bayesian-analysis/tree/main/projects/project2)

(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/projects/Project1_p2_288.ipynb)

Project 1 - Part 2

Optimization, Markov chain Monte Carlo, Bayesfast

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.

Write your partner's name here (if you have one).

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 - Supernova Cosmology Project

In this project, we use a compilation of supernovae data to show that the expansion of the universe is accelerating, and hence it contains dark energy. This is the Nobel prize winning research in 2011 (https://www.nobelprize.org/nobel_prizes/physics/laureates/2011/ (https://www.nobelprize.org/nobel_prizes/physics/laureates/2011/)), and Saul Perlmutter, a professor of physics at Berkeley, shared a prize in 2011 for this discovery.

"The expansion history of the universe can be determined quite easily, using as a “standard candle” any distinguishable class of astronomical objects of known intrinsic brightness that can be identified over a wide distance range. As the light from such beacons travels to Earth through an expanding universe, the cosmic expansion stretches not only the distances between galaxy clusters, but also the very wavelengths of the photons en route. By the time the light reaches us, the spectral wavelength λ has thus been redshifted by precisely the same incremental factor $z = \Delta\lambda/\lambda$ by which the cosmos has been stretched in the time interval since the light left its source. The recorded redshift and brightness of each such object thus provide a measurement of the total integrated expansion of the universe since the time the light was emitted. A collection of such measurements, over a sufficient range of distances, would yield an entire historical record of the universe’s expansion." (Saul Perlmutter, http://supernova.lbl.gov/PhysicsTodayArticle.pdf (http://supernova.lbl.gov/PhysicsTodayArticle.pdf)).

Supernovae emerge as extremely promising candidates for measuring the cosmic expansion. Type I Supernovae arises from the collapse of white dwarf stars when the Chandrasekhar limit is reached. Such nuclear chain reaction occurs in the same way and at the same mass, the brightness of these supernovae are always the same. The relationship between the apparent brightness and distance of supernovae depend on the contents and curvature of the universe.

We can infer the "luminosity distance" D_L from measuring the inferred brightness of a supernova of luminosity L. Assuming a naive Euclidean approach, if the supernova is observed to have flux F, then the area over which the flux is distributed is a sphere radius D_L, and hence

$$F = \frac{L}{4\pi D_L^2}.$$

In Big Bang cosmology, D_L is given by:

$$D_L = \frac{\chi(a)}{a}$$

where a is the scale factor ($\frac{\lambda_0}{\lambda} = 1 + z = \frac{a_0}{a}$, and the quantity with the subscript 0 means the value at present. Note that $a_0 = 1, z_0 = 0$.), and χ is the comoving distance, the distance between two objects as would be measured instantaneously today. For a photon, $cdt = a(t)d\chi$, so $\chi(t) = c\int_t^{t_0} \frac{dt'}{a(t')}$. We can write this in terms of a Hubble factor ($H(t) = \frac{1}{a}\frac{da}{dt}$), which tells you the expansion rate:
$\chi(a) = c\int_a^1 \frac{da'}{a'^2 H(a')} = c\int_0^z \frac{dz'}{H(z')}$. (change of variable using $a = \frac{1}{1+z}$.)

Using the Friedmann equation (which basically solves Einstein's equations for a homogenous and isotropic universe), we can write H^2 in terms of the mass density ρ of the components in the universe:
$H^2(z) = H_0^2[\Omega_m(1+z)^3 + (1-\Omega_m)(1+z)^2]$.

Ω is the density parameter; it is the ratio of the observed density of matter and energy in the universe (ρ) to the critical density ρ_c at which the universe would halt is expansion. So Ω_0 (again, the subscript 0 means the value at the present) is the total mass and energy density of the universe today, and consequently $\Omega_0 = \Omega_m$ (matter density parameter today; remember we obtained the best-fit value of this parameter in Project 1?) = $\Omega_{\text{baryonoic matter}}$ + $\Omega_{\text{dark matter}}$. If $\Omega_0 < 1$, the universe will continue to expand forever. If $\Omega_0 > 1$, the expansion will stop eventually and the universe will start to recollapse. If $\Omega_0 = 1$, then the universe is flat and contains enough matter to halt the expansion but not enough to recollapse it. So it will continue expanding, but gradually slowing down all the time, finally running out of steam only in the infinite future. Even including dark matter in this calculation, cosmologists found that all the matters in the universe only amounts to about a quarter of the required critical mass, suggesting a continuously expanding universe with deceleration. Then, using all this, we can write the luminosity distance in terms of the density parameters:

$$D_L = \frac{\chi(a)}{a} = c(1+z)\int_0^z \frac{dz'}{H(z')} = c(1+z)\int_0^z \frac{dz'}{H_0[\Omega_m(1+z')^3 + (1-\Omega_m)}$$

$$= \frac{2997.92458}{h}(1+z)\int_0^z \frac{dz'}{[\Omega_m(1+z')^3 + (1-\Omega_m)(1+z')^2]^{1/2}} \; [unit\ of\ Mpc$$

where $H_0 = 100 \cdot h \; [km \cdot s^{-1} Mpc^{-1}]$.

Fluxes can be expressed in magnitudes m, where $m = -2.5 \cdot \log_{10} F$ + const. The distance modulus is $\mu = m - M$ (M is the absolute magnitude, the value of m if the supernova is at a distance 10pc. Then, we have:

$$\mu = 25 + 5 \cdot \log_{10}\Big(D_L \; [in\ the\ unit\ of\ Mpc]\Big)$$

In this assignment, we use the SCP Union2.1 Supernova (SN) Ia compilation. (http://supernova.lbl.gov/union/ (http://supernova.lbl.gov/union/))

First, load the measured data: z (redshift), μ (distance modulus), $\sigma(\mu)$ (error on

In [52]:
```
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_Pro
# z
z_data = data[:,0]
# mu
mu_data = data[:,1]
# error on mu (sigma(mu))
mu_err_data = data[:,2]
```

1. Plot the measured distance modulus as a function of redshift with errorbars. Then, assume three different scenarios: $\Omega_m = 0, 0.3, 1.$

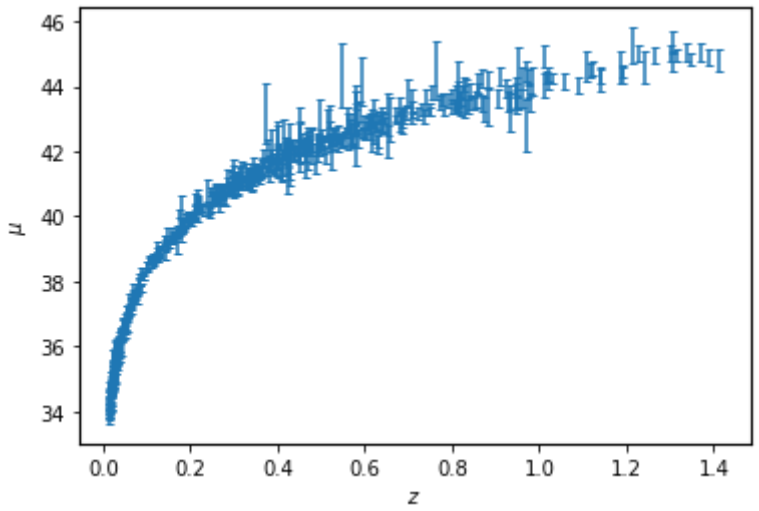
Remember:

$$D_L = \frac{2997.92458}{h}(1+z)\int_0^z \frac{dz'}{[\Omega_m(1+z')^3 + (1-\Omega_m)(1+z')^2]^{1/2}}$$

$$\mu = 25 + 5 \cdot \log_{10}(D_L)$$

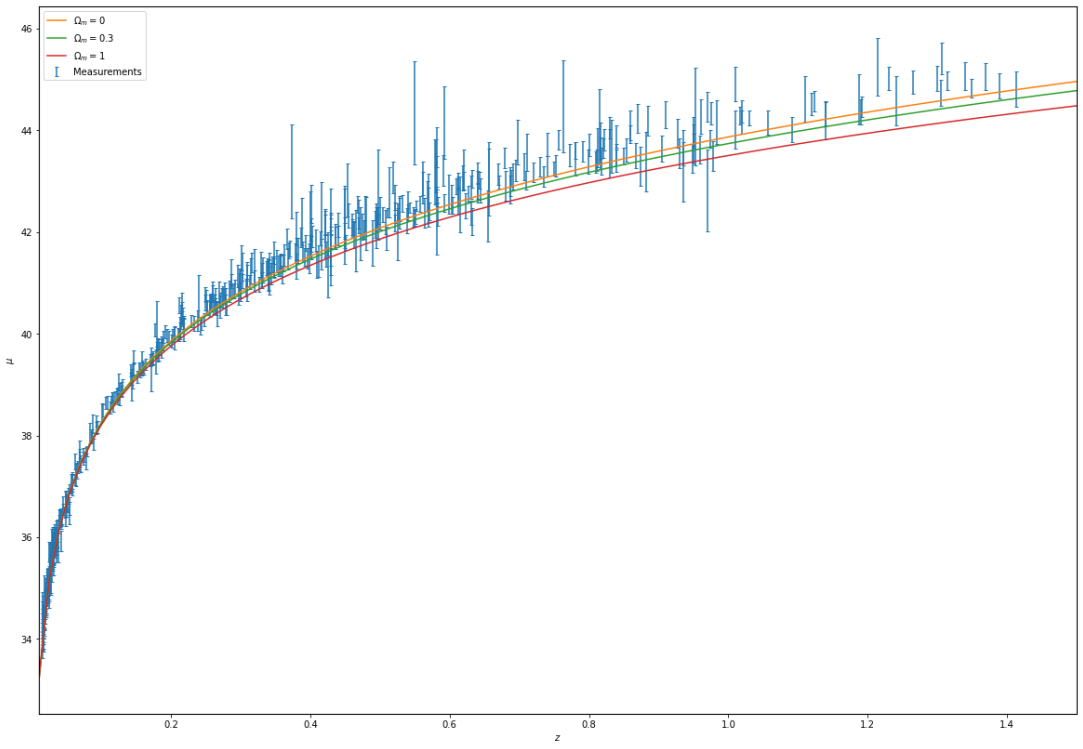
Now, plot three curves of μ *as a function of* z *for* $\Omega_m = 0, 0.3, 1$ *on top of the measured data (Calculate* D_L *using quad. For now, assume* $h = 0.7$*.) How do they fit?*

In [53]:
```
plt.figure()
plt.errorbar(z_data, mu_data, yerr=mu_err_data, fmt="none", caps
plt.xlabel("$z$")
plt.ylabel("$\mu$")
plt.show()
```



In [54]:

```
def _dL_integrand(z, omega_m):
  x = 1 + z
  return 1 / np.sqrt(omega_m * x**3 + (1-omega_m) * x**2)

def dL(z, omega_m, h=0.7):
  int_arr = np.empty_like(z)
  for i, zval in enumerate(z):
    int_arr[i] = quad(_dL_integrand, 0, zval, args=(omega_m))[0]
  prefactor = 2997.92458 / h * (1+z)
  return prefactor * int_arr

def mu(dist_lum):
  return 25 + 5 * np.log10(dist_lum)

zgrid = np.linspace(0.01, 1.5, num=100)
mu0 = mu(dL(zgrid, 0))
mu03 = mu(dL(zgrid, 0.3))
mu1 = mu(dL(zgrid, 1))

plt.figure(figsize = (20,14))

plt.errorbar(
  z_data, mu_data, yerr=mu_err_data, fmt="none", capsize=2, labe
)
plt.plot(zgrid, mu0, label="$\\Omega_m = 0$")
plt.plot(zgrid, mu03, label="$\\Omega_m = 0.3$")
plt.plot(zgrid, mu1, label="$\\Omega_m = 1$")

plt.legend()
plt.xlim(0.01, 1.5)
plt.xlabel('$z$')
plt.ylabel('$\mu$')
plt.show()
```



The theoretical predictions are consistently below the measured data for z > 0.3. The residuals are certainly not Gaussian random, indicating a bias in the model.

You should find that the measured data do not fit well to all three scenarios. "The high-redshift supernovae are fainter than would be expected even for an empty cosmos (corresponding to $\Omega_m = 0$)." So what's wrong?

"If these data are correct, the obvious implication is that the simplest cosmological model must be too simple. The next simplest model might be one that Einstein entertained for a time. Believing the universe to be static, he tentatively introduced into the equations of general relativity an expansionary term he called the “cosmological constant” (Λ) that would compete against gravitational collapse. After Hubble’s discovery of the cosmic expansion, Einstein famously rejected Λ as his “greatest blunder.” In later years, Λ came to be identified with the zero-point vacuum energy of all quantum fields. It turns out that invoking a cosmological constant allows us to fit the supernova data quite well." (Saul Perlmutter, https://www.nobelprize.org/nobel_prizes/physics/laureates/2011/ (https://www.nobelprize.org/nobel_prizes/physics/laureates/2011/))

So in short, the data indicates that faint supernovae are further away from the earth than had been theoretically expected. The expansion rate of the universe is increasing indeed. It seems that some mysterious material (which we call "dark energy") is causing such antigravity effects. The cosmological constant, Λ, the value of the energy density of the vacuum of space is widely accepted as a leading candidate of dark energy.

Now let us add a general form of dark energy to our model.

$$H^2(z) = H_0^2[\Omega_m(1+z)^3 + \Omega_{DE}(1+z)^{3(1+w)} + (1-\Omega_m-\Omega_{DE})(1+z)^2].$$

w is the dark energy equation of state, which is the ratio of its pressure to its energy density. $w = -1$ for the cosmological constant Λ.

$\Omega_0 = \Omega_m$ (matter density parameter today) + Ω_{DE} (dark energy density parameter today), and

$$D_L = \frac{\chi(a)}{a} = c(1+z)\int_0^z \frac{dz'}{H(z')} = c(1+z)\int_0^z \frac{}{H_0[\Omega_m(1+z')^3 + \Omega_{DE}(1+}$$

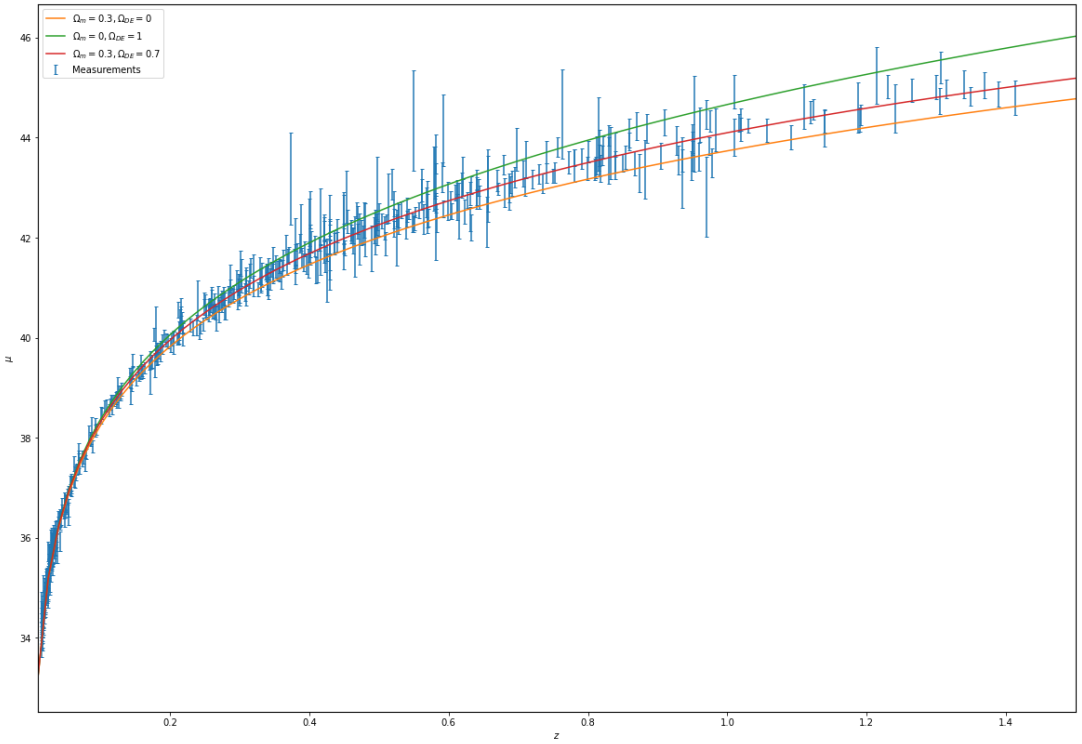
$$= \frac{2997.92458}{h}(1+z)\int_0^z \frac{dz'}{[\Omega_m(1+z')^3 + \Omega_{DE}(1+z')^{3(1+w)} + (1-\Omega_m-\Omega_D}$$

where $H_0 = 100 \cdot h\ [km \cdot s^{-1}Mpc^{-1}]$.

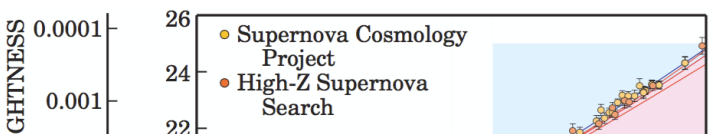
2. Now assume three different scenarios: ($\Omega_m = 0.3, \Omega_{DE} = 0$), ($\Omega_m = 0, \Omega_{DE} = 1, w = -1$), and ($\Omega_m = 0.3, \Omega_{DE} = 0.7, w = -1$). Again, plot three curves of μ as a function of z on top of data (assume $h = 0.7$)

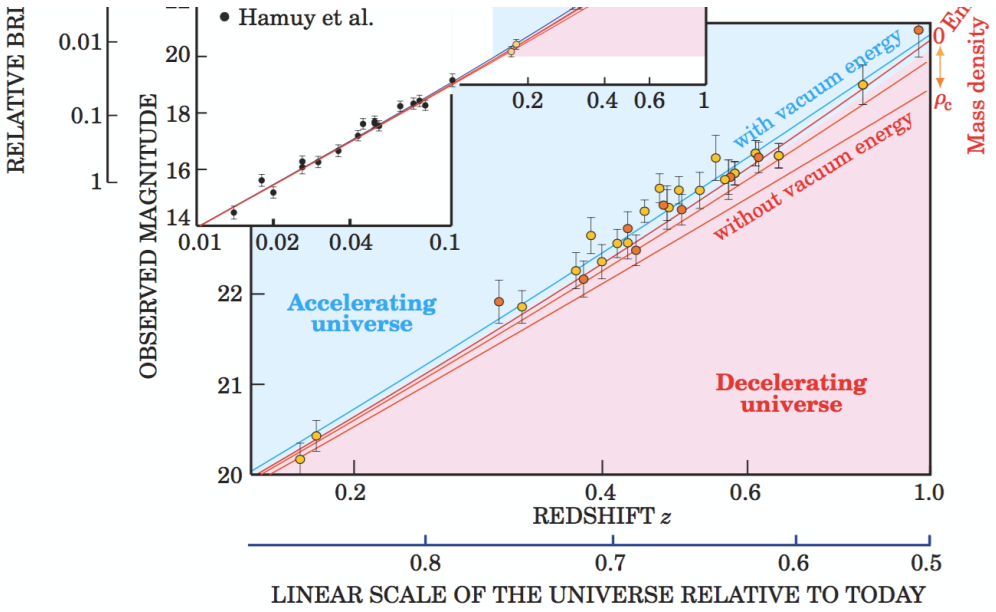
In [55]:
```
def _dL_integrand_DE(z, omega_m, omega_DE, w=-1):
  x = 1 + z
  den = omega_m * x**3 + omega_DE * x**(3+3*w) + (1-omega_m-omeg
  return 1 / np.sqrt(den)

def dL_DE(z, omega_m, omega_DE, w=-1, h=0.7):
  int_arr = np.empty_like(z)
  for i, zval in enumerate(z):
    int_arr[i] = quad(_dL_integrand_DE, 0, zval, args=(omega_m,
  prefactor = 2997.92458 / h * (1+z)
  return prefactor * int_arr
```

In [56]:
```
mu_DE_1 = mu(dL_DE(zgrid, 0.3, 0))
mu_DE_2 = mu(dL_DE(zgrid, 0, 1))
mu_DE_3 = mu(dL_DE(zgrid, 0.3, 0.7))

plt.figure(figsize = (20,14))

plt.errorbar(
  z_data, mu_data, yerr=mu_err_data, fmt="none", capsize=2, labe
)
plt.plot(zgrid, mu_DE_1, label="$\\Omega_m = 0.3, \\Omega_{DE} =
plt.plot(zgrid, mu_DE_2, label="$\\Omega_m = 0, \\Omega_{DE} = 1
plt.plot(zgrid, mu_DE_3, label="$\\Omega_m = 0.3, \\Omega_{DE} =
plt.legend()
plt.xlim(0.01, 1.5)
plt.xlabel('$z$')
plt.ylabel('$\mu$')
plt.show()
```



You basically reproduced the below figure!





You should see that $\Omega_m = 0.3$ and $\Omega_m = 0.7$ fits the data best. In combination with the CMB data, this shows that about 70% of the total energy density is vacuum energy and 30% is mass.

Now, with measurements of the distance modulus μ, use Bayesian analysis to estimate the cosmological parameters.

let us assume that the universe is flat (which is a fair assumption since the CMB measurements indicate that the universe has no large-scale curvature).
$\Omega_0 = \Omega_m + \Omega_{DE} = 1$. Then, we do not need to worry about the curvature term:

$$D_L = \frac{\chi(a)}{a} = c(1+z)\int_0^z \frac{dz'}{H(z')} = c(1+z)\int_0^z \frac{dz'}{H_0[\Omega_m(1+z')^3 + (1-\Omega_m)}$$

$$= \frac{2997.92458}{h}(1+z)\int_0^z \frac{dz'}{[\Omega_m(1+z')^3 + (1-\Omega_m)(1+z')^{3(1+w)}]^{1/2}} \; [unit\ of$$

where $H_0 = 100 \cdot h \; [km \cdot s^{-1} Mpc^{-1}]$.

Assuming that errors are Gaussian (can be justified by averaging over large numbers of SN; central limit theorem), we calculate the likelihood L as:

$$L \propto \exp\Big(-\frac{1}{2}\sum_{i=1}^{N_{SN}} \frac{[\mu_{i,\,data}(z_i) - \mu_{i,\,model}(z_i, \Omega_m, w)]^2}{\sigma(\mu_i)^2}\Big)$$

where $z_i, \mu_i, \sigma(\mu_i)$ are from the measurements, and we compute μ_{model} as a function of z, Ω_m, w.

Next, write an MCMC code using the **Metropolis algorithm**.

Now, assume a more general form of dark energy. (Do not fix w to -1; add w as a parameter.)

In the flat universe,

$$D_L = \frac{2997.92458}{h}(1+z)\int_0^z \frac{dz'}{[\Omega_m(1+z')^3 + (1-\Omega_m)(1+z')^{3(1+w)}]^{1/2}} \ [uni$$

where $H_0 = 100 \cdot h \ [km \cdot s^{-1} Mpc^{-1}]$. Here, we fix $h = 0.7$.

We calculate the likelihood L as:

$$\ln(L) \approx -\frac{1}{2}\sum_{i=1}^{N_{\text{SN}}} \frac{[\mu_{i,\,data}(z_i) - \mu_{i,\,model}(z_i, \Omega_m, w)]^2}{\sigma(\mu_i)^2} = -\frac{1}{2}\sum_{i=1}^{N_{\text{SN}}} \frac{\Delta\mu_i^2}{\sigma(\mu_i)^2}$$

where

$$\mu_{i,\,model}(z_i, \Omega_m, w) = 25 + 5 \cdot \log_{10}(D_{L,\,i})$$

$$D_{L,\,i} = \frac{2997.92458}{0.7}(1+z_i)\int_0^{z_i} \frac{dz'}{[\Omega_m(1+z')^3 + (1-\Omega_m)(1+z')^{3(1+w)}]^{1/2}}$$

3. Run the MCMC code to estimate w and Ω_m. Plot 1-d posterior of w and Ω_m as well as 2-d posterior (i.e. plot the chain in two-dimensional parameter space. Make sure that the chain has converged (you can change nsamples, nburn).

Hint:

Set the length of MCMC chains to be 15,000 (or even more if you think that the chain has not yet converged.) In the end, you should throw away the first 20% of the chain as burn-in. (20% is an arbitrary number. You can plot the chain and estimate the burn-in period.)

Then, set the random initial point in the parameter space (w, Ω_m): let w be negative and Ω_m be positive and draw a random number using np.random.uniform(). Set initial likelihood to low value (e.g. -1.e100) so that next point is accepted.

Now, draw a new sample starting from this random initial point. Here we assume that the proposal distribution is Gaussian with arbitrary width: in this problem, we assume that $\sigma = 0.01$ (This determines how far you propose jumps.) for distributions for both w, Ω_m.

For example, say that you start with $(w, \Omega_m) = (-0.3, 0.7)$. Then, draw a new sample of w from a Gaussian with $\mu = -0.3, \sigma = 0.01$ and a new sample of Ω_m from a Gaussian with $\mu = -0.7, \sigma = 0.01$.

Now, evaluate the log likelihood value of this new point.

If the value has gone up, accept the point.

Otherwise, accept it with probability given by ratio of likelihoods: Draw a random number from a uniform distribution between 0 and 1 (α = np.random.uniform()). If the ratio $ln(\frac{L_{new}}{L_{old}})$ is greater than $ln(\alpha)$ (i.e. $\frac{L_{new}}{L_{old}} > \alpha$), then accept it. Otherwise, reject it and stay at your old point.

Repeat this 15,000 times (the length of chain) and plot the distributions of (w, Ω_m).

In [57]:
```
# Import data
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_Pro
# z
z_data = data[:,0]
# mu
mu_data = data[:,1]
# error on mu (sigma(mu))
mu_err_data = data[:,2]
```

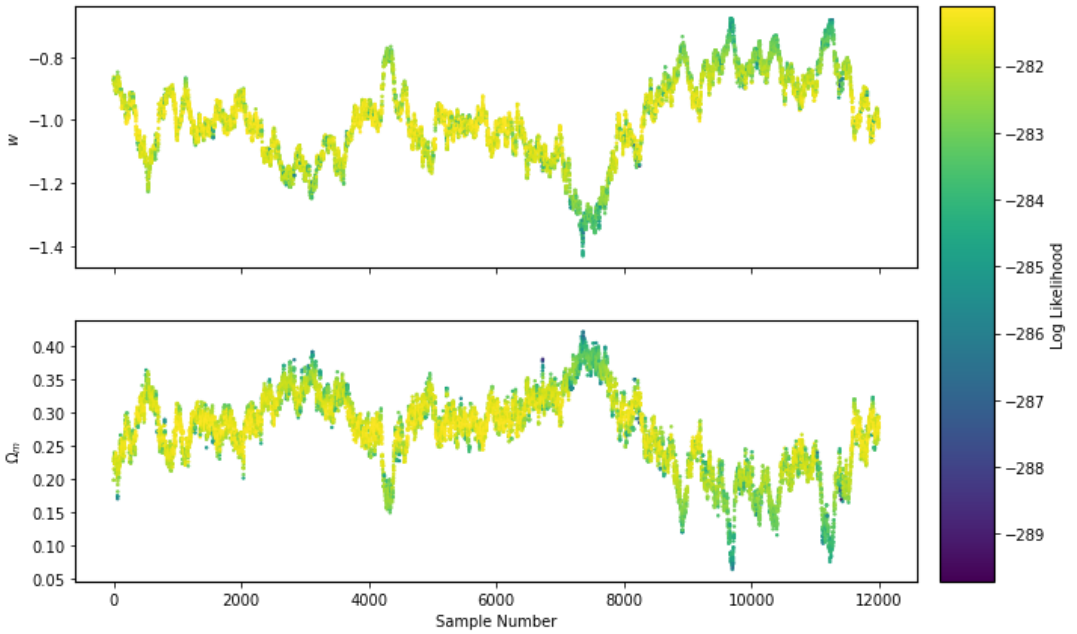
In [58]:
```
# Define the likelihood function:

def lnL(Omegam, w):

    # Treat unphysical regions by setting likelihood to (almost)
    if(Omegam<=0 or w>=0):
      lnL = -1.e100
    else:
      # Compute difference with theory mu at redshifts of the SN
      Omegal = 1 - Omegam  # flat universe
      DL = dL_DE(z_data, Omegam, Omegal, w=w, h=0.7)
      mu_model = 25 + 5 * np.log10(DL)
      delta_mu = mu_data - mu_model

      # Compute ln(likelihood) assuming gaussian errors
      lnL = - 1/2 * np.sum(delta_mu**2 / mu_err_data **2)

    return lnL
```

In [59]:
```
#from itertools import chain
# Draw new proposed samples from a proposal distribution, centre
# Accept or reject, and colour points according to ln(likelihood

def MH_chain(chain_len=int(15e3), burn_in=0.2, sigma=1e-2):
  """
  Run MH algorithm. Here: sigma is the standard deviation of the
  distribution (which is Gaussian).
  """
  w_arr = np.empty(chain_len)
  omega_m_arr = np.empty(chain_len)
  log_lh_arr = np.empty(chain_len)
  # initial values
  w = np.random.uniform(-1, 0)
  omega_m = np.random.uniform(0, 1)
  log_lh = -1e100

  for i in range(chain_len):
    omega_m_new = np.random.normal(loc=omega_m, scale=sigma)
    w_new = np.random.normal(loc=w, scale=sigma)
    new_lh = lnL(omega_m_new, w_new)
    alpha = np.random.uniform()
    if new_lh - log_lh > np.log(alpha):  # accept point
      w = w_new
      omega_m = omega_m_new
      log_lh = new_lh
    w_arr[i] = w
    omega_m_arr[i] = omega_m
    log_lh_arr[i] = log_lh

  N_burn = int(burn_in * chain_len)  # samples to burn
  return w_arr[N_burn:], omega_m_arr[N_burn:], log_lh_arr[N_burn
```

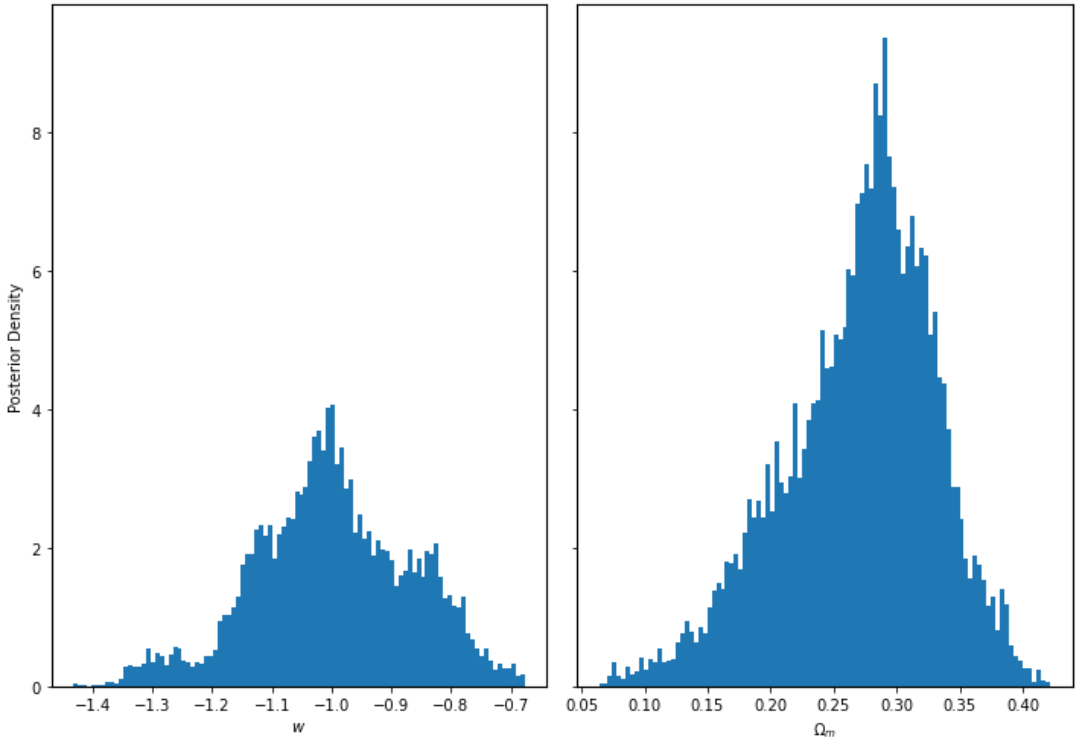
In [60]:
```
chain = MH_chain()
samples = np.arange(len(chain[0]))
```

In [85]:

```
from matplotlib import colors
from matplotlib.cm import ScalarMappable
cnorm = colors.Normalize(vmin=chain[2].min(), vmax=chain[2].max(
fig, axs = plt.subplots(figsize=(10, 7), nrows=2, sharex=True)
axs[0].scatter(samples, chain[0], c=chain[2], s=2)
axs[0].set_ylabel("$w$")
axs[1].scatter(samples, chain[1], c=chain[2], s=2)
axs[1].set_ylabel("$\Omega_m$")
axs[1].set_xlabel("Sample Number")
cax = fig.add_axes([.92, 0.125, 0.05, .755])
fig.colorbar(ScalarMappable(norm=cnorm), cax=cax, label="Log Lik
plt.show()
```



In [90]:
```
fig, axs = plt.subplots(figsize=(10,7), ncols=2, sharey=True)
axs[0].hist(chain[0], bins=100, density=True)
axs[1].hist(chain[1], bins=100, density=True)
axs[0].set_xlabel("$w$")
axs[1].set_xlabel("$\Omega_m$")
axs[0].set_ylabel("Posterior Density")
plt.tight_layout()
plt.show()
```



Problem 2 - Planck analysis continued - Nonlinear (Quadratic) Model

In Project 1 - part 1, we assumed a simple linear model of the theory CMB power spectrum with 6 cosmological parameters, $[\Omega_b h^2, \Omega_c h^2, H_0, \tau, A_s, n_s]$, and found the best-fit model using linear algebra and Gauss-Newton method, but optimization was rather trivial with the given linear model. In Part 2, we add quadratic terms to our model of the CMB power spectrum with 2 additional parameters: Ω_k (curvature density parameter) and y_{cal} (Planck survey calibration parameter). We also replace H_0 with $100\theta_{MC}$ (a measure of the sound horizon at last scattering), which can be converted into H_0.

In summary, we take a quadratic model of the CMB power spectrum with the following 8 paramters:

$$[\Omega_b h^2, \Omega_c h^2, 100\theta_{MC}, \tau, \Omega_K, \ln(10^{10} A_s), n_s, y_{\mathrm{cal}}]$$

In [3]:
```
%cd /content/drive/My Drive/P188_288/P188_288_Project1_p2new/ass
!bash install.sh
```

```
/content/drive/My Drive/P188_288/P188_288_Project1_p2new/assignm
running build_ext
running build_ext
running build_ext
running build_ext
```

In [4]:
```
import sys; sys.path.append('/content/drive/My Drive/P188_288/P1
sys.path.append('/content/drive/My Drive/P188_288/P188_288_Proje
from cosmofast.planck2018._simall import _simall_f, _simall_j, _
from cosmofast.planck2018._plik_lite_diag import _plik_lite_f, _
from cosmofast.planck2018._commander import _commander_f, _comma
import dill
dill.settings['recurse'] = True

import os
os.environ['OMP_NUM_THREADS'] = '1'
```

From the Planck measurements, we take 28 unbinned TT power spectrum for $2 \leq \ell < 30$, 28 unbinned EE power spectrum for $2 \leq \ell < 30$, and 215 binned TT power spectrum for $30 \leq \ell < 2500$ - total of 271 data points.

Our quadratic model defined below ("quadmodel") can evaluate the power spectrum in the above 271 ℓ bins, given 7 cosmological parameters $(\Omega_b h^2, \Omega_c h^2, 100\theta_{MC}, \tau, \Omega_K, \ln(10^{10} A_s), n_s)$ - note that the calibration parameter is excluded.

Suppose that our initial guess of the 8 model parameter is:
$\mathbf{x0} = [\Omega_b h^2, \Omega_c h^2, 100\theta_{MC}, \tau, \Omega_K, \ln(10^{10} A_s), n_s, y_{\rm cal}] = [0.022, 0.10, 1., 0.08, ($

In [5]:
```
# Load the quadratic model for Planck 2018 Omega_K lilelihood
# Note that we do linear extrapolation outside the sample range
with open('data/den2.p', 'rb') as f:
    quadmodel = dill.load(f)
```

In [6]:
```
x0 = np.array([ 0.022,  0.10 ,  1.  ,  0.08, 0., 3.  ,  0.958 ,
```

Then, given our initial guess x0, we can evaluate model power spectra in the following way:

In [7]:
```
model_ps = quadmodel._module_list[0](x0[:7])
model_ps
```

Out[7]:
```
array([6.91610226e+02, 3.63341831e+02, 2.08357261e+02, 1.3191217
       1.22423873e+02, 6.95430167e+01, 5.94164913e+01, 4.4654288
       4.93060628e+01, 4.38074337e+01, 2.06645719e+01, 1.8622758
       2.12285383e+01, 1.79326910e+01, 1.93605039e+01, 2.0633346
       1.71381889e+01, 1.27547324e+01, 1.06852909e+01, 1.0129209
       9.93998041e+00, 9.67933799e+00, 9.19967225e+00, 8.6446110
       8.14426492e+00, 7.70317244e+00, 7.30018898e+00, 6.9325285
       4.81106001e-02, 4.09130973e-02, 2.56204030e-02, 1.1409055
       3.14122506e-03, 3.75592108e-04, 2.15091107e-04, 4.3069134
       3.32425286e-04, 1.09954963e-04, 3.33979361e-05, 9.0989840
       1.31559635e-04, 1.16787888e-04, 8.26549141e-05, 4.7814765
       2.18076623e-05, 1.46417974e-05, 2.89304906e-05, 5.1327408
       7.09230046e-05, 8.53768765e-05, 9.46417097e-05, 9.9636914
       1.01812987e-04, 1.03986982e-04, 1.07703743e-04, 1.1131656
       4.89535692e+00, 6.63095324e+00, 7.64019410e+00, 8.3792232
       8.62942154e+00, 1.10637735e+01, 1.04380593e+01, 9.6001654
       9.55317435e+00, 1.34293842e+01, 1.05446641e+01, 1.0592468
       1.36148486e+01, 1.50774954e+01, 1.93819188e+01, 2.0880794
       2.27183390e+01, 2.34056044e+01, 2.53129817e+01, 2.6516765
       2.71114018e+01, 2.75450740e+01, 2.81909971e+01, 3.1040867
       3.17780403e+01, 3.18092964e+01, 3.42603674e+01, 3.4309467
       3.59277553e+01, 3.75192343e+01, 3.68341657e+01, 3.8918505
       4.01076724e+01, 4.08292542e+01, 4.14452365e+01, 4.3006318
       4.23491377e+01, 4.44981350e+01, 4.37679110e+01, 4.4527570
       4.68775135e+01, 4.46600230e+01, 4.68364451e+01, 4.5281323
       4.46688078e+01, 4.37877230e+01, 4.26613969e+01, 4.0892575
       4.08927198e+01, 4.01392549e+01, 3.86383684e+01, 3.9578460
       3.89086374e+01, 4.05685367e+01, 4.01728626e+01, 3.9638985
       4.17170766e+01, 4.28574232e+01, 4.46145044e+01, 4.5410940
       4.68318460e+01, 4.97966025e+01, 4.94697829e+01, 5.0618314
       5.19498873e+01, 5.17746140e+01, 5.26458242e+01, 5.3809659
       5.47751485e+01, 5.43927681e+01, 5.52720288e+01, 5.4105631
       5.38040953e+01, 5.16485467e+01, 5.27384996e+01, 4.9823598
       4.89202796e+01, 4.61825644e+01, 4.43234255e+01, 4.5070597
       4.15097528e+01, 4.17500528e+01, 4.11165434e+01, 4.1422794
       4.17751457e+01, 4.17644890e+01, 4.29210775e+01, 4.4084519
       4.45034534e+01, 4.79868222e+01, 4.84769150e+01, 4.9720200
       5.31910812e+01, 5.38784162e+01, 5.45136735e+01, 5.7954335
       5.96742192e+01, 6.06627819e+01, 6.23064873e+01, 6.4681587
       6.35125254e+01, 6.56860519e+01, 6.68588007e+01, 6.7374399
       6.68081578e+01, 6.66844997e+01, 6.61303738e+01, 6.5373045
       6.38941365e+01, 6.23325629e+01, 5.80608515e+01, 5.5388399
       5.22894573e+01, 4.86895181e+01, 4.52239310e+01, 4.1649873
       3.96066284e+01, 3.88483627e+01, 3.71341278e+01, 3.5753946
       3.52804856e+01, 3.60304799e+01, 3.75352055e+01, 3.9027260
       4.09576215e+01, 4.25188607e+01, 4.39583847e+01, 4.6477742
       4.84951935e+01, 5.08484949e+01, 5.20330072e+01, 5.3307480
       5.46636789e+01, 5.66825689e+01, 5.77958573e+01, 5.8531395
       5.72446451e+01, 5.70784022e+01, 5.71387176e+01, 5.6244882
       5.42069103e+01, 5.24743355e+01, 5.14883859e+01, 4.7766963
       4.50369389e+01, 4.13494508e+01, 3.88022893e+01, 3.5665481
       3.31906723e+01, 3.07080561e+01, 2.94335662e+01, 2.8041408
       2.71058658e+01, 2.77477815e+01, 2.77604632e+01, 2.8737537
       2.97803324e+01, 3.11803047e+01, 3.31859116e+01, 3.5072529
       3.65168685e+01, 3.88654550e+01, 4.05673499e+01, 4.2873863
       4.49671197e+01, 4.64631551e+01, 4.75890974e+01, 4.9149926
```

```
4.94074757e+01, 5.08533949e+01, 6.94698362e+01, 6.8302187
6.43876479e+01, 5.98171567e+01, 5.24071522e+01, 4.4562236
3.67690495e+01, 2.97141408e+01, 2.44398266e+01, 2.1352731
2.07591107e+01, 2.12741535e+01, 2.35531219e+01, 2.5884825
2.83904549e+01, 3.06498916e+01, 3.23492134e+01, 3.3546515
3.30804632e+01, 3.17868531e+01, 2.92072365e+01, 2.6190032
2.24134628e+01, 1.87424298e+01, 1.53613623e+01, 1.2426915
1.03259592e+01, 8.96282554e+00, 8.46409487e+00, 8.2999687
1.26421470e+01, 1.45134502e+01, 1.56845456e+01, 1.5095748
1.28871193e+01, 9.74977368e+00, 6.53837017e+00, 4.0737246
2.67216870e+00, 2.21582948e+00, 2.32342275e+00, 2.6670267
2.97040554e+00, 3.38029367e+00, 4.33758898e+00])
```

You see that it outputs the power spectra in 271 ℓ bins.

Note that 215 high-ℓ ($\ell > 30$) TT power spectrum from the model (the last 215 entries) is also diagonalized (multiplied by the cholesky of the likelihood covariance) so that the likelihood is simply -0.5*(data-model)^2.

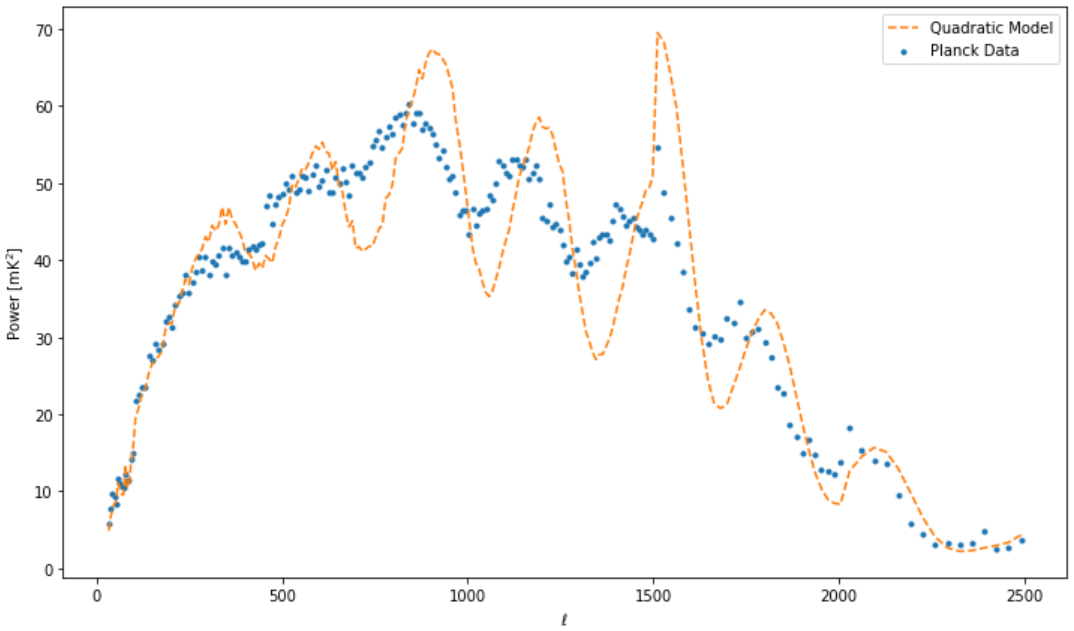
Now let's compare this diagonalized high-ℓ TT power spectrum to the measured data.

In [8]:
```
# Load measured data
Y = np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
ell = Y[0,:]
measured = Y[1,:]
```

"ell" is the effective ℓ values in 215 ℓ bins, and "measured" is the corresponding measured TT power spectra.

1. Show how well the high-ℓ TT model power spectra (given our initial guess x0) fits the data. i.e. Plot a "scatterplot" of the measured data (use '.' marker) and the high-ℓ TT model power spectra (use a dotted line '--') in one figure.

In [9]:
```
plt.figure(figsize=(12,7))
plt.scatter(ell, measured, label="Planck Data", marker=".")
plt.plot(ell, model_ps[-215:], label="Quadratic Model", c="C1",
plt.legend()
plt.xlabel("$\ell$")
plt.ylabel("Power [$\mathrm{mK}^2$]")
plt.show()
```



Next, let's try the **maximum a posteriori estimation (MAP)**.

quadmodel(x0) outputs the log-posterior (logp) distribution. To get the MAP estimate, we should maximize logp (i.e. (data-model) is minimized). Then, we need the graident and hessian of -logp.

Given x0, we can evaluate the gradient of -logp:

-quadmodel.grad(x0)

The hessian of -logp given x0 is:

Hessian(lambda xx: -quadmodel(xx), 1e-4)(x0)

Note that "-quadmodel.grad" and "Hessian(lambda xx: -quadmodel(xx), 1e-4)" are functions, and you can evaluate them with different model parameter values.

Now, let us use scipy.optimize.minimize (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html)) to get the MAP estimate of the model parameters: we can minimize -logp to get the MAP estimate.

2. Using the Newton conjugate gradient method ('Newton-CG'), find the MAP estimate of 8 model parameters with scipy.optimize.minimize. Print MAP estimates of the 8 model parameters.

Hint: you can do

opt_NCG = minimize(**"1.-logp function"**, **2.initial guess (x0 in this case)**, method='Newton-CG', jac=**3.function for gradient of -logp**, hess=**4.function for**

gradient of -logp, options={'return_all': True, 'disp': True})

After optimization terminated successfully, you can retrieve the best-fit parameters with:

opt_NCG.x

In [10]:
```
from scipy.optimize import minimize
from hessian import Hessian
```

In [11]:
```
opt_NCG = minimize(
    lambda x: -quadmodel(x),
    x0,
    method="Newton-CG",
    jac=lambda x: -quadmodel.grad(x),
    hess=Hessian(lambda xx: -quadmodel(xx), 1e-4),
    options={"return_all": True, "disp": True},
)
```

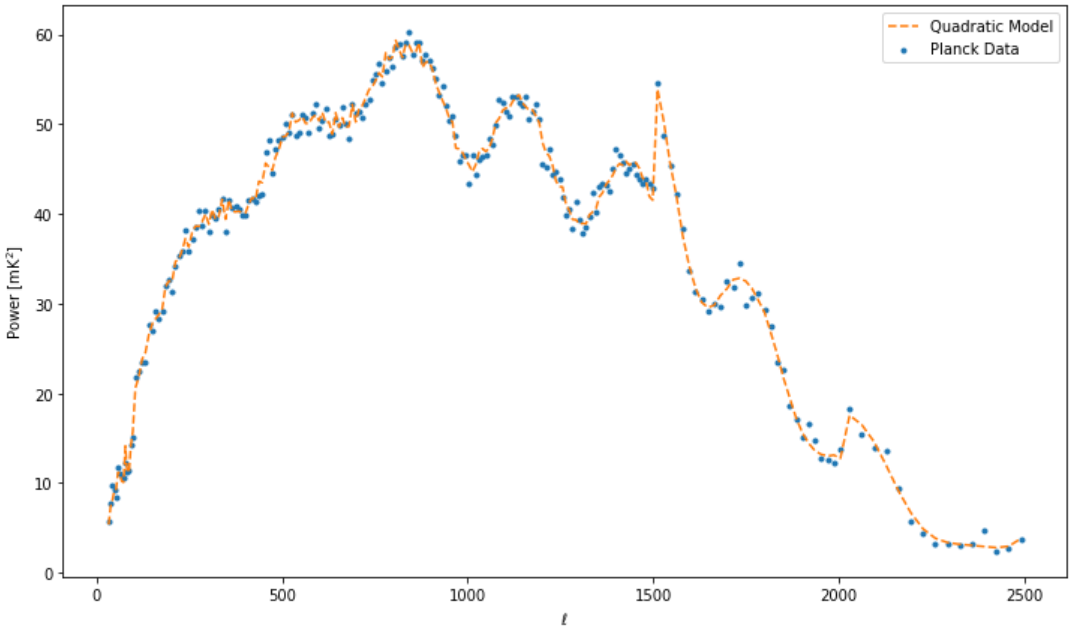
```
Optimization terminated successfully.
         Current function value: 302.831104
         Iterations: 13
         Function evaluations: 14
         Gradient evaluations: 14
         Hessian evaluations: 13
```

In [12]:
```
labels =  ['\\Omega_b h^2', '\\Omega_c h^2', '100\\theta_{MC}',
           '\Omega_K', '{\\rm{ln}}(10^{10} A_s)', 'n_s', 'y_{\\r
for i in range(len(labels)):
    print("MAP value of %s = %.5f" %(labels[i], opt_NCG.x[i]))
```

```
MAP value of \Omega_b h^2 = 0.02252
MAP value of \Omega_c h^2 = 0.11766
MAP value of 100\theta_{MC} = 1.04120
MAP value of \tau = 0.04899
MAP value of \Omega_K = -0.05048
MAP value of {\rm{ln}}(10^{10} A_s) = 3.02766
MAP value of n_s = 0.97079
MAP value of y_{\rm cal} = 0.99996
```

3. Repeat Part 1 with the best-fit model you have found in Part 2.

In [13]:
```
ncg_model = quadmodel._module_list[0](opt_NCG.x[:7])

plt.figure(figsize=(12,7))
plt.scatter(ell, measured, label="Planck Data", marker=".")
plt.plot(ell, ncg_model[-215:], label="Quadratic Model", c="C1",
plt.legend()
plt.xlabel("$\ell$")
plt.ylabel("Power [$\mathrm{mK}^2$]")
plt.show()
```



Run below cells to create an animation which shows how the model spectrum fits better with the measured data as we iterate in the optimization process.

You can actually play this animation.

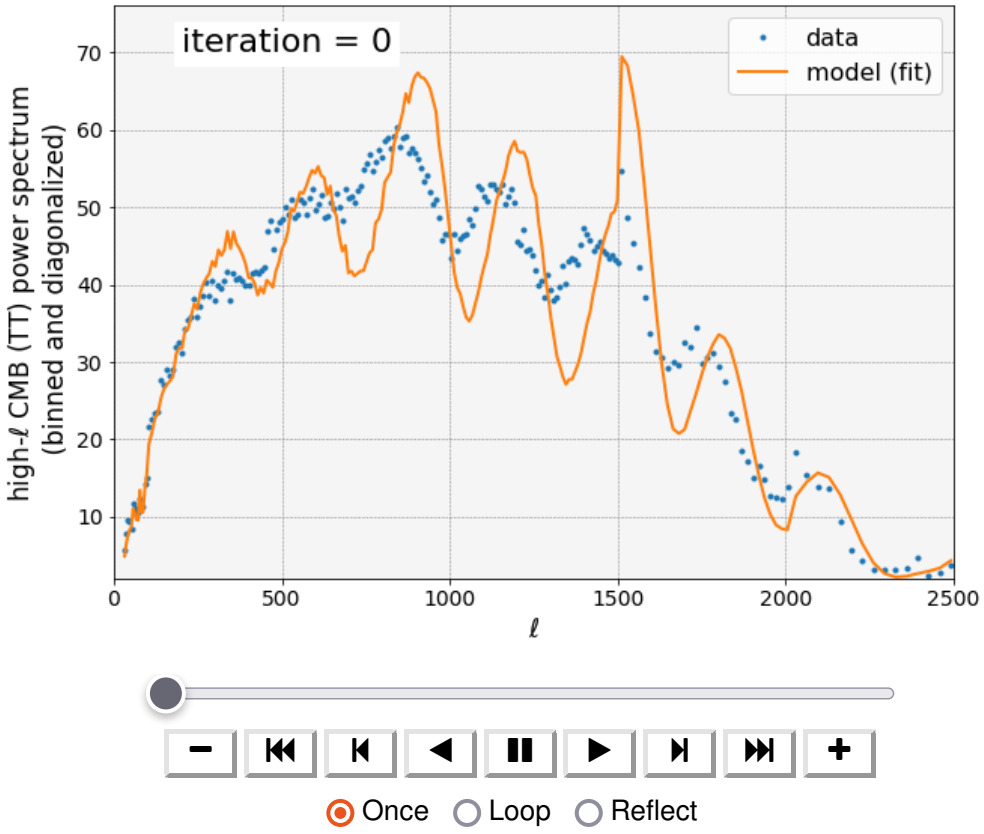
In [14]:
```
from matplotlib import animation, rc
from IPython.display import HTML
```

In [15]:

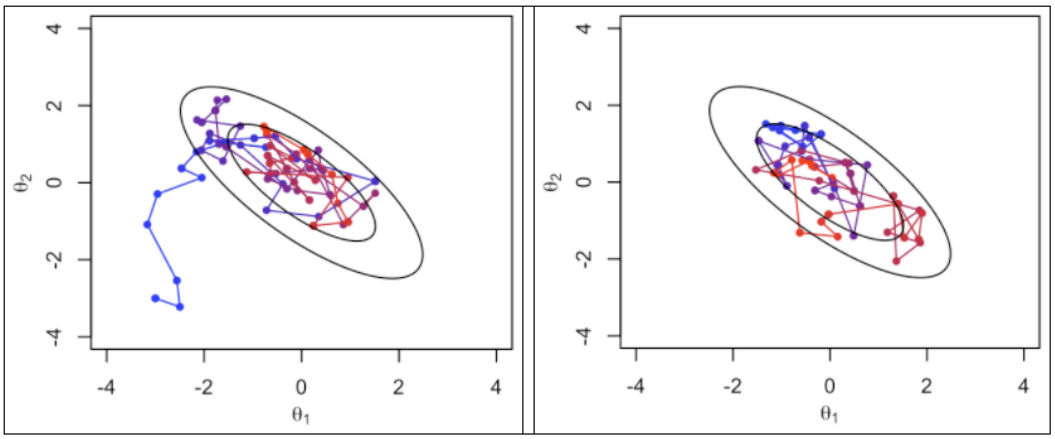
```
%%capture
# First set up the figure, the axis, and the plot element we wan
fig, ax = plt.subplots(figsize=(10, 7), edgecolor='black')
rc('axes', linewidth=1.1)


ax.set_xlim(( 0, 2500))
ax.set_ylim((2, 76))
ax.set_xticks([0, 500, 1000, 1500, 2000, 2500])
ax.set_xticklabels([0, 500, 1000, 1500, 2000, 2500], fontsize =

ax.set_yticks([10, 20, 30, 40, 50, 60, 70])
ax.set_yticklabels([10, 20, 30, 40, 50, 60, 70], fontsize = 14)

ax.set_xlabel('$\ell$', fontsize = 17)
ax.set_ylabel('high-$\ell$ CMB (TT) power spectrum \n (binned an


ax.plot(ell, measured, '.', label = 'data')
ax.set_facecolor('whitesmoke')
ax.grid(linestyle='--', linewidth='0.5', color='grey')
line, = ax.plot([], [], lw=2, label = 'model (fit)')

#initialization function: plot the background of each frame
def init():
    line, = ax.plot([], [], lw=2)
    line.set_data([], [])
    return (line,)

# animation function. This is called sequentially
def animate(i):
    y = quadmodel._module_list[0](opt_NCG.allvecs[i][:7])[56:]
    line.set_data(ell, y)
    ax.text(200, 70, 'iteration = %d' %i, color='black',fontsize
        bbox={'facecolor': 'white', 'alpha': 1.0, 'pad': 4.2, 'e
    return (line,)

ax.legend(fontsize = 16)


# call the animator. blit=True means only re-draw the parts that
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=opt_NCG.nit, interval=500,
```

In [16]:
```
HTML(anim.to_jshtml())
```

Out[16]:



Markov chain Monte Carlo is a general method based on drawing values of θ from approximate distributions and then correcting those draws to better aproximate the target posterior distribution. The sampling is done sequentially, wtih the distribution of the sampled draws depending on the last value drawn - hence, the draws from a Markov chain. (p. 275, *Bayesian Data Analysis*, Andrew Gelman et al.) (Remember that a sequence $x_1, x_2, \ldots$ of random events is called a Markov chain if x_{n+1} depends explicitly on x_n only (and not explicitly on previous steps).) Here, we consider our 8 model paramters, so the "chain" in this case is a random walk through the parameter space.



from https://github.com/KIPAC/StatisticalMethods/blob/master/chunks/montecarlo1.ipynb (https://github.com/KIPAC/StatisticalMethods/blob/master/chunks/montecarlo1.ipynb)

As shown in the above figure, chains take time to converge to the target distribution, and you can determine the "burn-in" period, the number of sequences it takes to reach convergence.

In Part 4, we provide you MCMC chains from Planck. You can plot the chains in the parameter space and estimate the posterior distribution.

References:
Bayesian Data Analysis, Andrew Gelman et al.

Now, we provide you with 4 independent Planck MCMC chains. For each chain, we load the data for 8 model parameters,
$[\Omega_b h^2, \Omega_c h^2, 100\theta_{MC}, \tau, \Omega_K, \ln(10^{10} A_s), n_s, y_{\rm cal}]$.

In [17]:
```
# These are the official Planck chains
x_mcmc = np.concatenate((
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
p_mcmc = np.concatenate((
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
    np.loadtxt('/content/drive/My Drive/P188_288/P188_288_Projec
```

In [18]:
```
!pip install getdist
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pk
Collecting getdist
  Downloading GetDist-1.3.4.tar.gz (777 kB)
     |████████████████████████████████| 777 kB 9.9 MB/s
Requirement already satisfied: numpy>=1.17.0 in /usr/local/lib/py
Requirement already satisfied: matplotlib!=3.5.0,>=2.2.0 in /usr/
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/py
Requirement already satisfied: PyYAML>=5.1 in /usr/local/lib/pyth
Requirement already satisfied: python-dateutil>=2.1 in /usr/local
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/py
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/li
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>
Requirement already satisfied: typing-extensions in /usr/local/li
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3
Building wheels for collected packages: getdist
  Building wheel for getdist (setup.py) ... done
  Created wheel for getdist: filename=GetDist-1.3.4-py3-none-any
  Stored in directory: /root/.cache/pip/wheels/b4/f0/9d/b16a7e7c
Successfully built getdist
Installing collected packages: getdist
Successfully installed getdist-1.3.4
```

In [19]:
```
from getdist import plots, MCSamples
```

Here, we use MCSamples from **getdist** package (https://getdist.readthedocs.io/en/latest/mcsamples.html (https://getdist.readthedocs.io/en/latest/mcsamples.html)) to

create a class of MCMC samples.

MCMC = MCSamples(samples=x_mcmc, weights=p_mcmc)

To get actual samples from this class, we can do:

MCMC_samples = MCMC.samples

We reshape this in the following way:

MCMC_samples = MCMC_samples.reshape(# of samples per chain, # of chains, # of parameters)

In [20]:
```
names = ["x%s"%i for i in range(8)]
labels =  ['\\Omega_b h^2', '\\Omega_c h^2', '100\\theta_{MC}',
           '\Omega_K', '{\\rm{ln}}(10^{10} A_s)', 'n_s', 'y_{\\r
MCMC = MCSamples(samples=x_mcmc, weights=p_mcmc, names=names, la
MCMC_samples = MCMC.samples
MCMC_samples = MCMC_samples.reshape(3724,4,8)
```

```
Removed no burn in
```

In MCMC, we need to make sure that chains converge to the posterior distribution. One useful test for convergence is "Gelman-Rubin statistic." For a given parameter, θ, the R statistic compares the variance across chains with the variance within a chain. Intuitively, if the chains are random-walking in very different places, i.e. not sampling the same distribution, R will be large.

In detail, given chains $J = 1, \ldots, m$, each of length n,
Let $B = \frac{n}{m-1} \sum_j \left(\bar{\theta}_j - \bar{\theta}\right)^2$, where $\bar{\theta_j}$ is the average θ for chain j and $\bar{\theta}$ is the global average. This is proportional to the variance of the individual-chain averages for θ.
Let $W = \frac{1}{m} \sum_j s_j^2$, where s_j^2 is the estimated variance of θ within chain j. This is the average of the individual-chain variances for θ.
Let $V = \frac{n-1}{n} W + \frac{1}{n} B$. This is an estimate for the overall variance of θ.

Finally, $R = \sqrt{\frac{V}{W}}$. We'd like to see $R \approx 1$ (e.g. $R < 1.1$ is often used). Note that this calculation can also be used to track convergence of combinations of parameters, or anything else derived from them.

Reference: https://github.com/KIPAC/StatisticalMethods/blob/master/chunks/montecarlo1.ipynb (https://github.com/KIPAC/StatisticalMethods/blob/master/chunks/montecarlo1.ipynb)

4. For all eight parameters, compute R and determine if the condition $R < 1.1$ is satisfied.

In [21]:
```
n, m, npar = MCMC_samples.shape
print("R:")
for i in range(npar):
  par_samples = MCMC_samples[:, :, i]
  chain_avg = par_samples.mean(axis=0)
  global_avg = chain_avg.mean()
  B = n / (m-1) * np.sum((chain_avg - global_avg)**2)
  chain_var = np.var(par_samples, axis=0)
  W = chain_var.mean()
  V = (n-1)/n * W + 1/n * B
  R = np.sqrt(V/W)
  print(f"{labels[i]}: {R:.5f}")
```

```
R:
\Omega_b h^2: 0.99988
\Omega_c h^2: 0.99988
100\theta_{MC}: 0.99988
\tau: 0.99990
\Omega_K: 0.99990
{\rm{ln}}(10^{10} A_s): 0.99990
n_s: 0.99988
y_{\rm cal}: 0.99987
```

The condition R < 1.1 is indeed satisfied for all 8 parameters.

The autocorrelation of a sequence, as a function of lag, k, is defined thusly:

$$\rho_k = \frac{\sum_{i=1}^{n-k} \left(\theta_i - \bar{\theta}\right) \left(\theta_{i+k} - \bar{\theta}\right)}{\sum_{i=1}^{n-k} \left(\theta_i - \bar{\theta}\right)^2} = \frac{\mathrm{Cov}_i \left(\theta_i, \theta_{i+k}\right)}{\mathrm{Var}(\theta)}$$

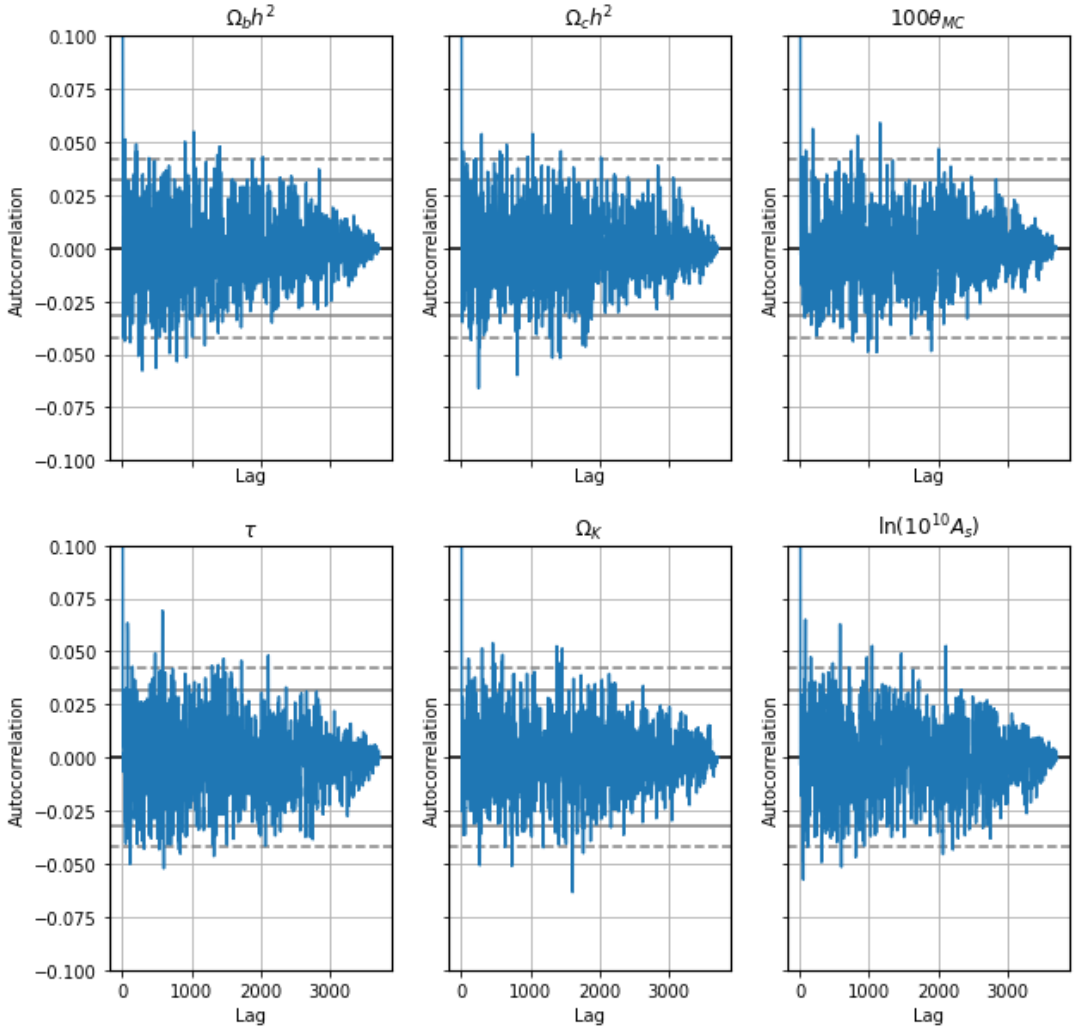
The larger lag one needs to get a small autocorrelation, the less informative individual samples are.

5. Using autocorrelation_plot from pandas (https://pandas.pydata.org/pandas-docs/stable/visualization.html#visualization-autocorrelation), plot the auto-correlation of six parameters and determine that it gets small for large lag. The given Planck MCMC chains are already heavily thinned, so you will not see much autocorrelation.

In [22]:
```
!pip install pandas
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.p
Requirement already satisfied: pandas in /usr/local/lib/python3.
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/p
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/py
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/lo
Requirement already satisfied: six>=1.5 in /usr/local/lib/python
```

In [33]:
```
from pandas.plotting import autocorrelation_plot

fig, axs = plt.subplots(figsize=(10,10), nrows=2, ncols=3, share
for i in range(6):
  ax = axs.ravel()[i]
  autocorrelation_plot(MCMC_samples[:, :, i], ax=ax)
  ax.set_title(f"${labels[i]}$")
plt.setp(axs, ylim=(-0.1, 0.1))
plt.show()
```



6. From the chain, obtain 1-d constraints on all 8 parameters. Print results. Also, show how far your MAP estimates are from the MCMC means (Say (MAP estimate-MCMC mean)/MCMC standard deviation = 2. Then, your MAP estimate is 2 sigma away from the MCMC mean.)

In [40]:
```
print("MCMC Constraints:\n")
mcmc_constraints = np.empty((npar, 2))
for i in range(npar):
  samples = MCMC_samples[:, :, i]
  mean = samples.mean()
  sigma = samples.std(axis=0).mean()
  mcmc_constraints[i] = [mean, sigma]
  print(f"{labels[i]}: {mean:.5f} +- {sigma:.5f}")
```

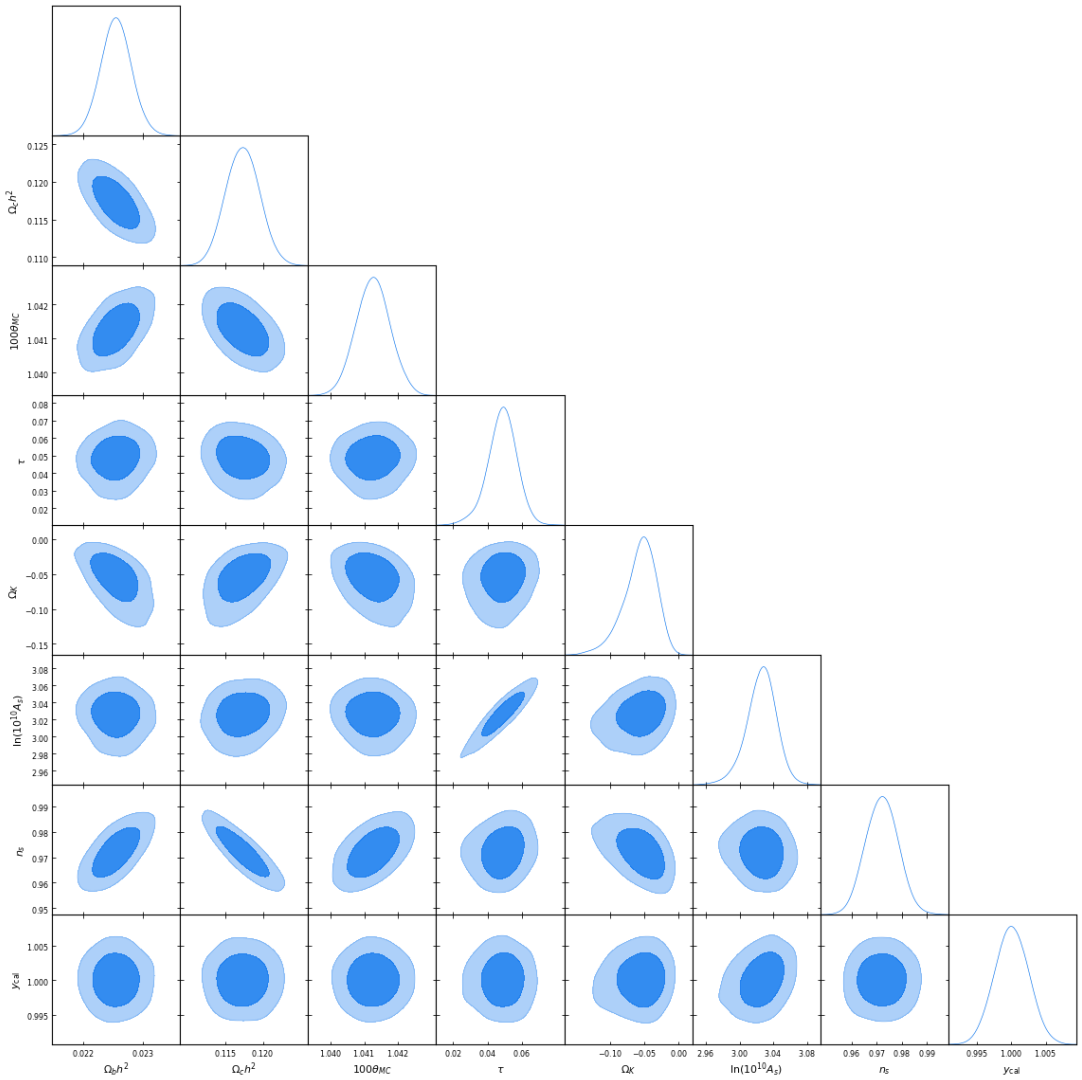
```
MCMC Constraints:

\Omega_b h^2: 0.02255 +- 0.00027
\Omega_c h^2: 0.11729 +- 0.00232
100\theta_{MC}: 1.04127 +- 0.00051
\tau: 0.04833 +- 0.00880
\Omega_K: -0.05698 +- 0.02470
{\rm{ln}}(10^{10} A_s): 3.02543 +- 0.01840
n_s: 0.97212 +- 0.00650
y_{\rm cal}: 1.00003 +- 0.00252
```

In [45]:
```
print("MAP Discrepancy from MCMC:\n")
for i in range(npar):
  mean, sigma = mcmc_constraints[i]
  map = opt_NCG.x[i]
  diff = (map - mean) / sigma
  print(f"{labels[i]}: {diff:.2g} sigma")
```

```
MAP Discrepancy from MCMC:

\Omega_b h^2: -0.14 sigma
\Omega_c h^2: 0.16 sigma
100\theta_{MC}: -0.13 sigma
\tau: 0.075 sigma
\Omega_K: 0.26 sigma
{\rm{ln}}(10^{10} A_s): 0.12 sigma
n_s: -0.21 sigma
y_{\rm cal}: -0.026 sigma
```

We can now plot 1-d and 2-d constraints of our model parameters using "getdist" package:

In [31]:
```
g = plots.getSubplotPlotter()
g.triangle_plot([MCMC], filled=True, contour_args={'alpha':0.8})
```



7. Back to MAP. Plot 1-d and 2-d constraints from MAP. For this task, we should first evaluate the hessian of -logp at our MAP estimates and convert it to the covariance. (See Part 2 for hints)

In [46]:
```
Hess = Hessian(lambda xx: -quadmodel(xx), 1e-4)(opt_NCG.x)
cov = np.linalg.inv(Hess)
mean = opt_NCG.x
```

To plot MCMC and MAP results in a single figure, we should create samples from MAP. First, define a Gaussian distribution with mean and covariance defined above. Then, draw 5000 samples from it.

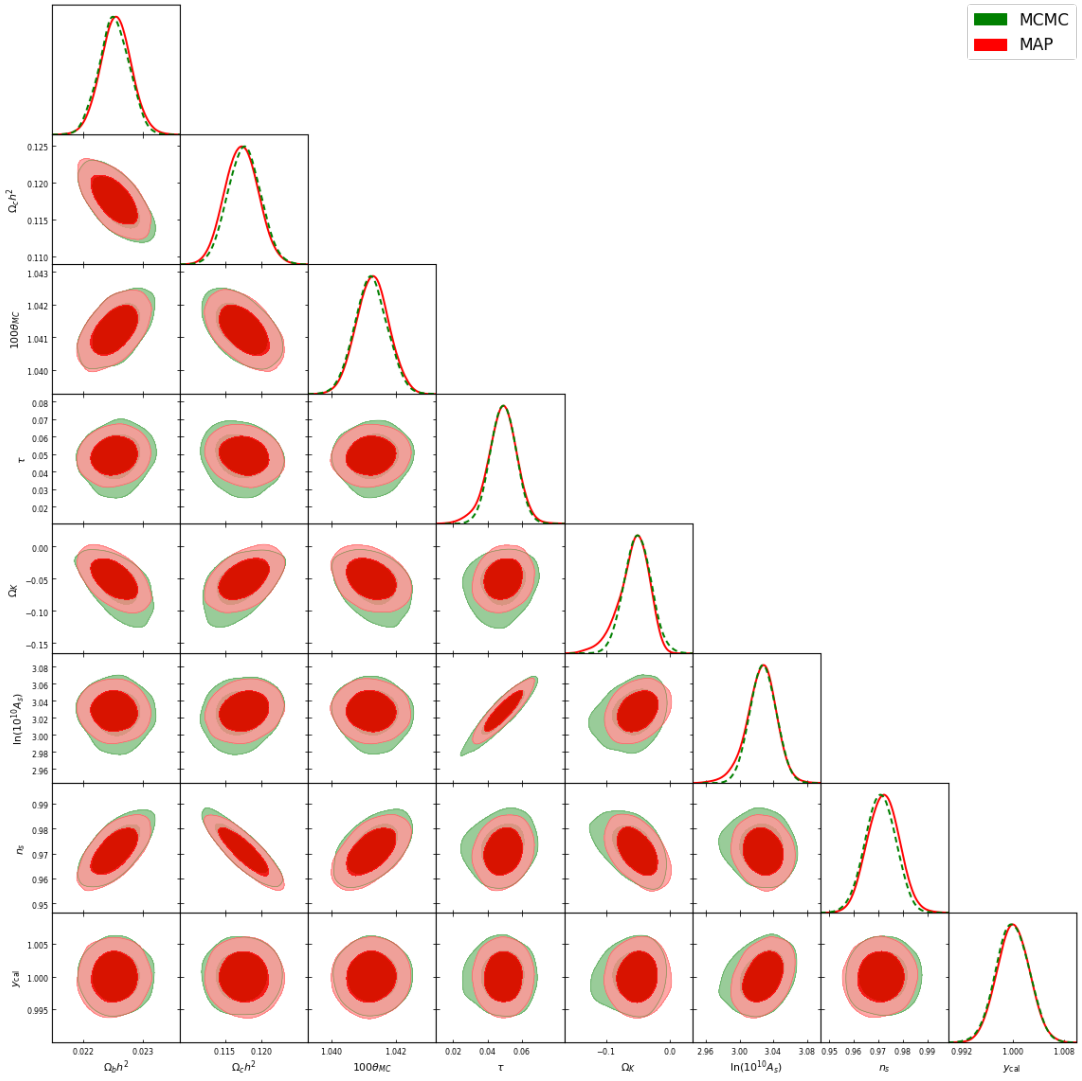
In [47]:
```
from getdist.gaussian_mixtures import GaussianND
GNd=GaussianND(mean, cov, label='MAP')
x_g = GNd.MCSamples(5000, names=names, labels=labels)
MAP = MCSamples(samples=x_g.samples, names=names, labels=labels)
```

```
Removed no burn in
Removed no burn in
```

8. Evaluate below cells to get results from MAP, MCMC, and bayesfast.

Finally, we can compare MCMC and MAP results: MAP gets the peak of the posterior most of the time, but the Laplace approximation fails to give the full posterior shape.

In [48]:
```
g = plots.getSubplotPlotter()
g.settings.figure_legend_frame = False
g.settings.legend_fontsize = 20
g.triangle_plot([MCMC,MAP], filled=True,
                colors=['green', 'red', 'royalblue'], legend_lab
                legend_loc='upper right', line_args=[{'lw':2, 'c
               )
```



We now use the method called "bayesfast" (https://www.bayesfast.org/ (https://www.bayesfast.org/)), which allows a fast evaluation of the posterior. We start from our MAP estimates and draw 2500 HMC samples.

In [49]:
```
from bayesfast.samplers.pymc3.nuts import NUTS
import bayesfast.utils.warnings as bfwarnings
import warnings

# Below is the temporary solution for printing the sampling prog
# We plan to rewrite it with e.g. tqdm in the future
warnings.showwarning = bfwarnings.showwarning_chain()
warnings.formatwarning = bfwarnings.formatwarning_chain()
```

In [50]:
```
nuts = NUTS(max_treedepth=8,
            logp_and_grad=(lambda xx: quadmodel.logp_and_grad(xx
            x_0=quadmodel.from_original(opt_NCG.x),
            random_state=np.random.RandomState(0),
            metric=None, step_size=1., target_accept=0.9)
sample_pre_transform = nuts.run(2500, 500)
sample_post_transform = quadmodel.to_original(np.array(sample_pr
bayesfast = MCSamples(samples=sample_post_transform, names=names
```

```
CHAIN #0: sampling proceeding [ 250 / 2500 ], last 250 samples u
CHAIN #0: sampling proceeding [ 500 / 2500 ], last 250 samples u
CHAIN #0: sampling proceeding [ 750 / 2500 ], last 250 samples u
CHAIN #0: sampling proceeding [ 1000 / 2500 ], last 250 samples
CHAIN #0: sampling proceeding [ 1250 / 2500 ], last 250 samples
CHAIN #0: sampling proceeding [ 1500 / 2500 ], last 250 samples
CHAIN #0: sampling proceeding [ 1750 / 2500 ], last 250 samples
CHAIN #0: sampling proceeding [ 2000 / 2500 ], last 250 samples
CHAIN #0: sampling proceeding [ 2250 / 2500 ], last 250 samples
CHAIN #0: sampling finished   [ 2500 / 2500 ], obtained 2500 sam
Removed no burn in
```

You can see that the bayesfast posterior is almost equivalent to the MCMC posterior, and it succeeds in finding the non-Gaussian feature of the posterior.

In [51]:

```
g = plots.getSubplotPlotter()
g.settings.figure_legend_frame = False
g.settings.legend_fontsize = 20
g.triangle_plot([MAP, MCMC, bayesfast], filled=True,
                colors=['green', 'red', 'royalblue'], legend_lab
                legend_loc='upper right', line_args=[{'ls':'--',
                                                      {'lw':2, 'c
               )
```