Open in Colab
(https://colab.research.google.com/github/christianhbye/bayesian-analysis/blob/main/homeworks/HW6_288.ipynb)

Homework 6

MLE, MCMC, Distributional Approximation, Expectation Maximization (EM)

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]:
```python
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]:
```python
from google.colab import drive
drive.mount('/content/drive')
```

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

Problem 1 - Back to Supernova

In this homework, 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:

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

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

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 distance modulus)

In [ ]:
```
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6/sn_z_
# z
z_data = data[:,0]
# mu
mu_data = data[:,1]
# error on mu (sigma(mu))
mu_err_data = data[:,2]
```

In Project1-part2, with measurements of the distance modulus μ, we used Bayesian analysis to estimate the cosmological parameters w and Ω_m.

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:

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

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

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:

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

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.

First, try the **maximum likelihood estimation (MLE)**.

1. Assuming that h = 0.7, find the maximum likelihood estimation of Ω_m and w (i.e. find Ω_m and w which maximizes the likelihood.

(Hint: This is very similar to HW5 problem. Take the log of the likelihood and maximize it using scipy.optimize.fmin (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.optimize.fmin.html (https://docs.scipy.org/doc/scipy-0.19.1/reference/generated/scipy.optimize.fmin.html)). Note that you need to make initial guesses on the parameters in order to use fmin. You can set them to be 0. Caveat: "fmin" minimizes a given function, so you should multiply the log-likelihood by -1 in order to maximize it using fmin.)

In [ ]:
```python
c = 2997.92458

def integrand(z, omegam, w):
  """
  Function to inegrate to compute D_L
  """
  x = 1 + z
  d = omegam * x**3 + (1-omegam) * x**(3 + 3 * w)
  return 1 / np.sqrt(d)

def distL(z, omegam, w, h=0.7):
  """
  Compute Distance Luminosity
  """
  z = np.array(z).ravel()
  integral = np.empty(z.size)
  for i in range(z.size):
    integral[i] = quad(integrand, 0, z[i], args=(omegam, w))[0]
  return c/h * (1+z) * integral

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

In [ ]:
```python
from scipy import optimize

def minus_log_likelihood(param):
  Omegam, w = param
  if(Omegam<=0 or w>=0):
    lnL = -1.e100
  else:
    dL = distL(z_data, Omegam, w)
    mu_model = dist_mod(dL)
    lnL = -1/2 * np.sum((mu_data - mu_model)**2 / mu_err_data**2)
  return -lnL
```

In [ ]:
```python
Omegam, w = optimize.fmin(minus_log_likelihood, (0, 0))
print('MAP solution')
print('Omega_m = ', Omegam, ', w = ', w)
```

```
Optimization terminated successfully.
         Current function value: 281.112865
         Iterations: 63
         Function evaluations: 124
MAP solution
Omega_m =  0.2796236523092833 , w =  -1.0044614428682217
```

Now, use emcee package (reference: Problem 2 and 3 in HW5) to get the full posteriors of w and Ω_m.

2. (1) Run few parallel sequences of Metropolis algorithm simulations using the package "emcee". (2) Print your constraints on w and Ω_m. (3) Plot 1-d posterior of w and Ω_m as well as 2-d posterior using the package "corner" (reference: Problem 2 in HW5). Make sure that your chains have converged.

Hint: You may want to start with a small number of "nsteps" for debugging.

"Walkers" are the members of the ensemble. They are almost like separate Metropolis-Hastings chains but, of course, the proposal distribution for a given walker depends on the positions of all the other walkers in the ensemble. (https://emcee.readthedocs.io/en/stable/user/faq/?highlight=walker#what-are-walkers (https://emcee.readthedocs.io/en/stable/user/faq/?highlight=walker#what-are-walkers)) It is recommended that nwalkers > 2 · number of parameters.

In [ ]:
```
!pip install emcee
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev,
Collecting emcee
  Downloading emcee-3.1.3-py2.py3-none-any.whl (46 kB)
     |████████████████████████████████| 46 kB 3.3 MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-
Installing collected packages: emcee
Successfully installed emcee-3.1.3
```

In [ ]:
```python
import emcee
```

```
In [ ]: def log_posterior(param):
          """
          With a flat prior the posterior is proportional to the likelihood
          """
          return -1 * minus_log_likelihood(param)

        nchains = 10
        nparams = 2
        nwalkers = 10
        nburn = 100
        nsteps = 250
        trace = np.empty((nchains, nwalkers*(nsteps-nburn), nparams))

        # iniitalize walkers in a small ball around the MAP
        map = np.array([Omegam, w])
        scale = 0.1 * np.abs(map)  # set standard dev to be 10% of MAP

        for i in range(nchains):
          x0 = np.random.normal(loc=map, scale=scale, size=(nwalkers, nparams)
          sampler = emcee.EnsembleSampler(nwalkers, nparams, log_posterior)
          sampler.run_mcmc(x0, nsteps)
          trace[i] = sampler.chain[:, nburn:, :].reshape(-1, nparams)
```

```
In [ ]: def gelman_rubin(trace):
          m, n, npar = trace.shape
          R = np.empty(npar)
          for i in range(npar):
            par_samples = trace[:, :, i]
            chain_avg = par_samples.mean(axis=-1)
            global_avg = chain_avg.mean()
            B = n / (m-1) * np.sum((chain_avg - global_avg)**2)
            chain_var = np.var(par_samples, axis=-1)
            W = chain_var.mean()
            V = (n-1)/n * W + 1/n * B
            R[i] = np.sqrt(V/W)
          return R
```

```
In [ ]: # ensure that chains have converged with Gelman-Rubin test
        params = ["Omega_m", "w"]
        R = gelman_rubin(trace)
        print("R:")
        for i in range(len(R)):
          print(f"{params[i]}: {R[i]:.5f}")
```

```
R:
Omega_m: 1.02451
w: 1.02659
```

By the Gelman-Rubin test, we conclude that the chains converge for both parameters.

```
In [ ]: omega_emcee, w_emcee = trace.mean(axis=(0, 1))
        # compute variance for each chain and avg across chains
        emcee_var = np.var(trace, axis=1).mean(axis=0)
        omega_std, w_std = np.sqrt(emcee_var)
```

```
In [ ]: print('MCMC result:')
        print('Omega_m = ', omega_emcee, '+/-' , omega_std)
        print('w = ', w_emcee, '+/-' , w_std)
```

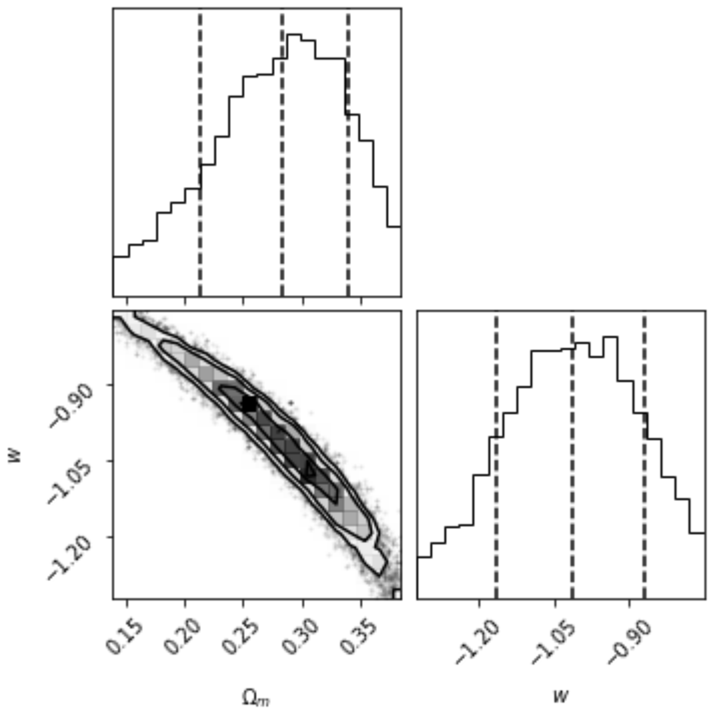
```
MCMC result:
Omega_m =  0.2778836336141023 +/- 0.06294251455655188
w =  -1.0182393565385401 +/- 0.14526295691957733
```

In [ ]:
```
!pip install corner
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/
Collecting corner
  Downloading corner-2.2.1-py3-none-any.whl (15 kB)
Requirement already satisfied: matplotlib>=2.1 in /usr/local/lib/pytho
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/pyt
Requirement already satisfied: typing-extensions in /usr/local/lib/pyt
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/di
Installing collected packages: corner
Successfully installed corner-2.2.1
```

In [ ]:
```
import corner
```

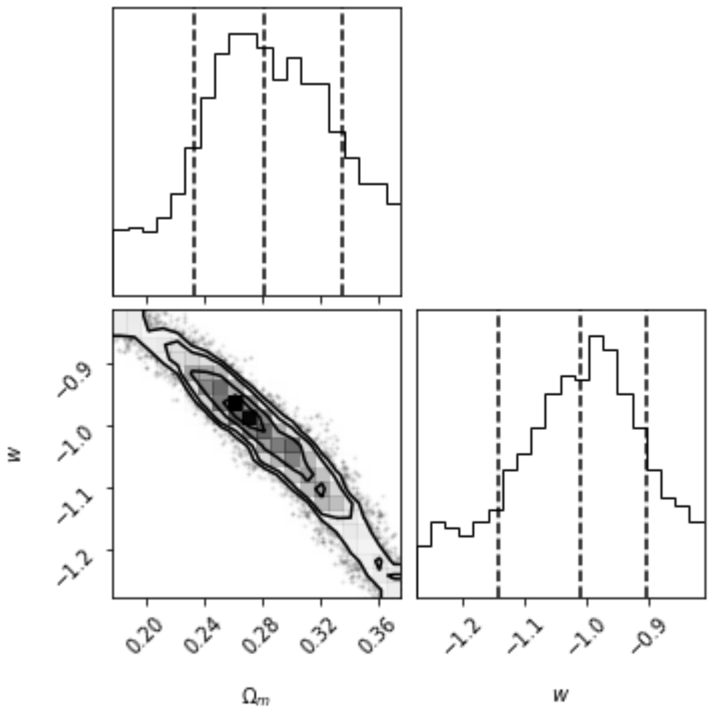
In [ ]:
```
fig = corner.corner(trace.reshape(-1, nparams), labels=["$\\Omega_m$",
```



3. Load the following samples of Ω_m and w generated by the Metropolis-Hastings sampler in Project 1 part 2. Make a corner plot for these samples and compare the two. How do the posteriors compare?

In [ ]:
```
mh_samples = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6
```

In [ ]:
```
fig = corner.corner(mh_samples, labels=["$\\Omega_m$", "$w$"], quantil
```



Comment on how they compare here: The two plots are similar but the results from Project1 favour slighlty larger values of Ω_m and smaller values of $w$. Given the 2D-constraints, this makes sense, as the two parameters are negatively correlated.

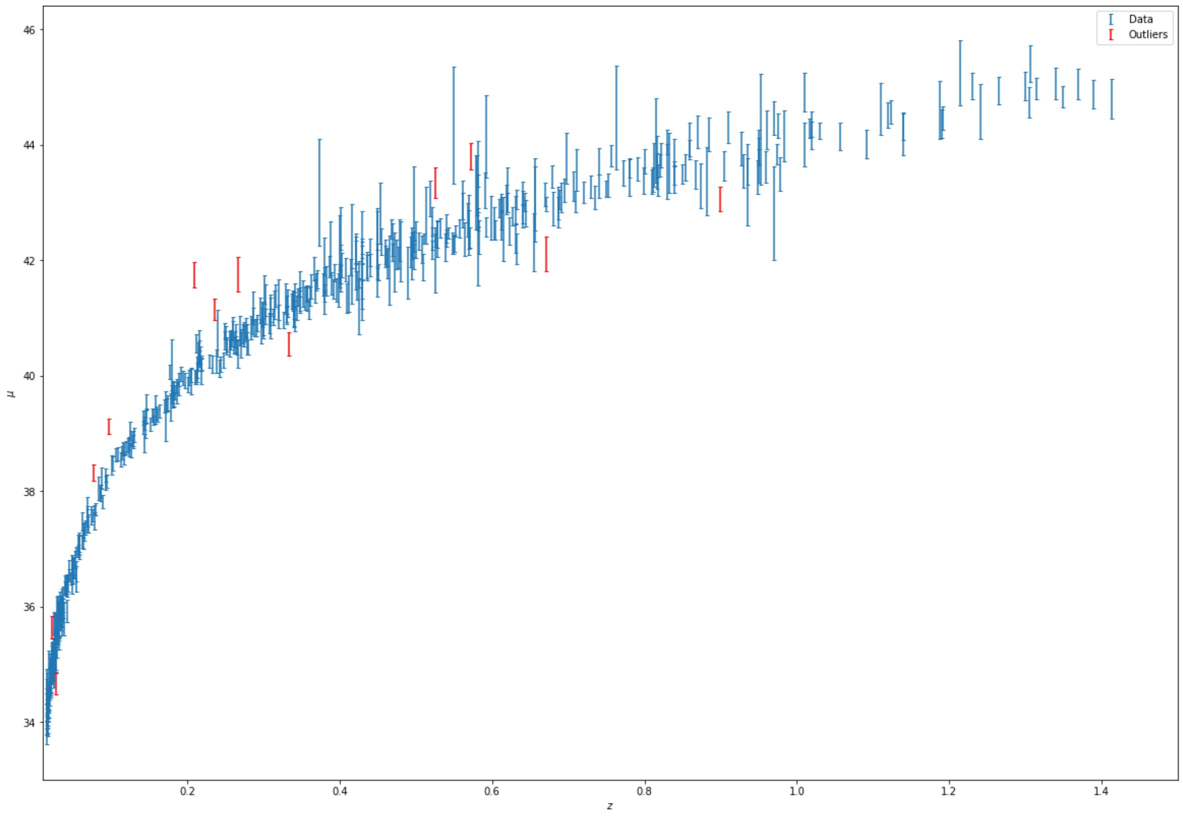
Now, include the distance modulus of 12 additional supernovae, which are not-so-good standard candles. They are 3σ away from the best-fit mode.

In [ ]:
```
# with outliers included
data = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6/sn_z_
# z
z_data = data[:,0]
# mu
mu_data = data[:,1]
# error on mu (sigma(mu))
mu_err_data = data[:,2]
```

So we have a total of 592 supernovae, and we can see that the last 12 supernovae seem to be outliers. (i.e. mu_data[580:] contains the distance modulus measurements of these 12 supernovae.)

4. Plot the measurements of all 592 supernovae (with errorbar). Show the last 12 supernovae (outliers) with a different color.

In [ ]:
```
plt.figure(figsize = (20,14))

plt.errorbar(
  z_data[:-12], mu_data[:-12], yerr=mu_err_data[:-12], fmt="none", cap
  label="Data", c="C0",
)

plt.errorbar(
  z_data[-12:], mu_data[-12:], yerr=mu_err_data[-12:], fmt="none", cap
  label="Outliers", c="red",
)

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



5. With 12 outliers included, run few parallel sequences of Metropolis algorithm simulations using the package "emcee". Print your constraints on w and Ω_m and plot 1-d posterior of w and Ω_m as well as 2-d posterior using the package "corner". Make sure that your chains have converged.

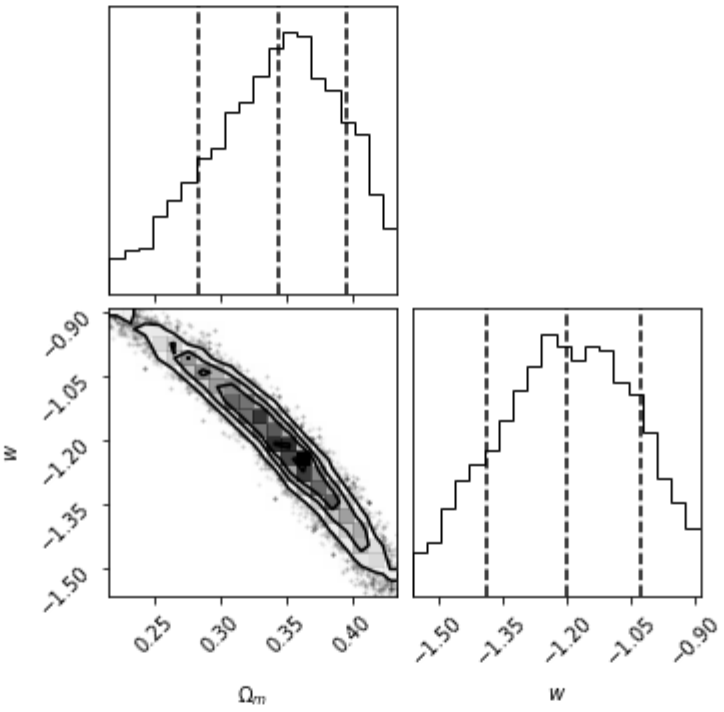
In [ ]:
```
trace_out = np.empty((nchains, nwalkers*(nsteps-nburn), nparams))

for i in range(nchains):
  x0 = np.random.normal(loc=map, scale=scale, size=(nwalkers, nparams)
  sampler = emcee.EnsembleSampler(nwalkers, nparams, log_posterior)
  sampler.run_mcmc(x0, nsteps)
  trace_out[i] = sampler.chain[:, nburn:, :].reshape(-1, nparams)
```

In [ ]:
```
R = gelman_rubin(trace_out)
print("R:")
for i in range(len(R)):
  print(f"{params[i]}: {R[i]:.5f}")
```

```
R:
Omega_m: 1.00963
w: 1.00887
```

The chains converged by the Gelamn-Rubin test (R < 1.1).

In [ ]:
```
fig = corner.corner(trace_out.reshape(-1, nparams), labels=["$\\Omega_
```



In [ ]:
```
omega_out, w_out = trace_out.mean(axis=(0, 1))
# compute variance for each chain and avg across chains
emcee_out_var = np.var(trace_out, axis=1).mean(axis=0)
omega_std_out, w_std_out = np.sqrt(emcee_out_var)
```

In [ ]:
```
print('(outliers included) MCMC result - naive model:')
print('Omega_m = ', omega_out, '+/-' , omega_std_out)
print('w = ', w_out, '+/-' , w_std_out)
```

```
(outliers included) MCMC result - naive model:
Omega_m =  0.3394297453909932 +/- 0.05555740538586726
w =  -1.2061919725799715 +/- 0.17342713346859673
```

Remember that in HW5, we used the Gaussian mixture to better model the measurements with outliers. Let us apply the same technique in this case.

$$L = \prod_{i=1}^{N_{\rm SN}} \Big[\frac{g}{\sqrt{2\pi\sigma(\mu_i)^2}} \exp\Big(-\frac{1}{2} \frac{[\mu_{i,\,data}(z_i) - \mu_{i,\,model}(z_i, \Omega_m, w)]^2}{\sigma(\mu_i)^2} \Big) + \frac{1-g}{\sqrt{2\pi\sigma_B^2}} \exp\Big($$

Here, we have 5 free parameters: $\Omega_m, w, g, \sigma_B, \Delta\mu$.

With outliers, we think there is something in the noise we do not really understand, which makes error distribution non-Gaussian. So we hope adding a second Gaussian to the model would better describe the pdf. g determines weights on the two Gaussians. σ_B^2 is the variance of the second Gaussian, which we assume to be larger than the variance of the first Gaussian. $\Delta\mu$ is the distance modulus offset in the second Gaussian.

6. Using emcee, run few parallel sequences of Metropolis algorithm simulations with this new model. Print your constraints on w and Ω_m and plot 1-d posterior of w and Ω_m as well as 2-d posterior using the package "corner". Make sure that your chains have converged.

In [ ]:
```
def log_posterior_mix(param):
  Omegam, w, g, sigmaB, dmu = param
  if(Omegam<=0 or w>=0 or g<0 or g>1):
    lnP = -1.e100
  else:
    dL = distL(z_data, Omegam, w)
    mu_model = dist_mod(dL)
    res = mu_data - mu_model
    exp_gauss = -1/2 * (res/ mu_err_data)**2
    gauss = g / np.sqrt(2*np.pi * mu_err_data**2) * np.exp(exp_gauss)
    expB = -1/2 * (res + dmu)**2 / sigmaB**2
    background = (1-g) / np.sqrt(2*np.pi*sigmaB**2) * np.exp(expB)
    lnP = np.sum(gauss + background)
  return lnP
```

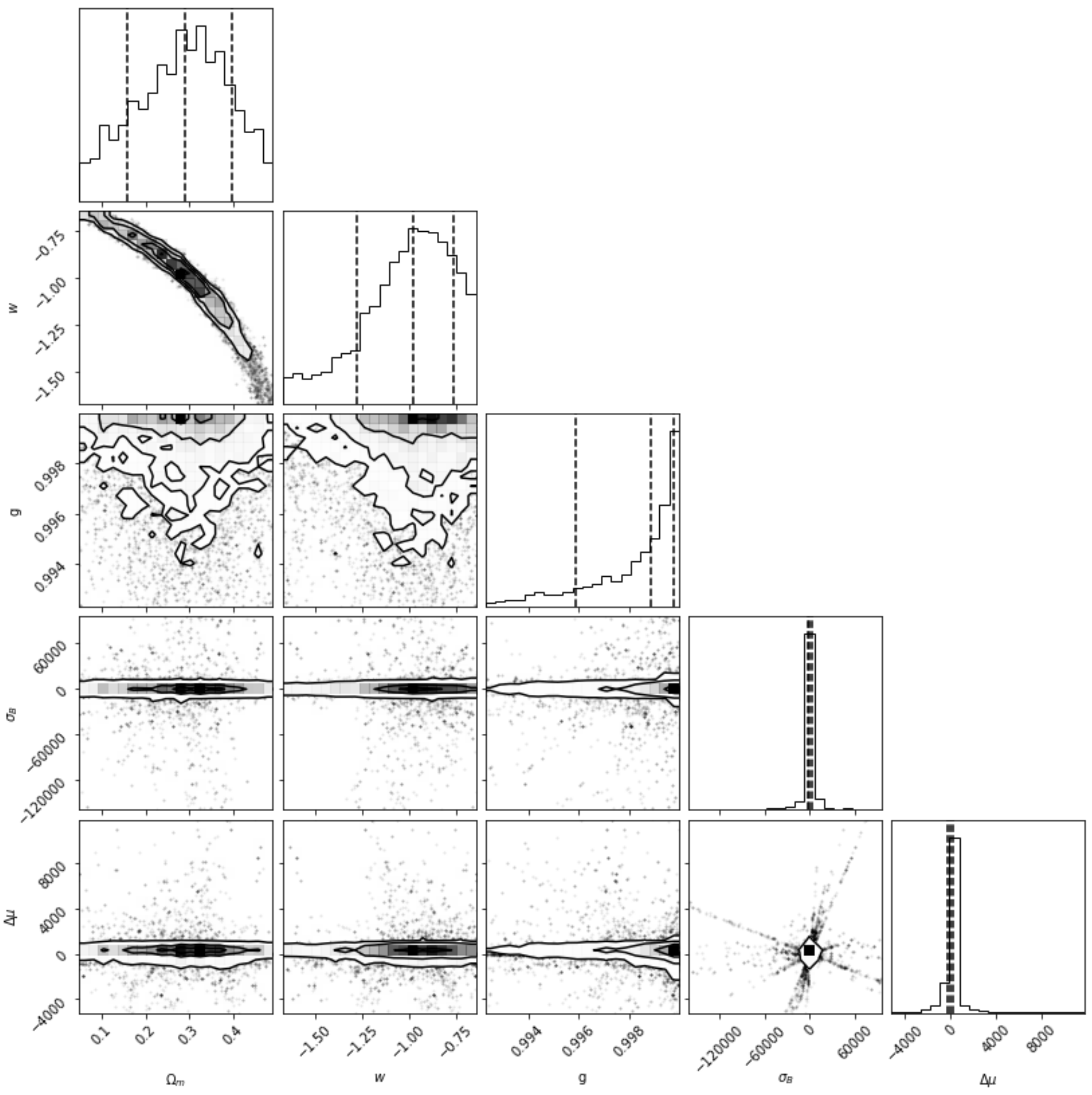
In [ ]:
```
nparams_mix = 5
trace_mix = np.empty((nchains, nwalkers*(nsteps-nburn), nparams_mix))

# set iniital position to be MAP from before with g=0.5, sigmaB=50, dm
x0_mean = np.concatenate((map, np.array([0.5, 50, 5])))
x0_scale = 0.1 * np.abs(x0_mean)

for i in range(nchains):
  x0 = np.random.normal(loc=x0_mean, scale=x0_scale, size=(nwalkers, n
  sampler = emcee.EnsembleSampler(nwalkers, nparams_mix, log_posterior
  sampler.run_mcmc(x0, nsteps)
  trace_mix[i] = sampler.chain[:, nburn:, :].reshape(-1, nparams_mix)
```

In [ ]:
```
params += ["g", "sigmaB", "delta_mu"]

R = gelman_rubin(trace_mix)
print("R:")
for i in range(len(R)):
  print(f"{params[i]}: {R[i]:.5f}")
```

```
R:
Omega_m: 1.10205
w: 1.07044
g: 1.04341
sigmaB: 1.01575
delta_mu: 1.00840
```

```
In [ ]: fig = corner.corner(
          trace_mix.reshape(-1, nparams_mix),
          labels=["$\\Omega_m$", "$w$", "g", "$\\sigma_B$", "$\\Delta \\mu$"],
          quantiles=[0.16, 0.5, 0.84],
          range = 0.95*np.ones(nparams_mix),
        )
```

```
WARNING:root:Too few points to create valid contours
```



```
In [ ]: omega_mix, w_mix = trace_mix.mean(axis=(0, 1))[:2]
        emcee_mix_var = np.var(trace_mix, axis=1).mean(axis=0)[:2]
        omega_std_mix, w_std_mix = np.sqrt(emcee_mix_var)[:2]
```

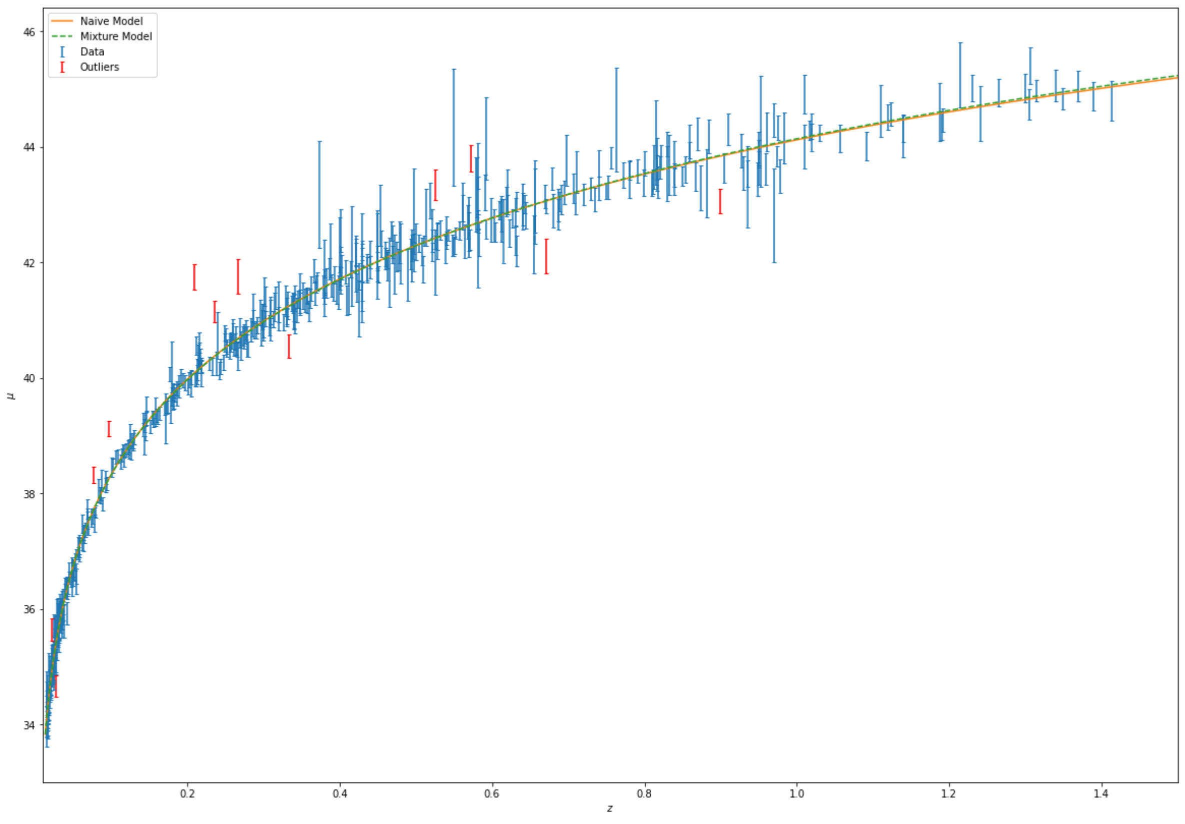
```
In [ ]: print('(outliers included) MCMC result - Gaussian mixture model:')
        print('Omega_m = ', omega_mix, '+/-' , omega_std_mix)
        print('w = ', w_mix, '+/-' , w_std_mix)
```

```
(outliers included) MCMC result - Gaussian mixture model:
Omega_m =  0.28370630929738594 +/- 0.10863000973629072
w =  -1.0461006481024386 +/- 0.38090607171652857
```

7. Using the estimates of Ω_m and w from your MCMC chain in Part 5 and Part 6, calculate the distance modulus from theory and plot the curves on top of the measured data. See how they fit.

```
In [ ]: zgrid = np.linspace(0.9*z_data.min(), 1.1*z_data.max(), num=1000)
        mu_out = dist_mod(distL(zgrid, omega_out, w_out))
        mu_mix = dist_mod(distL(zgrid, omega_mix, w_mix))
```

```
In [ ]: plt.figure(figsize = (20,14))

        plt.errorbar(
          z_data[:-12], mu_data[:-12], yerr=mu_err_data[:-12], fmt="none", cap
          label="Data", c="C0",
        )

        plt.errorbar(
          z_data[-12:], mu_data[-12:], yerr=mu_err_data[-12:], fmt="none", cap
          label="Outliers", c="red",
        )

        plt.plot(zgrid, mu_out, label="Naive Model", c="C1")
        plt.plot(zgrid, mu_mix, label="Mixture Model", c="C2", ls="--")

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



Reference: See pg. 8-16 (https://lear.inrialpes.fr/~jegou/bishopreadinggroup/chap9.pdf (https://lear.inrialpes.fr/~jegou/bishopreadinggroup/chap9.pdf))

For this Gaussian mixture model, we wish to maximize the likelihood function with respect to the parameters $g, \sigma_B, \Delta\mu$ for $\Omega_m = 0.3, w = -1$. In order to do this, we will apply the **expectation-maximization (EM)** algorithm. This is an iterative method to find maximum likelihood in the case where the model depends on the hidden/latent variable. Here, we call binary variable **a** as our latent variable such that $p(a_k = 1) = \pi_k$

Re-write the likelihood as:

$$L = \prod_{i=1}^{N_{\rm SN}} \Big[\frac{\pi_1}{\sqrt{2\pi\sigma(\mu_i)^2}} \exp\Big(-\frac{1}{2} \frac{[\mu_{i,\,data}(z_i) - \mu_{i,\,model}(z_i, \Omega_m = 0.3, w = -1)]^2}{\sigma(\mu_i)^2} \Big)$$
$$+ \frac{\pi_2}{\sqrt{2\pi\sigma_B^2}} \exp\Big(-\frac{1}{2} \frac{[\mu_{i,\,data}(z_i) - \mu_{i,\,model}(z_i, \Omega_m = 0.3, w = -1) - \mu_{\rm offset}]^2}{\sigma_B^2} \Big) \Big]$$

$$= \prod_{i=1}^{N_{\rm SN}} \Big[\pi_1 \cdot \mathrm{Normal}\big(\Delta\mu_i = \mu_{i,\,data} - \mu_{i,\,model} \big| \, \overline{\Delta\mu}_{\rm class\,1} = 0, \sigma(\mu_i)^2\big) + \pi_2 \cdot \mathrm{Normal}\big(\Delta\mu$$

where $\mu_{i,\,model}$ assumes $\Omega_m = 0.3, w = -1$. Suppose that we measure $\Delta\mu = \mu_{i,\,data} - \mu_{i,\,model}$. For the first Gaussian (expected to describe the distribution of 580 non-outlier, standard-candle supernovae), the mean value of $\Delta\mu$ is 0, and its variance is the measurement noise $\sigma(\mu)^2$. For the second Gaussian which expects to describe the distribution of 12 outliers, we assume that there will be some offset in μ ($\mu_{\rm offset}$), so the mean value of $\Delta\mu$ is $\mu_{\rm offset}$, and it has some unknown variance σ_B^2.

Now apply the EM algorithm.

1. First, initialize: choose $\pi_1 = 0.95$ and $\pi_2 = 0.05$. Let $\mu_{\rm offset} = 0, \sigma_B = 0.5$ initially.

2. **Expectation (E) step**: Evaluate the responsibilities using the current parameter values.

$$\gamma_{1,\,i} = \frac{\pi_1 \cdot \mathrm{Normal}\big(\Delta\mu_i \big| \, \overline{\Delta\mu}_{\rm class\,1}, \sigma(\mu_i)^2\big)}{\pi_1 \cdot \mathrm{Normal}\big(\Delta\mu_i \big| \, \overline{\Delta\mu}_{\rm class\,1}, \sigma(\mu_i)^2\big) + \pi_2 \cdot \mathrm{Normal}\big(\Delta\mu_i \big| \, \overline{\Delta\mu}_{\rm class\,2}, \sigma_B^2\big)}$$
$$\gamma_{2,\,i} = \frac{\pi_2 \cdot \mathrm{Normal}\big(\Delta\mu_i \big| \, \overline{\Delta\mu}_{\rm class\,2}, \sigma_B^2\big)}{\pi_1 \cdot \mathrm{Normal}\big(\Delta\mu_i \big| \, \overline{\Delta\mu}_{\rm class\,1}, \sigma(\mu_i)^2\big) + \pi_2 \cdot \mathrm{Normal}\big(\Delta\mu_i \big| \, \overline{\Delta\mu}_{\rm class\,2}, \sigma_B^2\big)}$$

   where $i = 1, \ldots, N_{SN}$ (number of measurements). Note that γ_1 and γ_2 are vectors of length N_{SN}. For a supernova i, $\gamma_{1,\,i}$ describes its probability of belonging to the first class (described by the first Gaussian). (Note: Therefore, in the end, we expect 12 outliers have much higher values of γ_2 than normal 580 supernovae - i.e. they have much greater probability of belonging to the second class. This is a systematic way to identify an outlier.)

3. **Maximization (M) step**: Re-estimate the parameters using the current responsibilities

$$\text{The mean } (\overline{\Delta\mu}_{\rm class\,1} = 0) \text{ and variance, } \sigma(\mu_{\rm i})^2, \text{ of the first Gaussian are fixed a}$$
$$N_1 = \sum_{i=1}^{N_{SN}} \gamma_{1,\,i}, \quad N_2 = \sum_{i=1}^{N_{SN}} \gamma_{2,\,i}$$
$$\overline{\Delta\mu}_{\rm class\,2} = \frac{1}{N_2} \sum_{i=1}^{N_{SN}} \gamma_{2,\,i} \cdot \Delta\mu_i$$
$$\sigma_B^2 = \frac{1}{N_2} \sum_{i=1}^{N_{SN}} \gamma_{2,\,i} \cdot (\Delta\mu_i - \overline{\Delta\mu}_{\rm class\,2})^2$$
$$\pi_1 = \frac{N_1}{N_{SN}}, \quad \pi_2 = \frac{N_2}{N_{SN}}$$

4. Evaluate the log-likelihood and check for convergence of either the parameters or the log likelihood. If the convergence criterion is not satisfied return to step 2.

8. Using EM, calculate the converged values of π_1, π_2, and N_2. N_2 is the total number of SN in the second class (can be identified as outliers). Iterate until you reach the convergence (parameters not changing). Then, print out the values of γ_2 and show that 12 outliers have

In [ ]:
```python
def normal(x, mean, var):
  amp = 1 / np.sqrt(2 * np.pi * var)
  z = (x-mean)**2 / var
  return amp * np.exp(-z/2)


mu_model = dist_mod(distL(z_data, 0.3, -1))  # nominal model
delta_mu = mu_data - mu_model

def em_likelihood(pi1, pi2, mu_off, sigmaB_sq):
  L = pi1 * normal(delta_mu, 0, mu_err_data**2)
  L += pi2 * normal(delta_mu, mu_off, sigmaB_sq)
  return np.log(L).sum()

def EM_step(pi1, pi2, mu_off, sigmaB_sq, return_gammas=False):
  N_SN = len(mu_data)
  # E step
  gamma1 = pi1 * normal(delta_mu, 0, mu_err_data**2)
  gamma2 = pi2 * normal(delta_mu, mu_off, sigmaB_sq)
  gamma_sum = gamma1 + gamma2
  gamma1 /= gamma_sum
  gamma2 /= gamma_sum

  if return_gammas:
    return gamma1, gamma2

  # M step
  N1 = gamma1.sum()
  N2 = gamma2.sum()
  mu_off = np.sum(gamma2 * delta_mu) / N2
  sigmaB_sq = np.sum(gamma2 * (delta_mu - mu_off)**2) / N2
  pi1 = N1 / N_SN
  pi2 = N2 / N_SN

  return pi1, pi2, mu_off, sigmaB_sq


conv_tol = 1e-7  # tolerance for convrergence
# iniitalize values for EM optimization
pi1 = 0.95
pi2 = 0.05
mu_off = 0
sigmaB = 0.5
sigmaB_sq = sigmaB**2
diff = 1
old_lh = em_likelihood(pi1, pi2, mu_off, sigmaB_sq)
i=0
while diff > conv_tol:
  pi1, pi2, mu_off, sigmaB_sq = EM_step(pi1, pi2, mu_off, sigmaB_sq)
  new_lh = em_likelihood(pi1, pi2, mu_off, sigmaB_sq)
  diff = np.abs(new_lh - old_lh)
  old_lh = new_lh
  i+=1

print(f"Took {i} terations to converge.")

# compute gammas like in the EM step
gamma1, gamma2 = EM_step(pi1, pi2, mu_off, sigmaB, return_gammas=True)
print("gamma2:")
print("Expected outliers:", gamma2[-12:])
print("Maximum value of remaining parameters:", np.max(gamma2[:-12]))
```

```
Took 25 terations to converge.
gamma2:
Expected outliers: [0.97180723 0.87928525 0.99407263 0.99366972 0.9999
```

```
 0.99981385 0.60957329 0.34684037 0.92070605 0.92463803 0.77520654]
Maximum value of remaining parameters: 0.6356660660955238
```

Problem 2 - Back to Quasar

In HW3, we performed Principal Component Analysis (PCA) on the quasar (QSO) spectra from the Sloan Digital Sky Survey (SDSS); we filtered for high S/N to apply the standard PCA and selected 18 high-S/N spectra of QSOs with redshift 2.0 < z < 2.1, trimmed to $1340 < \lambda < 1620\ \mathring{A}$. Then, using the first three principal eigenvectors from the covariance matrix, we reconstructed each of the 18 QSO spectra.

In this assignment, we do Expectation Maximization PCA with and without per-observation weights. We use a simple noise fit of PCA components to individual spectra. Finally, using a Gaussian process, we compute the posterior distribution of the QSO's true emission spectrum and sample from it.

The following analysis is based on https://arxiv.org/pdf/1208.4122.pdf (https://arxiv.org/pdf/1208.4122.pdf), and https://arxiv.org/pdf/1605.04460.pdf (https://arxiv.org/pdf/1605.04460.pdf)

In [34]:
```
# Load data
wavelength = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6
X = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6/HW5_Prob
ivar = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6/HW5_P
```

In [35]:
```
# Data dimension
print( np.shape(wavelength) )
print( np.shape(X) )
print( np.shape(ivar) )
```

```
(824,)
(18, 824)
(18, 824)
```

In the above cell, we load the following data: wavelength in Angstroms ("wavelength"), a 2D array of spectra x fluxes ("X"), and another 2D array of inverse variances $(1/\sigma^2)$ of the flux array ("ivar").

We have 824 wavelength bins, so "X" is a 18 $\times$ 824 matrix, each row containing fluxes of different QSO spectra and each column containing fluxes in different wavelength bins. (e.g. X[i,j] is the measured flux of QSO i in wavelength bin j.) Similarly, "ivar" is a 18 $\times$ 824 matrix. (e.g. ivar[i,j] is the inverse variance of the flux of QSO i in wavelength bin j.)

Remember that in HW3, we computed the eigenvectors of the covariance of the quasars, sorted by their descending eigenvalues; we call them the principal components (henceforth denoted by ϕ). Suppose that we have k eigenvectors, each of length 824. Construct the matrix of eigenvectors $\phi = [\phi_1\ \phi_2\ \ldots\ \phi_k]$, with ϕ_i the ith principal eigenvector.

We can reconstruct the data as:

$$\hat{X} = \mu + \sum_k c_k \phi_k$$

where μ is the mean of the initial dataset and c_k is the reconstruction coefficient for eigenvector ϕ_k.

More specifically, we define μ as:

μ =

$$\begin{bmatrix} \overline{x}_1 & \overline{x}_2 & \dots & \overline{x}_{824} \end{bmatrix}$$

The mean-centered data matrix X_c can be defined as:

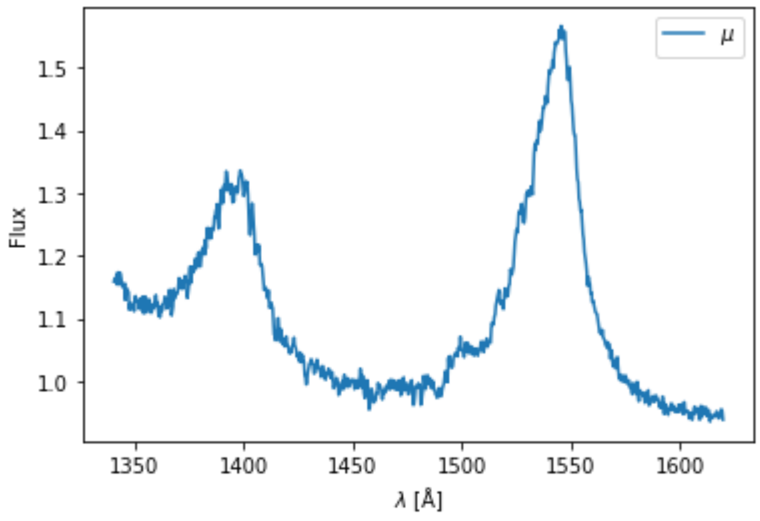
$X_c = X - \mu$ =

$$\begin{bmatrix} x_{(1,1)} - \overline{x}_1 & x_{(1,2)} - \overline{x}_2 & \dots & x_{(1,824)} - \overline{x}_{824} \\ x_{(2,1)} - \overline{x}_1 & x_{(2,2)} - \overline{x}_2 & \dots & x_{(2,824)} - \overline{x}_{824} \\ \vdots & \vdots & \vdots & \vdots \\ x_{(18,1)} - \overline{x}_1 & x_{(18,2)} - \overline{x}_2 & \dots & x_{(18,824)} - \overline{x}_{824} \end{bmatrix}$$

where $x_{m,n}$ denote the flux of mth QSO in nth wavelength bin, and $\overline{x}_k$ is the mean flux in kth wavelength bin.

1. Plot μ as a function of wavelength λ.

In [36]:
```
mu = X.mean(axis=0)

plt.figure()
plt.plot(wavelength, mu, label="$\\mu$")
plt.xlabel("$\\lambda$ [Å]")
plt.ylabel("Flux")
plt.legend()
plt.show()
```



"Expectation Maximization (EM) is an iterative technique for solving parameters to maximize a likelihood function for models with unknown hidden (or latent) variables. Each iteration involves two steps: finding the expectation value of the hidden variables given the current model (E-step), and then modifying the model parameters to maximize the fit likelihood given the estimates of the hidden variables (M-step)." (https://arxiv.org/pdf/1208.4122.pdf (https://arxiv.org/pdf/1208.4122.pdf))

Now, do Expectation Maximization PCA. In this case, we wish to solve for the eigenvectors, and the latent variables are the coefficients c. The likelihood is "the ability of the eigenvectors to describe the data."

First, find the eigenvector ϕ_1 with the highest eigenvalue (the first principal eigenvector):

1. Initialize: Let ϕ is a random vector of length 824.

2. **E-step**: For each QSO j,

$$c_j = X_{row\ j} \cdot \phi$$

   Here, "$\cdot$" represents a dot product, so $X_{row\ j}$ and ϕ are vectors of length 824, so c_j is a number. $c = [c_1\ c_2\ \ldots\ c_{18}]$ is a vector of length 18 (because we have 18 QSOs in this problem). So for each QSO j, we solve the coefficient c_j which best fits that QSO using ϕ.

3. **M-step**:

$$\phi = \frac{\sum_j c_j\ X_{row\ j}}{\sum_j c_j^2}$$

   Using the coefficients c_j, we update ϕ to find the vector which best fits the data given c_j.

4. Normalize:

$$\phi = \frac{\phi}{|\phi|}$$

5. Iterate until converged. Once converged, $c_1 = c$, and $\phi_1 = \phi$

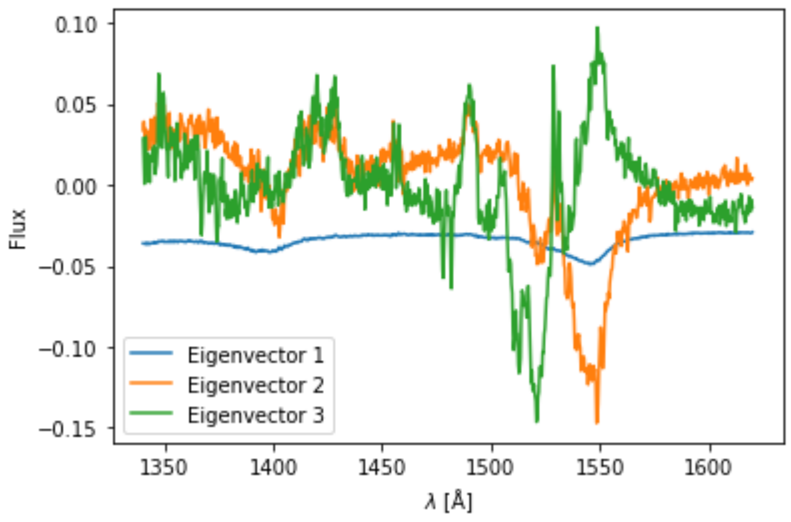
After you get ϕ_1, subtract the projection of ϕ from X ($X - c_1 \otimes \phi_1$, where "$\otimes$" is the outer product (https://en.wikipedia.org/wiki/Outer_product (https://en.wikipedia.org/wiki/Outer_product)). c_1 is a vector of length 18, and ϕ_1 is a vector of length 824, so $c_1 \otimes \phi_1$ is a 18×824 matrix.) and repeat the EM algorithm.

(So to find ϕ_2, you should use a data matrix $X - c \otimes \phi_1$. To find ϕ_2, use $X - c_1 \otimes \phi_1 - c_2 \otimes \phi_2$), and so on.

2. Using EM PCA, find the first three principal eigenvectors ϕ_1, ϕ_2, ϕ_3 and plot them as a function of wavelength.

In [46]:
```
def EM_PCA(data, nvecs, conv_tol=1e-7):
  """
  Run EM PCA for a data set using ``nvecs'' eigenvectors. Convergence
  as when the difference in norm between two vectors are below conv_to
  """
  nspec, nchans = data.shape
  evecs = np.empty((nvecs, nchans))
  cvals = np.empty((nvecs, nspec))

  # make a copy of the data that we can subtract eigenvectors from
  X_mat = data.copy()
  for i in range(len(evecs)):
    # initialize phi
    phi = np.random.normal(size=nchans)
    phi /= np.sqrt(np.sum(phi**2))  # normalize
    diff = 1  # iniitalize diff between new and previous phi
    while diff > conv_tol:
      c = X_mat @ phi
      new_phi = np.sum(c[:, None] * X_mat, axis=0)
      new_phi /= np.sum(c**2)
      new_phi /= np.sqrt(np.sum(new_phi**2))  # normalize
      diff = np.sum((new_phi - phi)**2)
      phi = new_phi
    evecs[i] = phi
    cvals[i] = c
    X_mat -= np.outer(c, phi)

  Xhat = np.sum([np.outer(cvals[k], evecs[k]) for k in range(len(cvals

  return Xhat, evecs
```

In [38]:
```
Xhat, evecs = EM_PCA(X, 3)

plt.figure()
for i, ev in enumerate(evecs):
  plt.plot(wavelength, ev, label=f"Eigenvector {i+1}")
plt.xlabel("$\\lambda$ [Å]")
plt.ylabel("Flux")
plt.legend()
plt.show()
```

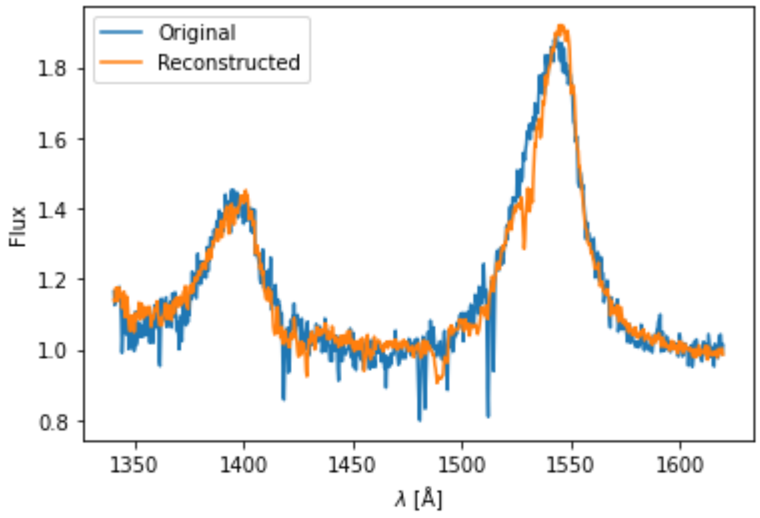


Finally, reconstruct the data using the first principal eigenvectors:

$$\hat{X} = \sum_{k=1}^{3} c_k \otimes \phi_k$$

3. For any one QSO spectra, plot the original and reconstructed spectra, using the above equation.

In [39]:
```
plt.figure()
plt.plot(wavelength, X[0], label="Original")
plt.plot(wavelength, Xhat[0], label="Reconstructed")
plt.xlabel("$\\lambda$ [Å]")
plt.ylabel("Flux")
plt.legend()
plt.show()
```



Alternatively, you can also reconstruct the data using "PC scores." (Call the PC score matrix Z)

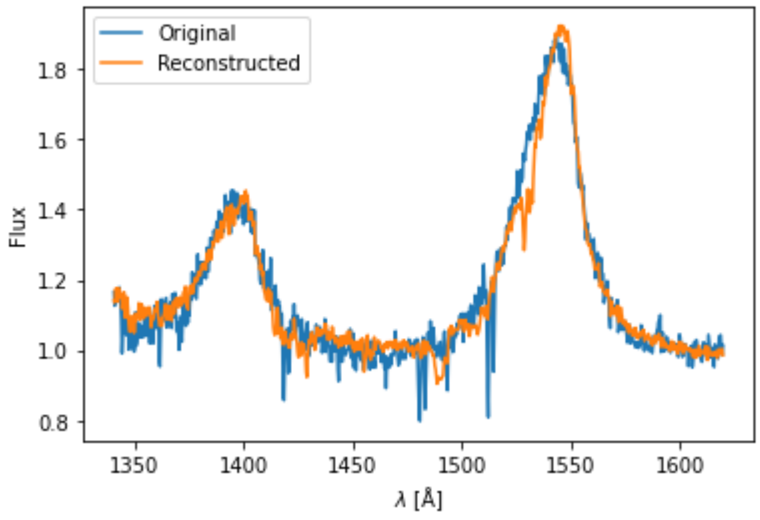
$$Z = X_c \phi$$

Then, we can reconstruct the data by mapping it back to 824 dimensions with $\phi^{\mathbf{T}}$:

$$\hat{X} = \mu + Z\phi^T$$

4. For any one QSO spectra, plot the original and reconstructed spectra, using PC scores.

In [40]:
```
Xc = X - mu
Z = Xc @ evecs.T
Xhat_PC = mu + Z @ evecs

plt.figure()
plt.plot(wavelength, X[0], label="Original")
plt.plot(wavelength, Xhat_PC[0], label="Reconstructed")
plt.xlabel("$\\lambda$ [Å]")
plt.ylabel("Flux")
plt.legend()
plt.show()
```



Now, include noisier QSO spectra.

In [41]:
```
# Load data
wavelength = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6
X = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6/HW5_Prob
ivar = np.loadtxt("/content/drive/My Drive/P188_288/P188_288_HW6/HW5_P
```

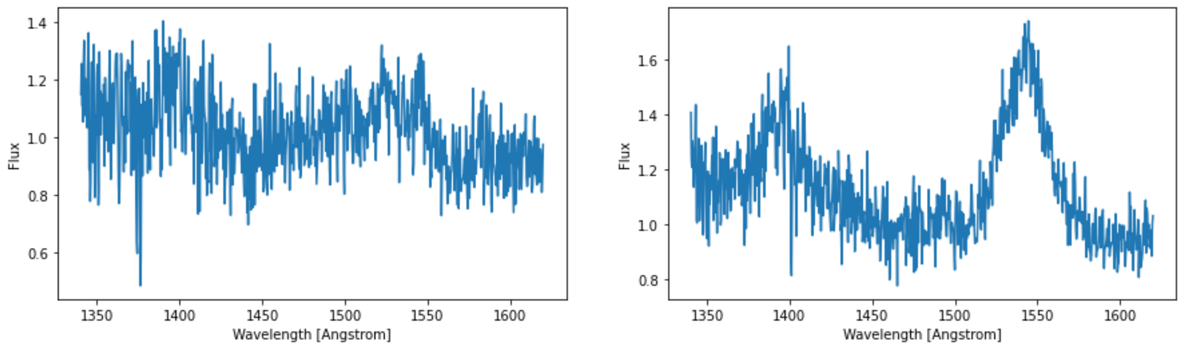
In [42]:
```
ivar[ivar==0] = 1.e-4
```

In [43]:
```
# Data dimension
print( np.shape(wavelength) )
print( np.shape(X) )
print( np.shape(ivar) )
```

```
(824,)
(2562, 824)
(2562, 824)
```

We now have 2562 quasars (including 18 high S/N quasars we had before). The below cell plots the spectra of two quasars; you can see how noisy they are.

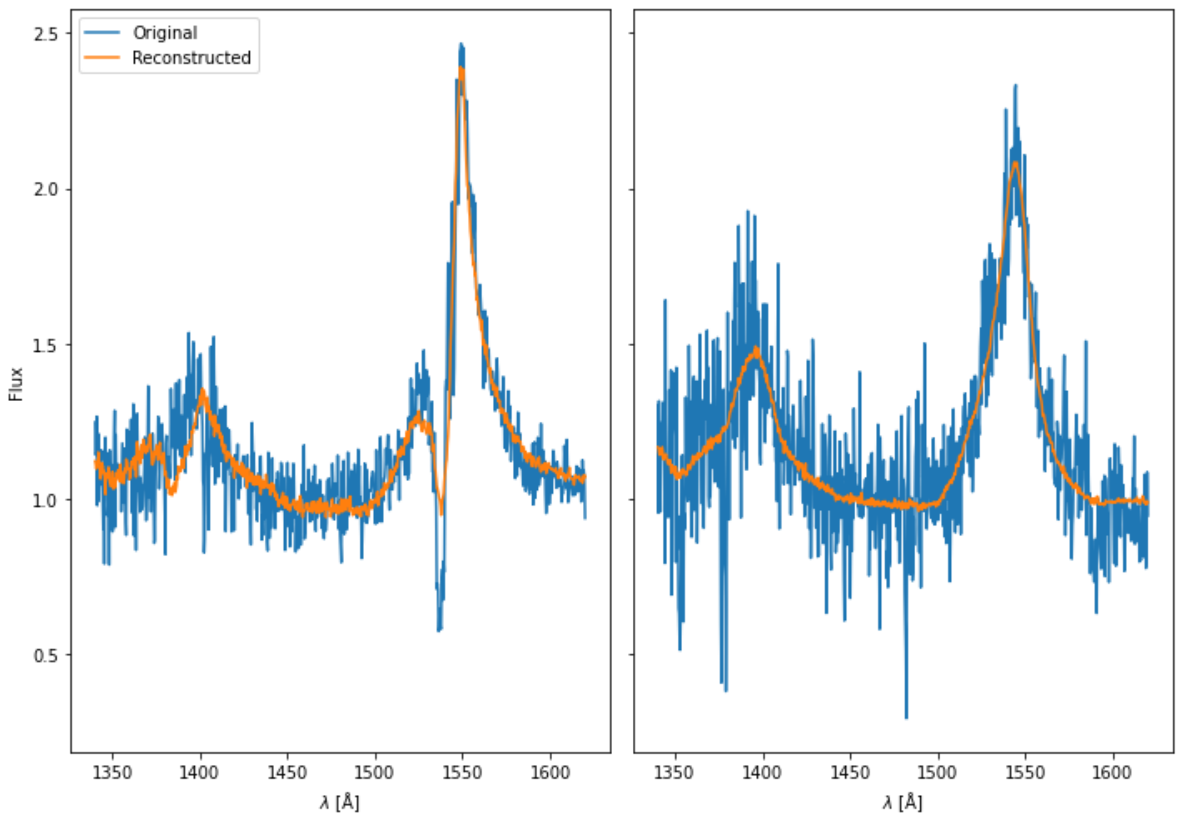
In [44]:
```
fig, axes = plt.subplots(1,2,figsize=(15,4))
ax = axes[0]; i = 50
ax.plot(wavelength, X[i,:])
ax.set_xlabel('Wavelength [Angstrom]'); ax.set_ylabel('Flux')

ax = axes[1]; i = 500
plt.plot(wavelength, X[i,:])
ax.set_xlabel('Wavelength [Angstrom]'); ax.set_ylabel('Flux')
plt.show()
```



Now, perform EM PCA on 2562 quasars.

5. Using EM PCA, find the first 10 principal eigenvectors $\phi_1, \phi_2, \ldots, \phi_{10}$ and reconstruct the data using them. ($\hat{X} = \sum_{k=1}^{10} c_k \otimes \phi_k$) For any two spectra, plot the original and reconstructed spectra.

```
In [48]: Xhat, evecs = EM_PCA(X, 10)

         fig, axs = plt.subplots(figsize=(10, 7), ncols=2, sharex=True, sharey=
         for i, ax in enumerate(axs.ravel()):
           ax.plot(wavelength, X[i], label="Original")
           ax.plot(wavelength, Xhat[i], label="Reconstructed")
           ax.set_xlabel("$\\lambda$ [Å]")
         axs[0].set_ylabel("Flux")
         axs[0].legend()
         plt.tight_layout()
         plt.show()
```



So far we treated all data equally when solving for the eigenvectors. However, we find that some data have considerably larger measurement noise, and they can unduly influence the solution. Now, we perform EM PCA with per-observation weights (called weighted EMPCA) so that the high S/N data receive greater weight. (See https://arxiv.org/pdf/1208.4122.pdf (https://arxiv.org/pdf/1208.4122.pdf) for more detailed explanation. The following description is paraphrased from this paper.)

Basically, we add weights w to the measured data in M-step: $\phi = \sum_j w_j \; c_j \; X_{row\ j}$

In this case, the situation is more complicated since the measured flux in each wavelength bin for each quasar has a different weight. So we cannot do a simple dot product to derive c; instead, we must solve a set of linear equations for c. Similarly, M-step must solve a set of linear equations to update ϕ instead of just performing a simple sum. Hence, the weighted EMPCA starts with a set of random orthonormal vectors and iterates over.

1. Initialize: Let ϕ is a set of random orthonormal vectors.

In [57]:
```
# Create an aray of random orthonormal vectors
# Reference: https://github.com/sbailey/empca
def _random_orthonormal(nvec, nvar, seed=1):
    """
    Return array of random orthonormal vectors A[nvec, nvar]
    Doesn't protect against rare duplicate vectors leading to 0s
    """

    if seed is not None:
        np.random.seed(seed)

    A = np.random.normal(size=(nvec, nvar))
    for i in range(nvec):
        A[i] /= np.linalg.norm(A[i])

    for i in range(1, nvec):
        for j in range(0, i):
            A[i] -= np.dot(A[j], A[i]) * A[j]
            A[i] /= np.linalg.norm(A[i])

    return A

# Number of quasars
nQSO = len(X)
# Number of wavelength bins
nLambda = len(wavelength)
# Number of eigenvectors we want
nEigvec = 10

# A set of random orthonormal vectors
phi = _random_orthonormal(nLambda, nEigvec, seed=1)
```

1. **E-step**: $X_{row\ j} = \phi\ c_{col\ j}$. ($X_{row\ j}$ refers to jth row of X, and $c_{col\ j}$ is jth column of c. Note that X is a matrix of dimension "nQSO" x "nLambda", ϕ is a matrix of dimension "nLambda" x "nEigvec", and c is a matrix of dimension "nEigvec" x "nQSO".) Solve for c assuming weights w.

   We define weight w as the inverse variance ("ivar"). (So w is a matrix of dimension "nQSO" x "nLambda") This makes sense. "We weight the measured data by the estimated measurement variance so that noisy observations do not unduly affect the solution, while allowing PCA to describe the remaining signal variance."

   Now, solve $X_{row\ j} = \phi\ c_{col\ j}$ for $c_{col\ j}$ with weights $w_{row\ j}$. More generally, let $A = \phi, x = c_{col\ j}, b = X_{row\ j}, w = w_{row\ j}$:

   $$b = Ax$$

   $$wb = wAx$$

   $$A^T wb = (A^T wA)x$$

   $$(A^T wA)^{-1} A^T wb = x$$

   Hence, we get:

   $$c_{col\ j} = (\phi^T w_{row\ j}\ \phi)^{-1}\ \phi^T w_{row\ j}\ X_{row\ j}$$

In the below cell, we define the function "_solve."
_solve(A, b, w) solves $Ax = b$ with weights w. This function solves $Ax = b$ with weights w using $x = (A^T wA)^{-1} A^T wb$

In [50]:
```
# Solve Ax = b with weights w using the above set of equations
# Reference: https://github.com/sbailey/empca
import scipy
def _solve(A, b, w):
    """
    Solve Ax = b with weights w; return x

    A : 2D array
    b : 1D array length A.shape[0]
    w : 1D array same length as b
    """

    #- Apply weights
    # nvar = len(w)
    # W = dia_matrix((w, 0), shape=(nvar, nvar))
    # bx = A.T.dot( W.dot(b) )
    # Ax = A.T.dot( W.dot(A) )

    b = A.T.dot( w*b )
    A = A.T.dot( (A.T * w).T )

    if isinstance(A, scipy.sparse.spmatrix):
        x = scipy.sparse.linalg.spsolve(A, b)
    else:
        x = np.linalg.lstsq(A, b)[0]

    return x
```

Now, in the E-step, for each QSO j, we can solve $X_{row\ j} = \phi c_{col\ j}$ for $c_{col\ j}$ with weights $w_{row\ j}$ using the function "*solve".*

*Similarly in the M-step, for each wavelength bin j, we can solve $X*{col\ j} = c^T \phi*{row\ j}for \phi*{row\ j}$with weights$w_{col\ j}$using the function "_s olve ". < br >< br >< span style =" color : blue "><i> 6.l$
Iterate 20 times. (See Undergrad version for hints) </i></span>

In [62]:
```
c = np.empty((nEigvec, nQSO))

# make a copy of the data that we can subtract eigenvectors from
niter = 20  # number of iterations
for i in range(20):
  # E step
  for j in range(nQSO):
    c[:, j] = _solve(phi, X[j], ivar[j])
  # M step
  for j in range(nLambda):
    phi[j] = _solve(c.T, X[:, j], ivar[:, j])
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:25: Futur
To use the future default and silence this warning we advise to pass `
```

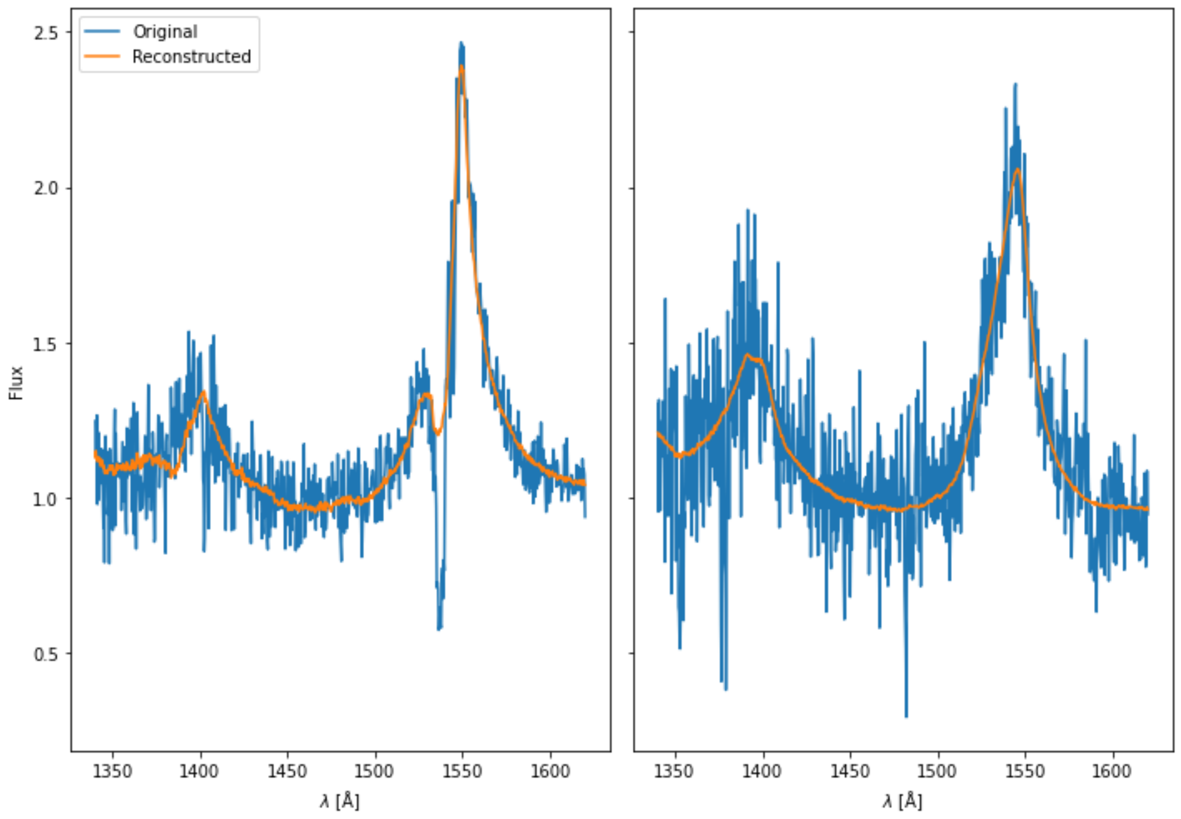
Reconstruct the data using ϕ:

$$\hat{X} = (\phi c)^T$$

ϕ is a matrix of dimension "nLambda" x "nEigvec", and c is a matrix of dimension "nEigvec" x "nQSO". So $\hat{X}$ is a matrix of dimension "nQSO" x "nLambda" as expected.

7. Reconstruct the data using the above equation. Remember that you chose two spectra in Part 5. For the same two spectra, plot the original and reconstructed spectra. Part 5 uses EMPCA without weights. Compared to Part 5, does your reconstructed spectra become less noisy?

In [63]:
```
Xhat = np.transpose(phi @ c)

fig, axs = plt.subplots(figsize=(10, 7), ncols=2, sharex=True, sharey=
for i, ax in enumerate(axs.ravel()):
  ax.plot(wavelength, X[i], label="Original")
  ax.plot(wavelength, Xhat[i], label="Reconstructed")
  ax.set_xlabel("$\\lambda$ [Å]")
axs[0].set_ylabel("Flux")
axs[0].legend()
plt.tight_layout()
plt.show()
```



These are indeed less noisy than before.