Markov Chain Monte Carlo methods

Statistics and Data Science

Spring 2025

Goals for today: you should be able to...

- Introduction to MCMC methods
- **lecture 26/27 notebook:**
  - Use emcee to perform MCMC evaluation of a multidimensional posterior distribution

- **Reminder**: please see Slack/Canvas for updates to schedule, and Slack me with any conflicts you have Wed. 4/23, Friday 4/25 or Monday 4/28!

MCMC and related methods

Review: Recipe for Maximum Likelihood

1. Write down $L=p(data \mid \boldsymbol{\theta})$, where $\boldsymbol{\theta}=[\theta_1, \theta_2,...]$; let $LL = \ln L$. Then $LL = \ln p(data \mid \boldsymbol{\theta})$
2. Either:
   A. Figure out the analytic derivatives, $\delta LL / \delta \theta_i$. Find the point(s) where all of these derivatives = 0, which indicates that they must be minima, maxima, or saddle points.
   B. Write a Python function that returns -*LL* for your actual dataset, given the value of the parameters. You can then use a variety of Python routines to find the minimum of that function (most can be applied via `scipy.optimize.minimize`)
   C. Set up a grid of possible values for $\boldsymbol{\theta}$. Calculate *LL* at each point of the grid; choose the maximum value; or
   D. apply Markov Chain Monte Carlo methods to explore $p(data \mid \boldsymbol{\theta})$
3. Check that your solution is actually a maximum of LL.

As the number of dimensions/free parameters gets large, **these methods each break down**... the "curse of dimensionality"

A. Figure out the analytic derivatives, $\delta LL / \delta \theta_i$. Find the point(s) where all of these derivatives = 0, which indicates that they must be minima, maxima, or saddle points.
B. Write a Python function that returns -*LL* for your actual dataset, given the value of the parameters. You can then use a variety of Python routines to find the minimum of that function (most can be applied via `scipy.optimize.minimize`)
C. Set up a grid of possible values for $\boldsymbol{\theta}$. Calculate *LL* at each point of the grid; choose the maximum value; or

Markov Chain Monte Carlo analysis provides an alternative

- **Markov Chain** -- a system for which the next step it takes is determined only by its current state, not its past history
- **Monte Carlo** - a calculation that proceeds via random sampling
  - MCMC is a family of methods that generate values distributed as a desired probability distribution -- generally, the posterior probability distribution function for a set of parameters
  - The points are generated by evolving states according to a rule which ensures the probability of going from state A to state B is proportional to the posterior value at state B divided by the posterior value at state A
  - As the number of steps goes to infinity (if things are well-behaved), the distribution of points generated should then be proportional to the posterior when following such a rule

The classic MCMC algorithm: Metropolis-Hastings

Algorithm 1 The procedure for a single Metropolis-Hastings MCMC step.

1: Draw a proposal $Y \sim Q(Y; X(t))$
2: $q \leftarrow [p(Y)\, Q(X(t); Y)]/[p(X(t))\, Q(Y; X(t))]$ *// This line is generally expensive*
3: $r \leftarrow R \sim [0, 1]$
4: **if** $r \leq q$ **then**
5: $\quad X(t+1) \leftarrow Y$
6: **else**
7: $\quad X(t+1) \leftarrow X(t)$
8: **end if**

- Pseudocode from Foreman-Mackey et al. 2013
  - *X(t)* is the state of the system (=set of parameter values) at step t
  - *Y* is a **potential** new state
  - *p(X)* represents the posterior value at a point in parameter space X
  - *Q(X(t);Y)* is the probability of transitioning from point Y to X(t), and *Q(Y;X(t))* is the probability of transitioning from point X(t) to Y; we can choose ~any rule we want for how to generate the possible next step
  - Most commonly, Q is a multivariate Gaussian distribution, in which case Q(Y; X) = Q(X; Y) by construction.

The classic MCMC algorithm: Metropolis-Hastings

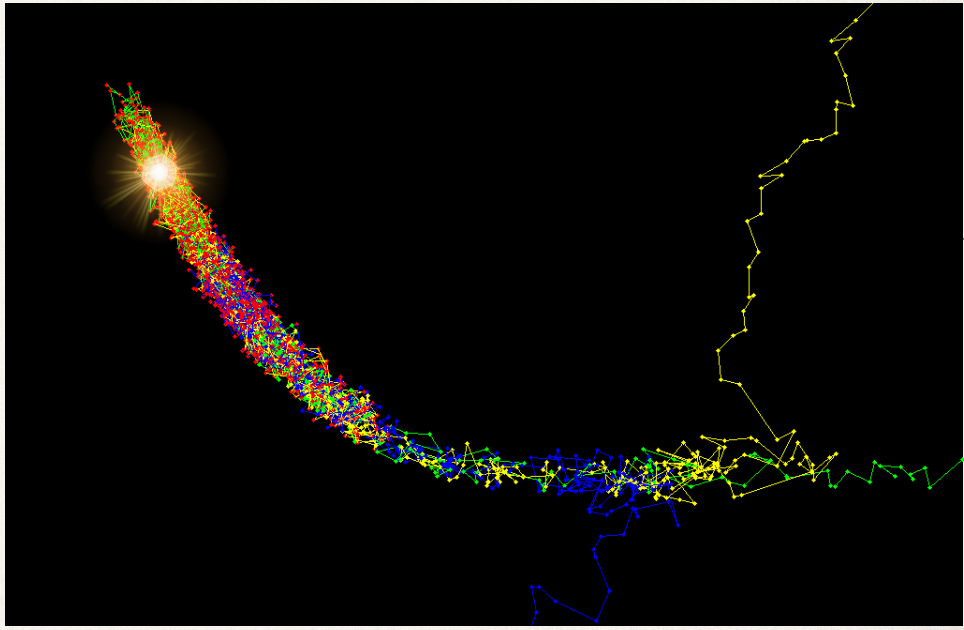
Algorithm 1 The procedure for a single Metropolis-Hastings MCMC step.

1: Draw a proposal $Y \sim Q(Y; X(t))$
2: $q \leftarrow [p(Y)\, Q(X(t); Y)]/[p(X(t))\, Q(Y; X(t))]$ *// This line is generally expensive*
3: $r \leftarrow R \sim [0, 1]$
4: **if** $r \leq q$ **then**
5: $\quad X(t+1) \leftarrow Y$
6: **else**
7: $\quad X(t+1) \leftarrow X(t)$
8: **end if**

- Pseudocode from Foreman-Mackey et al. 2013
  - *q* is the 'acceptance probability'
    - the way this is set up, if we change state from *X(t)* to *Y* with probability *q*, the distribution of state values will eventually converge to be proportional to *p(X)*
  - We generate a random number and compare to the acceptance probability *q*
  - *If r < q* we accept the step, so *X(t+1)* = Y; otherwise the chain stays in its current position

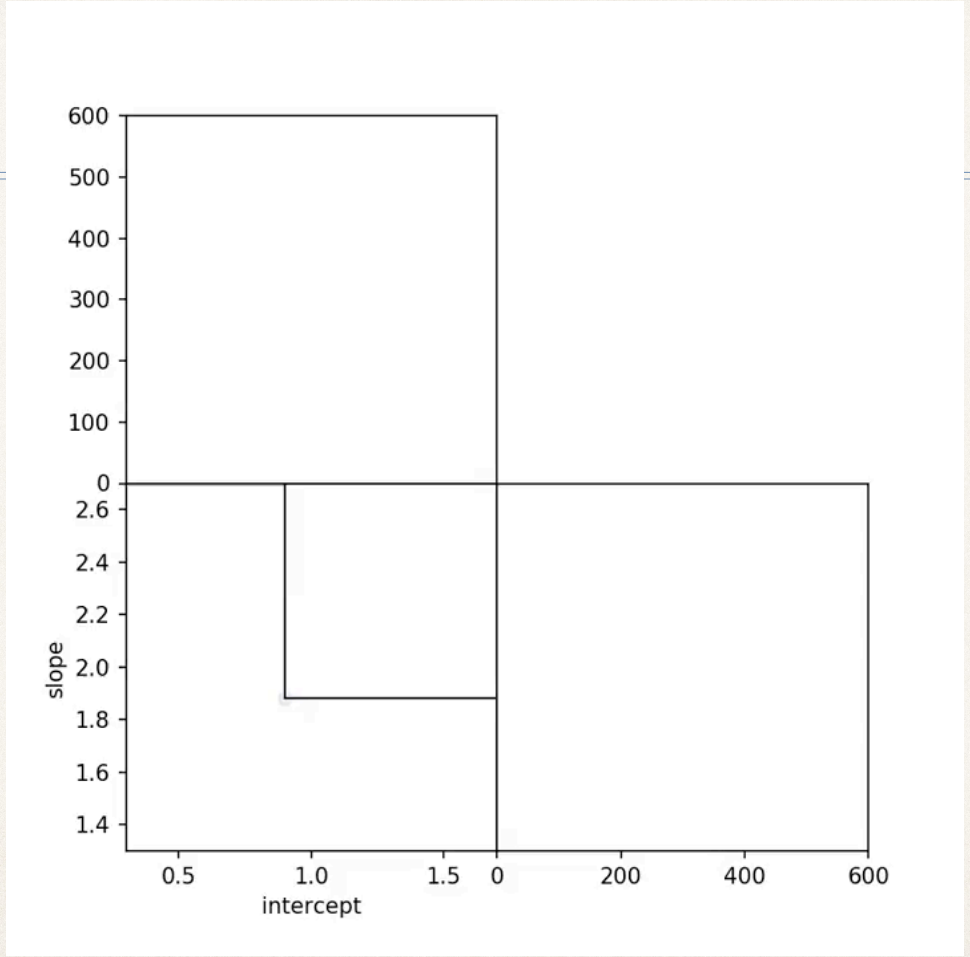
The net result...

- Transitions to states with higher posterior probability at Y than X are always accepted, but those with lower posterior probability are only accepted sometimes
- As a result the points in the chain tend to spend most of their steps in high-probability regions
  - After a 'burn-in' period that retains memory of whatever initial guess was used to initialize the chain, the distribution of points should match random draws from the posterior



- Figure from wikimedia.org . Red points are the ones after burn-in.

Animation of Metropolis sampling for fitting a line



Source: https://twiecki.io/blog/2014/01/02/visualizing-mcmc/

At this point many variants of MCMC exist

- I explored a number of MCMC options for this lecture, but the easiest to implement (by far) was the emcee sampler created by Foreman-Mackey et al. (http://dfm.io/emcee/current/ , described at https://arxiv.org/abs/1202.3665)

emcee: The MCMC Hammer

Daniel Foreman-Mackey[1,2], David W. Hogg[2,3], Dustin Lang[4,5], Jonathan Goodman[6]

ABSTRACT

We introduce a stable, well tested Python implementation of the affine-invariant ensemble sampler for Markov chain Monte Carlo (MCMC) proposed by Goodman & Weare (2010). The code is open source and has already been used in several published projects in the astrophysics literature. The algorithm behind emcee has several advantages over traditional MCMC sampling methods and it has excellent performance as measured by the autocorrelation time (or function calls per independent sample). One major advantage of the algorithm is that it requires hand-tuning of only 1 or 2 parameters compared to $\sim N^2$ for a traditional algorithm in an N-dimensional parameter space. In this document, we describe the algorithm and the details of our implementation. Exploiting the parallelism of the ensemble method, emcee permits *any* user to take advantage of multiple CPU cores without extra effort. The code is available online at http://dan.iel.fm/emcee under the MIT License.

The affine-invariant ensemble sampler

- Relies on sending a number of 'walkers' on individual chains, rather than only following one Markov chain
- Test points are along the line between a pair of walkers (instead of in a neighborhood of the original value)
- Can explore parameter space more efficiently than MH

Algorithm 2 A single stretch move update step from GW10

for $k = 1, \ldots, K$ **do**
  Draw a walker X_j at random from the complementary ensemble $S_{[k]}(t)$
  $z \leftarrow Z \sim g(z)$, Equation (10)
  $Y \leftarrow X_j + z\,[X_k(t) - X_j]$
  $q \leftarrow z^{N-1}\,p(Y)/p(X_k(t))$ *// This line is generally expensive*
  $r \leftarrow R \sim [0, 1]$
  if $r \leq q$, Equation (9) **then**
    $X_k(t+1) \leftarrow Y$
  else
    $X_k(t+1) \leftarrow X_k(t)$
  end if
end for

$$g(z) \propto \begin{cases} \dfrac{1}{\sqrt{z}} & \text{if } z \in \left[\dfrac{1}{a}, a\right], \\ 0 & \text{otherwise} \end{cases}$$

where a is an adjustable scale parameter that GW10 set to 2.

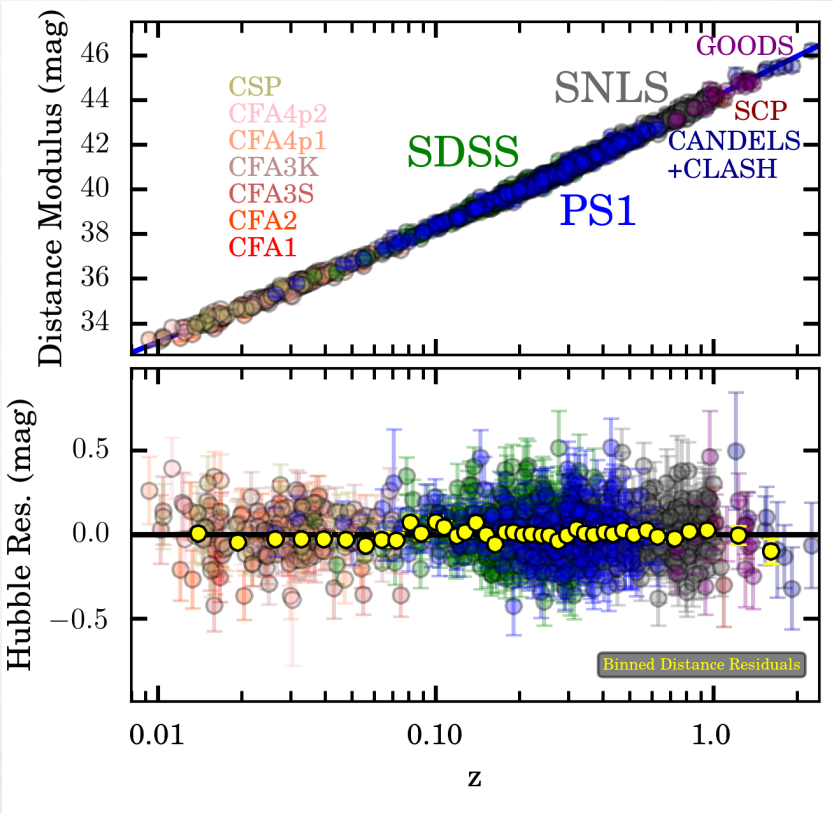
Applying MCMC methods to an astronomical dataset

- We will work with the Pantheon compilation of supernova distances from Scolnic et al. 2018
- For each SN in the Pantheon sample, we have redshift (z) and apparent magnitude, m_B, which we can convert to distance modulus (μ):

$$\mu = m\text{-}M = 5 \log_{10} (d_L / 10 \text{ pc})$$

where m is apparent magnitude, M is absolute magnitude, and d_L is luminosity distance.

- M is calibrated by correcting for deviations in brightness as a function of light curve shape & color, and assuming a value of the Hubble constant (M=-19.35 for $H_0 = 70$, which we'll adopt here).



Reading in the data

```
# CHANGE THE BELOW LINE TO POINT TO THE DIRECTORY CONTAINING pantheon.csv
path = './'

# the file is in comma-separated (csv)  format so we use read_csv
data=pd.read_csv(path+'pantheon.csv')

# for convenience, make variables for each column.
redshift = data['redshift'].values
mu = data['mu'].values
error = data['error'].values

# we want to sort the arrays to be in redshift order.
# We can get the set of indices that would sort array `x` with the function
#     x.argsort()
# This is a method of numpy arrays
# (i.e. a function that is attached to/provided by the variable)
index = redshift.argsort()

# sort all of our arrays
redshift=redshift[index]
mu=mu[index]
error=error[index]
```

We will compare the measured distance moduli to predicted moduli for different cosmological models

- The `astropy.cosmology` package can calculate a wide variety of cosmological quantities. We will use `cosmology.distmod()`, which calculates distance modulus.
- First, we need to choose a cosmological model. For simple dark energy models with equation of state $P = -\rho$ (as for a cosmological constant), the routine `cosmo.LambdaCDM` does this setup.
- The key parameters we are interested in are the fraction of the critical density of the Universe in dark matter (Ω_m) and the fraction in dark energy (Ω_Λ)
- We will fit for these parameters by calculating the posterior for them from the Pantheon dataset, via MCMC
- Get help on the `cosmo.LambdaCDM` class using ? . Then, in the next code box, we initialize it with H0=70, $\Omega_{m0} = 0.3$, and $\Omega_{de0}=0.5$; store the result in a variable named `cosmology` , and print out the contents of that variable.

```
omegam=0.3
omegal=0.5
cosmology=cosmo.LambdaCDM(H0=70,Om0=omegam,Ode0=omegal)
```

Calculating distance modulus

- Next, we want to calculate the distance modulus for our chosen cosmological model; we can do this with `model_modulus = cosmology.distmod(z)`

where **`z`** is an array of redshifts and `cosmology` is the cosmology description object set up by an initialization routine like `cosmo.LambdaCDM`

- To avoid any bugs later, we explicitly convert the result into a numpy array.

```
model=np.array( cosmology.distmod(redshift) )
```

Checking the residuals

- We next plot the data with our model curve overlaid. **Try to assess whether this is a good fit!**
- To clarify whether and where a model is failing, it is often helpful to plot the residuals: i.e., the data values (here, the mu array) minus the model.
- Ideally they should scatter evenly around zero.
- We next plot the residuals between mu and the model distance modulus for each supernova, as a function of redshift.
- **Assess whether this is a satisfactory fit to the data: why or why not?**

Making multi-panel plots

```
#  Code for putting  plots of both trend + residuals side-by-side

# set up a largely horizontal plotting space
plt.figure(figsize=[12,4])

# we'll make 2 plots: one of distance modulus vs. redshift,
# and one of residuals.

plt.subplot(1,2,1)
# distance modulus plot
plt.plot(redshift,mu,'b.')
plt.plot(redshift,model,'r-')
plt.xlabel('Redshift',fontsize=16)
plt.ylabel('Distance modulus',fontsize=16)

plt.subplot(1,2,2)
        # residuals plot
plt.plot(redshift,mu-model,'b.',alpha=0.2)
       # calculate the average error: note that we can get this from
       #     another method, .mean(),  attached to numpy arrays
mean_error = error.mean()

       # shade the 2 sigma region.  95% of the points (i.e. all but
       #   about 50, on average) should be in this region.
plt.fill_between([redshift.min(),redshift.max()],[-2*mean_error,-2*mean_error],
                         [2*mean_error,2*mean_error],alpha=0.1,color='purple')
```

Defining our likelihood, prior and posterior

- We will use the `emcee` package for our analysis; see http://dfm.io/emcee/current/ for documentation.
- Our MCMC calculations will need us to define functions for the prior, likelihood, and posterior. In our case, we assume the supernova distance modulus errors are all Gaussian, so the likelihood looks like

$$L = \prod \frac{1}{(2\pi\sigma^2)^{1/2}} \mathrm{e}^{\frac{-(\mu_i - \mu_{model,i})^2}{(2\sigma_i^2)}}$$

where μ_i is the distance modulus for the i'th supernova, $\mu_{model,i}$ is the model prediction for that supernova, and σ_i is the uncertainty in μ_i

- We calculate log likelihood instead of likelihood, and use $\log \Omega_m$ and $\log \Omega_\Lambda$ as the parameters we will fit for (to avoid any possibility of values of Ω_m *or* Ω_Λ below zero)

Defining our likelihood, prior and posterior

- You should be able to show: a prior which is flat in Ω_m is exponential in $\log \Omega_m$
- We return a prior value of 0 (so a log prior of -infinity) if the prior does not evaluate to be finite

```
def log_prior(theta):
    log_omegam, log_omegal = theta
# flat prior on omegas whereever omegam, omegal > 0,
# translated into priors on log quantities
    output = np.exp(log_omegam)*np.exp(log_omegal)
    output = np.log(np.minimum(output,1.))
# if the output is not finite,
# return a prior value of 0 (so log of prior = -infinity)
    if not np.isfinite(output):
        return -np.inf
    return output
```

Defining our likelihood, prior and posterior

- Here we again use Python exception handling (via the **try** and **except** keywords)
- If the code after **try** returns an error, we can execute different commands depending on what the error is

```
def log_likelihood(theta, redshift, mu, sigma):
# perform tuple unpacking on theta
    log_omegam, log_omegal = theta
# calculate distance modulus
    cosmology=cosmo.LambdaCDM(H0=70,Om0=np.exp(log_omegam),Ode0=np.exp(log_omegal))
#astropy is a bit fragile (and fairly slow) for this application.  We will do some exception
# handling to try to prevent crashes:
    try:
        mu_predicted = np.array( cosmology.distmod(redshift) )
    except ZeroDivisionError:
        mu_predicted=redshift*0.+40.
    except OverflowError:
        mu_predicted=redshift*0.+40.
    except RuntimeWarning:
        mu_predicted=redshift*0.+40.
    return -0.5 * np.sum(np.log(2 * np.pi * sigma ** 2)
                         + (mu - mu_predicted) ** 2 / sigma ** 2)
```

Defining our likelihood, prior and posterior

- `emcee` performs Bayesian inference; we need to provide it with the log of the posterior function
- We get this by combining the results of the log prior and log likelihood functions...

```
def log_posterior(theta, redshift, mu, sigma):
    return log_prior(theta) + log_likelihood(theta, redshift, mu, sigma)
```

Providing initial guesses

- `emcee` will run multiple Markov chains simultaneously; we will need to initialize each one to start somewhere.
- We'll start them all in the vicinity of a model with omegam = 0.4 and omegal = 0.5 (which we do not expect to be correct)

```
Nens = 100   # number of 'walkers' in  the ensemble

mean_omegam = 0.4     # mean of the Gaussian prior for initial guesses
sigma_omegam = 0.1 # standard deviation of the Gaussian prior

mean_omegal = 0.5     # mean of the Gaussian prior for initial guesses
sigma_omegal = 0.1 # standard deviation of the Gaussian prior

# create initial omega_m points:
initial_omegam = np.random.normal(mean_omegam, sigma_omegam, Nens)
# create initial omega_l points
initial_omegal = np.random.normal(mean_omegal, sigma_omegal, Nens)

# avoid negative values (nonsensical) by using 1E-3 as minimum possible value
initial_omegam = np.maximum(1E-2,initial_omegam)
initial_omegal = np.maximum(1E-2,initial_omegal)

# initial samples formatted to use as inputs
initial_samples = np.array([np.log(initial_omegam),
                            np.log(initial_omegal)]).T

ndims = initial_samples.shape[1] # number of parameters/dimensions
```

Basic parameters for running emcee

- We want to choose a 'burnin' number of steps we will ignore results from (as they still retain some memory of the initial guesses), and then a number of steps to continue beyond that point.

```
Nburnin = 80   # number of burn-in samples.  Often one will do >1000.
Nsamples = 100  # number of final posterior samples
```

Basic parameters for running emcee

- Next we initialize an Ensemble Sampler object:

```
import emcee # import the emcee package

print(f'emcee version: {emcee.__version__}')

# the data arrays used in calculating the posterior are passed
#  as 'arguments' in emcee:(the redshifts, the distance moduli,
#  and errors in our case).  We pass them as a tuple.
argslist = (redshift, mu, error)

# set up the sampler: we provide the number of walkers;
# the number of dimensions we are sampling over;
# the name of our posterior function;
# and then the arguments, which provide the data values.
sampler = emcee.EnsembleSampler(Nens, ndims, log_posterior,
                                args=argslist)
```

Doing all the work (this may take a few minutes)

```
# we will use this to see how long things take
from time import time

# pass the initial samples and total number of samples required

t0 = time() # store the time at start
sampler.reset()

# running with progress=True will show a progress bar
sampler.run_mcmc(initial_samples, Nsamples + Nburnin,
                 progress=True,store=True);
t1 = time() # store the time at end

print(f"Time taken to run 'emcee' is {t1-t0:.2f} seconds")

# extract the samples (removing the burn-in).
# The first index in sampler.chain runs over the different walkers;
#  the second, over the steps each walker takes; and the third,
#  over the different parameters being fit.
samples_emcee = sampler.chain[:, Nburnin:, :].reshape((-1, ndims))

lnprob_emcee = sampler.lnprobability[:,Nburnin:]
```

Ways to check results from MCMC

- 1) **Look at the acceptance rate of steps for each walker.** If this is near zero, the walkers are having trouble finding good parts of parameter space; if one, they aren't exploring at all. 20-50% is generally seen as ideal

```
print(f'Mean of acceptance fraction across walkers: {np.mean(sampler.acceptance_fraction):.4f}')

print(f'Std. dev. of acceptance fraction across walkers: {np.std(sampler.acceptance_fraction):.4f}')
```

Ways to check results from MCMC

- 2) **Look at the trajectory the walkers take in each parameter**. You should find that after an initial period where they start close to their initial position, they bounce around a more limited range once they find a good regime in parameter space. The initial part of the chains (the 'burn-in') should be discarded and not used in analysis.

```
plt.figure(figsize=(10,5))
for i in arange(0,Nens,10):
    plt.plot(np.exp(sampler.chain[i,:,0]),alpha=0.37)
plt.ylabel(r'$\Omega_m$')
```

Ways to check results from MCMC

- 3) Look at the distributions of values in the chains in one dimension (which should be proportional to the posterior for each parameter) and in each pair of dimensions. Make sure that if there are multiple islands of nonnegligible probability they are well- explored, and that there are not islands of probability concentrated around your initial guess.
- Dan Foreman-Mackey's `corner` Python package provides a good way of doing this
  - There is one **substantial** caveat: it draws contours that do NOT correspond to 68, 95%, etc. credible intervals, by default. See https://corner.readthedocs.io/en/latest/pages/sigmas/ for an explanation of what it does actually do.

Ways to check results from MCMC

```
import corner
plt.figure(figsize=(10,10))

# corner takes as input the values to be plotted against each other
# (as combined scatter/contour plots) and for which histograms will be shown,
# as well many optional inputs such  as labels for each parameter,
# true values of the parameters (or best-fit values) to overplot;
# whether to overplot e.g. 68% ranges on the histograms; and whether to add
# titles to each subplot with mean + uncertainty from the MCMC results.
fig = corner.corner(np.exp(samples_emcee), labels=["$\Omega_m$", "$\Omega_\Lambda$"],
                    truths=[0.284,0.716],bins=40, quantiles=[0.16, 0.5, 0.84],
                    show_titles=True,title_kwargs={"fontsize": 16},label_kwargs={"fontsize": 16})
```

Ways to check results from MCMC

- There are a couple of options to get the contour levels right:

A. A Bayesian credible region will contain (68%, 95%, etc.) of the total probability, and hence that fraction of the number of points. The smallest possible credible region will be the one that contains the (68%, 95%, etc.) of points with highest probability.
   - Conveniently, in corner the contour levels are set as quantiles; if you want just a 68% contour specify `levels=(0.68,)` (with the comma!); for 68 + 95%, do `levels = (0.68,0.95)`

B. If you are doing a frequentist (likelihood) analysis, the difference in -2 ln (likelihood) (compared to the maximum likelihood) evaluated at the true value of parameters should be distributed as a chi-squared distribution with (number of parameters) degrees of freedom. We can therefore set cutoff values of -2 ln (likelihood) corresponding to the 68, 95%, etc. confidence regions using the percentile point function for the chi-squared distribution. ln(likelihood) should be stored for all the points in your chain by the sampler.

Ways to check results from MCMC

- 4) Overlay your data with the best-fit model (or, better yet, an ensemble of models, each plotted with a small alpha) to make sure the data and model are a reasonable match.
- We can use the `sampler.flatlnprobability` attribute to find the chain step (combining all walkers, and including burnin) with the highest log posterior.
- The argmax attribute will be the index corresponding to that point.

```
# get the index in the flattened chain with the maximum probability

whmax=(sampler.flatlnprobability.argmax())

# determine omegam and omegal at that point.
omegam_opt,omegal_opt = np.exp(sampler.flatchain[whmax,:])
print(f'omegam: {omegam_opt:.4f}, omegal: {omegal_opt:.4f}')
```

Ways to check results from MCMC

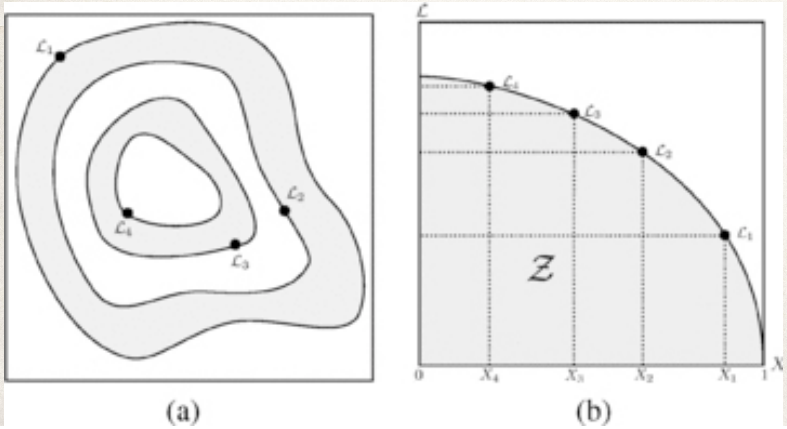
- 5) Check the autocorrelation time of your chains. See https://emcee.readthedocs.io/en/latest/tutorials/autocorr/ .
- To calculate the autocorrelation time (a measure of how fast the samples get uncorrelated), you can do: `sampler.get_autocorr_time()`
- Unfortunately with Nsamples=200 emcee is unable to calculate this quantity accurately, longer chains are needed.
- We want the autocorrelation time to be much less than the length of the chain.
- Some analyses will use only every N'th sample from a chain, where N >~ the autocorrelation time, to avoid using correlated samples.

Calculating the Evidence (for Bayesian model comparison)

- As we saw previously, the classic Bayesian method of determining the best model for a dataset is calculating the ratio of the evidences for each model.
- It turns out a simple calculation of the evidence using the the MCMC chain and the likelihood at the chain points is unstable
- Instead a better solution turns out to be a trick from statistical mechanics; the method is known as thermodynamic integration
  - See https://eriqande.github.io/sisg_mcmc_course/thermodynamic-integration.nb.html for details
- A class of MCMC-like algorithms known as nested sampling algorithms can do the necessary integrals

An alternative to MCMC: Nested sampling

- Nested sampling algorithms produce a set of contours of constant likelihood at many different likelihood levels, as well as a chain of points that can be reweighted to match the posterior
- Iteratively throw out lowest-likelihood point and replace it with a higher-likelihood one to explore the posterior
- Can calculate the evidence integral from the volume within contours of different likelihood levels
- E.g.: Multinest: https://academic.oup.com/mnras/article/398/4/1601/981502 or dynesty: https://dynesty.readthedocs.io/en/latest/index.html



Cartoon illustrating (a) the posterior of a two-dimensional problem and (b) the transformed $\mathcal{L}(X)$ function where the prior volumes X_i are associated with each likelihood $\mathcal{L}_i$.

Some useful resources

- *Bayesian inference and computation: a beginner's guide*: https://odysee.com/@BrendonBrewer:3/wsbook:f
  - This is a good general introduction to the ideas behind MCMC and nested samplers
- http://mattpitkin.github.io/samplers-demo/pages/samplers-samplers-everywhere/
  - This includes some examples for nested samplers, which are capable of calculating the Bayesian evidence correctly
- http://bjlkeng.github.io/posts/markov-chain-monte-carlo-mcmc-and-the-metropolis-hastings-algorithm/
- https://arxiv.org/abs/1411.5018 (Frequentism and Bayesianism: A Python-driven Primer by Vanderplas)
- https://ixkael.github.io/fitting-a-line-to-data-a-quick-tutorial/
  - includes examples of fitting lines with errors on X + Y via MCMC