Recap

The Bayesian data analysis workflow

1. Model the data generating process probabilistically.
   A. Understand the data
      - PCA
      - Bootstrap
      - Correlations
   B. Model the data & build a likelihood
      - Estimators
      - Non-Gaussian likelihoods
      - Fisher information & forecasts
      - Jeffreys prior

2. Condition the generative process on the observed data.
   - Sampling the posterior
     - MCMC (Metropolis-Hastings, slice sampling, Hamiltonian Monte Carlo)
     - Nested sampling
     - Simulation-based inference

3. Check that the results fit the observed data and improve the model.
   A. Goodness of fit
      - Posterior predictive distributions
      - Chi-square test
   B. Compare models
      - Evidences, Bayes' ratio
      - DIC, WAIC

The paper Bayesian Workflow, Gelman et al. 2021 gives a nice overview of these topics.

Things we did not cover

- Hierarchical modelling
- Random walks
- Charateristic functions, proof central limit theorem
- Importance sampling
- Summarising posteriors
- Gaussian processes & Gaussian random fields
- Non-Gaussian likelihoods
- Classification, AUROC

An analysis of the Pantheon+ SH0ES supernovae data set

Here we reproduce the main cosmological constraints from the paper The Pantheon+ Analysis: Cosmological Constraints by Brout et al. 2022.

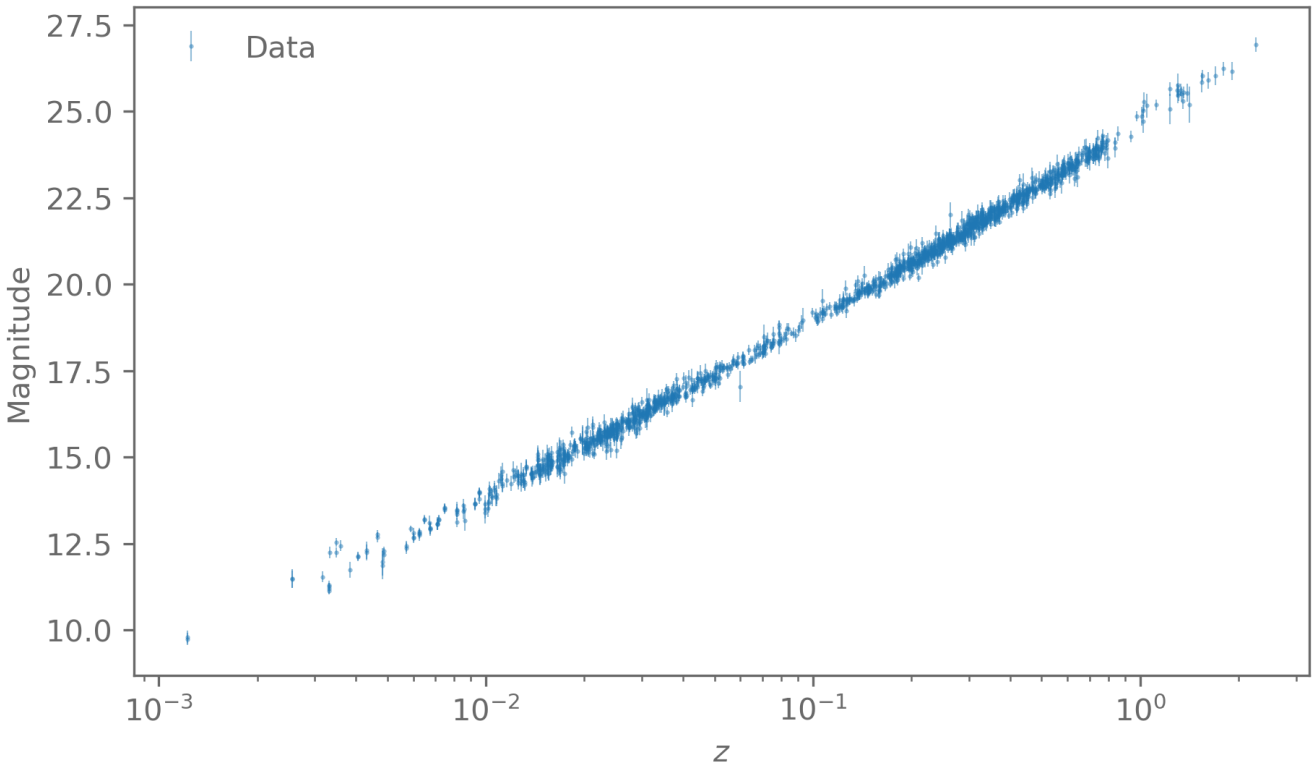
The main work of the paper is producing and characterising the data and the cosmological analysis can be done relatively easily.

```
import numpy as np
import scipy.stats

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

from jax import jit, vmap, hessian
import jax.random as random
import jax.numpy as jnp
from jax.lax import cond

import jaxopt

import emcee

import getdist
import getdist.plots

import tensorflow_probability.substrates.jax as tfp
tfd = tfp.distributions

import bayesian_stats_course_tools.analyse

import warnings
warnings.simplefilter("ignore")
```

```
import sys
sys.path.append("../projects/dark_energy_SN/")
from likelihood import JAXPantheonSH0ESDataModel
```

```
# This holds the data and takes care of the modelling
pantheon_sh0es = JAXPantheonSH0ESDataModel(
    data_file_name="../projects/dark_energy_SN/data/pantheon_sh0es.npz",
)
```

Plot the data.



Define the model and the likelihood.

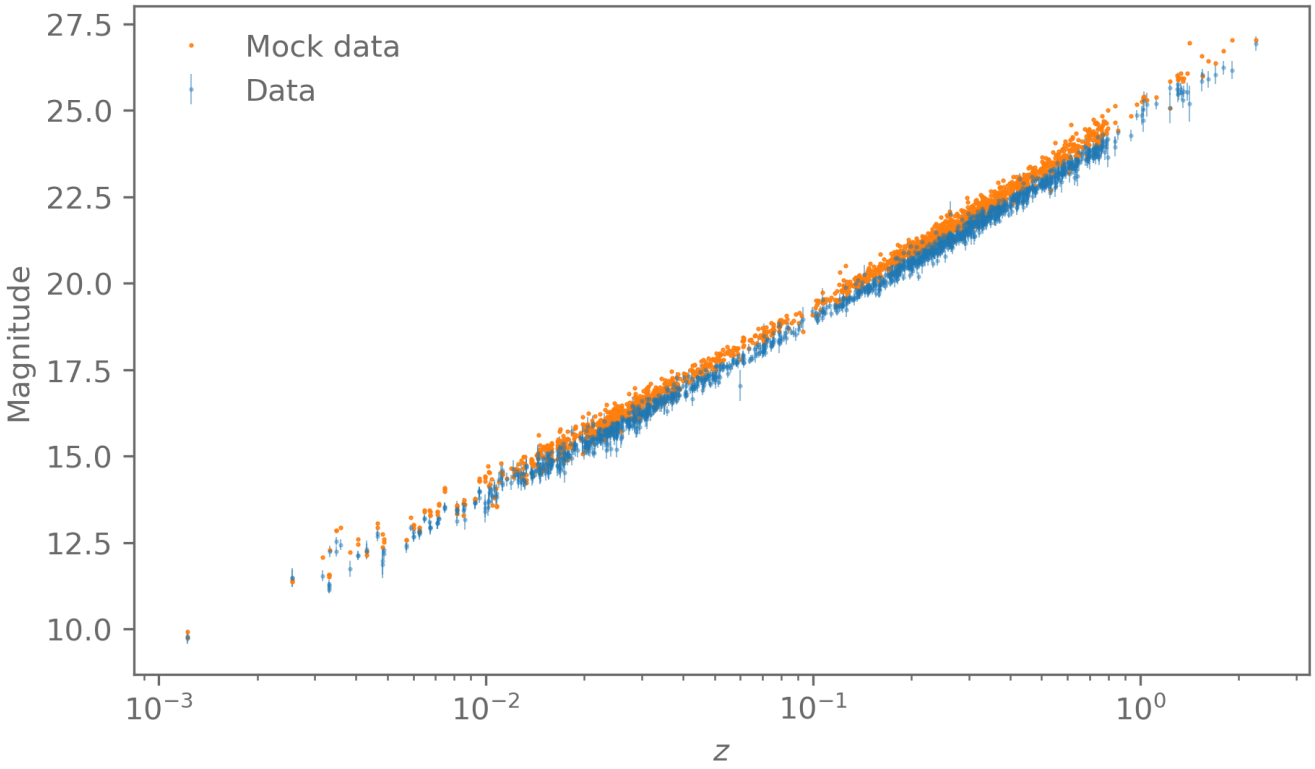
The model is relatively simple here. It mostly computes the distance given a redshift and from that the magnitude.

As in the paper we assume a Gaussian likelihood:

```
# We use a Gaussian likelihood
# To make sampling easier later on, we use the
# tensorflow-probability implementation
def create_likelihood_distribution(params):
    mu = pantheon_sh0es.model(params)

    return tfd.MultivariateNormalTriL(
        loc=mu, scale_tril=pantheon_sh0es.covariance_cholesky
    )
```

```
# Set up some initial parameters
param_names = ["Omega_m", "H0", "M"]
param_labels = [r"\Omega_m", "H_0", "M"]

params_initial = dict(
    Omega_m=0.3, H0=70.0, M=-19.0
```

```
)
# Some functions require an array instead of a dict
params_initial_flat = jnp.array([params_initial[i] for i in param_names])

initial_likelihood = create_likelihood_distribution(params_initial)

# Create a mock data vector from the likelihood
key, subkey = random.split(random.PRNGKey(42))
mock_data = initial_likelihood.sample(seed=subkey)
```



Now that we have defined a likelihood, let us make some forecasts on what the parameter constraints will look like user Fisher information matrices.

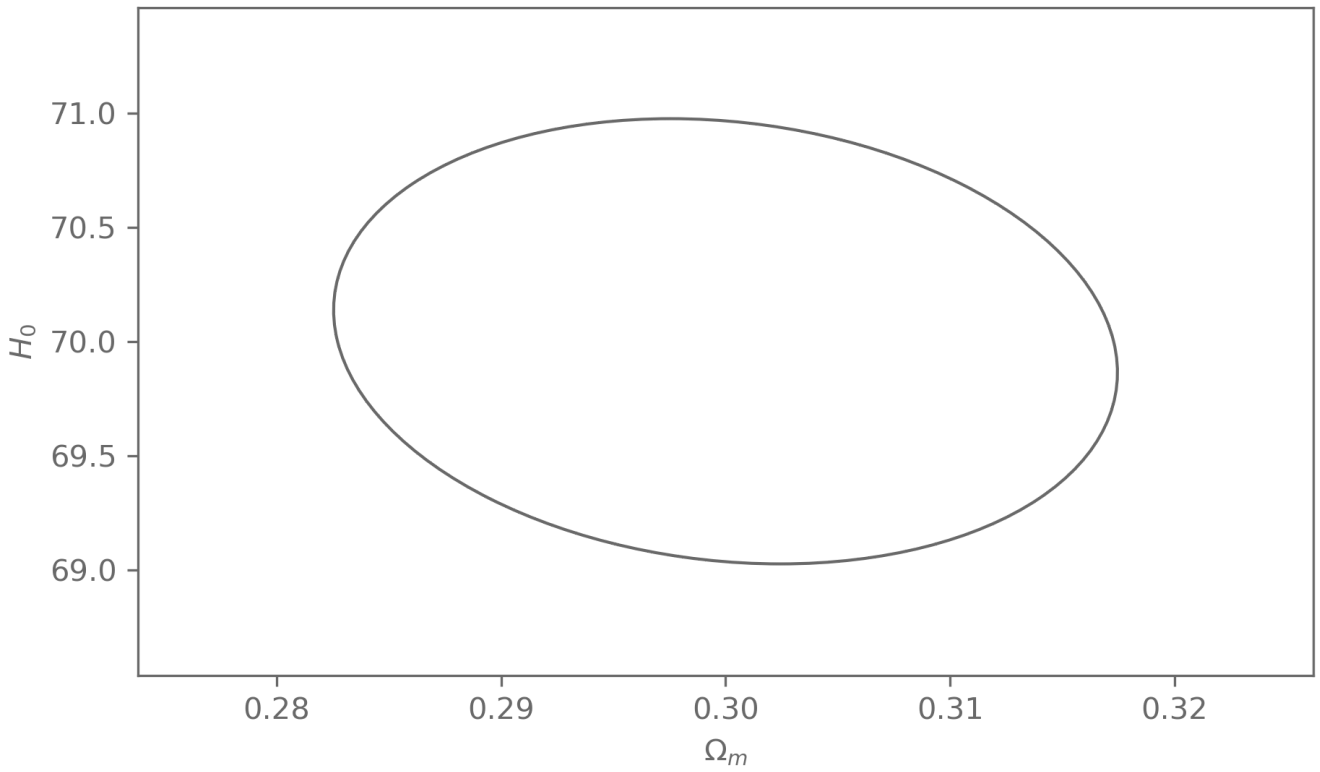
```
# Define a function that evaluates the log likelihood at a set of parameters,
# for a fixed datavector
def log_likelihood_fixed_data(params):
    log_L = create_likelihood_distribution(
        params
    ).log_prob(mock_data)
    return log_L

# Use JAX's autodiff tools to get the Hessian of the log likelihood
log_likelihood_hessian = jit(hessian(log_likelihood_fixed_data))
```

```
# Evaluate the Hessian and put it into a matrix form
# We need to do this because the parameters are a dict here.
# If they were in a flat array, we would get a matrix directly
log_likelihood_hessian_matrix = log_likelihood_hessian(params_initial)
fisher_matrix = -jnp.array(
    [[log_likelihood_hessian_matrix[i][j] for i in param_names]
```

```
     for j in param_names]
)
```

```
# Define a function to plot contour ellipses
def plot_contours(cov, pos,  nstd=1., ax=None, **kwargs):
    """
    Plot 2D parameter contours given a Hessian matrix of the likelihood
    From the jax-cosmo docs: https://jax-cosmo.readthedocs.io/en/latest/notebooks/jax-
    """

    def eigsorted(cov):
        vals, vecs = np.linalg.eigh(cov)
        order = vals.argsort()[::-1]
        return vals[order], vecs[:, order]

    sigma_marg = lambda i: np.sqrt(cov[i, i])

    if ax is None:
        ax = plt.gca()

    vals, vecs = eigsorted(cov)
    theta = np.degrees(np.arctan2(*vecs[:, 0][::-1]))

    # Width and height are "full" widths, not radius
    width, height = 2 * nstd * np.sqrt(vals)
    ellip = Ellipse(xy=pos, width=width,
                    height=height, angle=theta, **kwargs)

    ax.add_artist(ellip)
    sz = max(width, height)
    s1 = 1.5*nstd*sigma_marg(0)
    s2 = 1.5*nstd*sigma_marg(1)
    ax.set_xlim(pos[0] - s1, pos[0] + s1)
    ax.set_ylim(pos[1] - s2, pos[1] + s2)
    plt.draw()
    return ax
```

```
Expected uncertainty on Omega_m: 0.017
Expected uncertainty on H0: 0.97
```



Now define our priors and posterior to do things like finding the MAP and sampling from the posterior.

```
H_0_prior = tfd.Uniform(low=55, high=91)
Omega_m_prior = tfd.Uniform(low=0.1, high=0.9)
M_prior = tfd.Uniform(low=-20, high=-18)

def log_prior(params):
    return (
        H_0_prior.log_prob(params["H0"])
        + Omega_m_prior.log_prob(params["Omega_m"])
        + M_prior.log_prob(params["M"])
    )

def log_posterior(params, data):
    likelihood = create_likelihood_distribution(params)
    return likelihood.log_prob(data) + log_prior(params)
```

Because we are using JAX, we can just-in-time compile the posterior function, making it fast.

```
%timeit jit(log_posterior)(params_initial, pantheon_sh0es.data).block_until_ready()
```

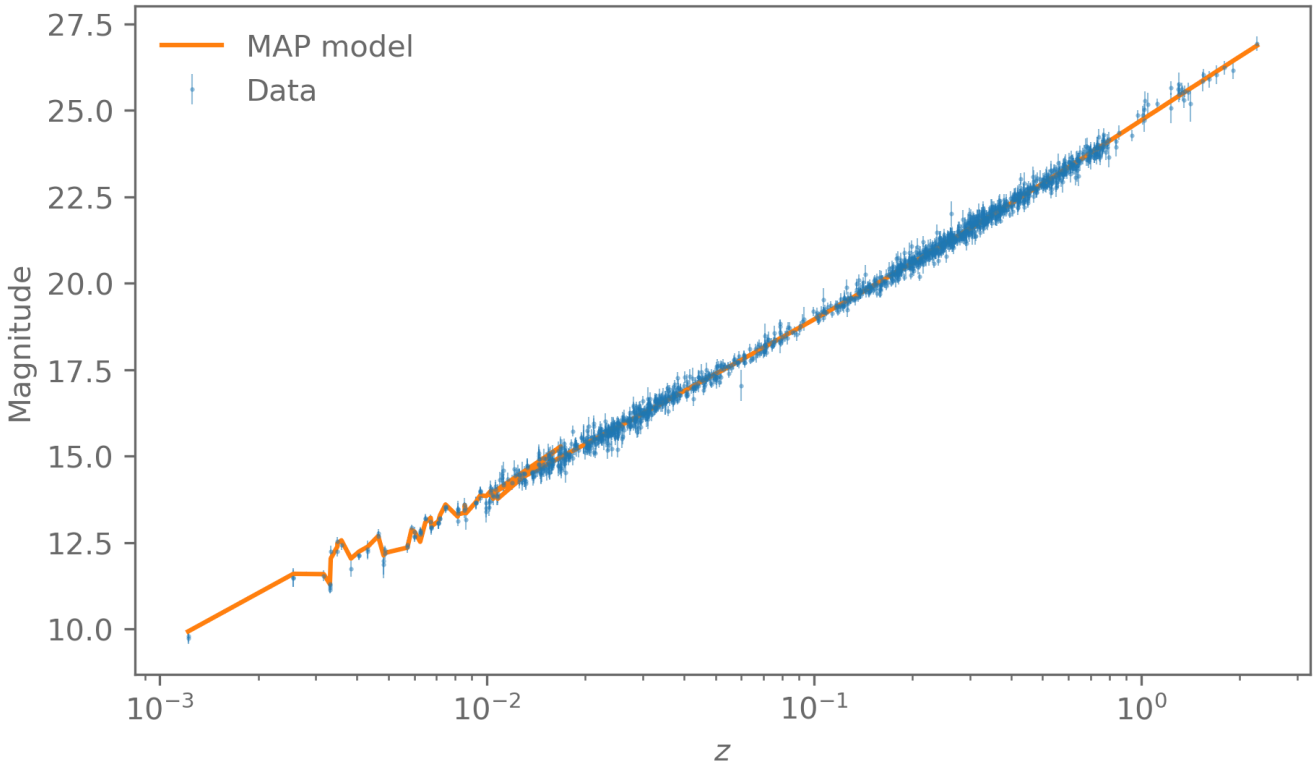
```
5.64 ms ± 637 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Find the MAP.

```
# We minimise the negative log posterior to find the maximum of the posterior
def neg_log_posterior(params, data):
    return -log_posterior(params, data)
```

```
solver = jaxopt.ScipyMinimize(fun=jit(neg_log_posterior), method="L-BFGS-B")
solution = solver.run(params_initial, data=pantheon_sh0es.data)
MAP_params = solution.params

print("MAP")
for name, value in MAP_params.items():
    print(f"{name} = {value:.3f}")
```

```
MAP
H0 = 73.738
M = -19.244
Omega_m = 0.326
```



Now to sampling from the posterior.

Here we use emcee to do the sampling. This wastes some of potential from JAX by not using gradients but the combination of vectorising the posterior evaluation and just-in-time compilation is still very powerful.

```
# Make wrapper around the posterior so it works with emcee
def log_posterior_wrapper(params):
    log_p = log_posterior(
        dict(Omega_m=params[0], H0=params[1], M=params[2]),
        data=pantheon_sh0es.data
    )
    # Use cond here to allow jit compilation
    log_p = cond(jnp.isfinite(log_p), lambda x: x, lambda x: -jnp.inf, log_p)
    return log_p

# Set the configuration for emcee
n_param = len(params_initial)
```

```
n_walker = 3*n_param
n_step = 5000

# Set initial positons for the walkers
params_init_walkers = (
    params_initial_flat + np.random.normal(scale=0.01, size=(n_walker, n_param))
)

sampler = emcee.EnsembleSampler(
    nwalkers=n_walker, ndim=n_param,
    log_prob_fn=jit(vmap(log_posterior_wrapper)),
    vectorize=True
)
state = sampler.run_mcmc(params_init_walkers, nsteps=n_step, progress=True)
```

```
100%|██████████| 5000/5000 [00:52<00:00, 94.60it/s]
```

We now need to check that the chain is well behaved and converged.

First check the integrated auto-correlation time to get a sense of how many independent samples we got in our chain.

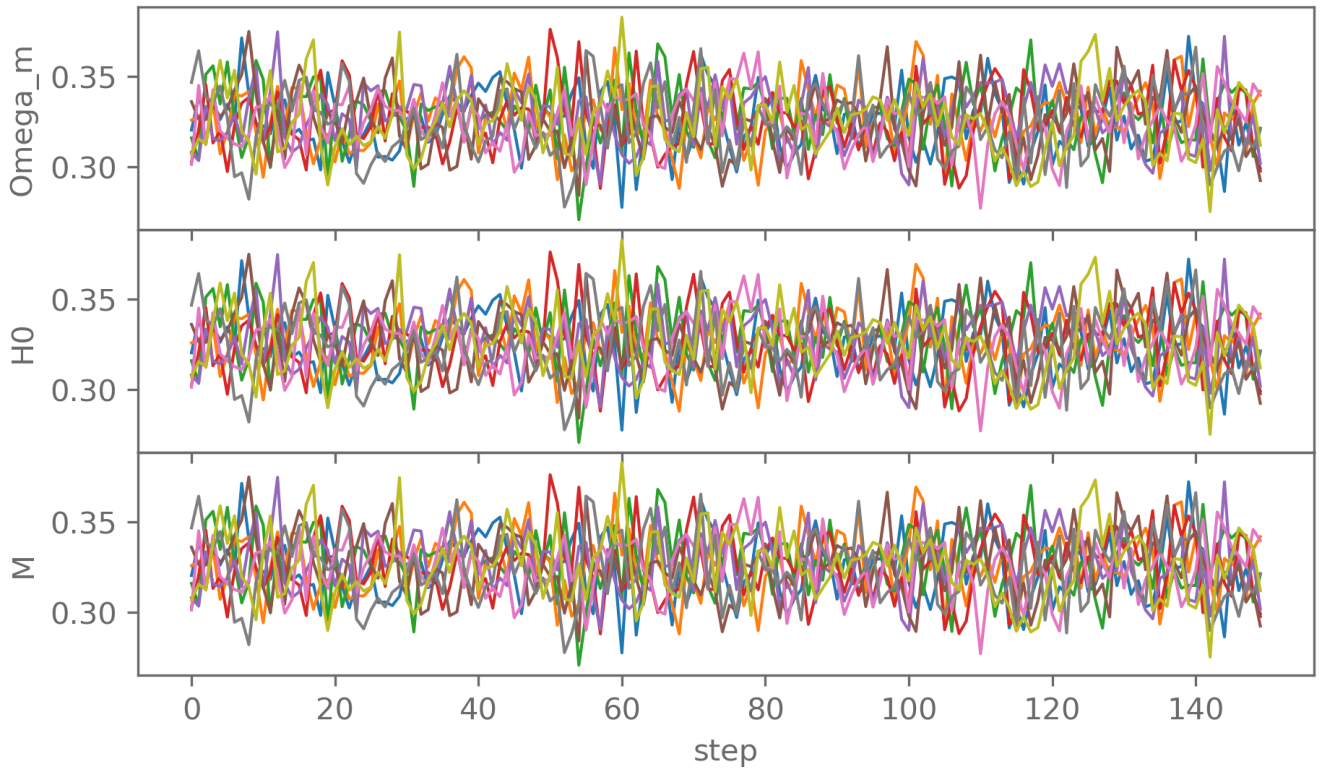
```
# Check the autocorrelation times
print("Integrated auto-correlation time")
for name, iat in zip(param_names, sampler.get_autocorr_time()):
    print(f"{name}: {iat:.1f}")
```

```
Integrated auto-correlation time
Omega_m: 39.0
H0: 56.8
M: 30.9
```

Remove a few auto-correlation times for the burn-in and thin out the chain.

```
burn_in = 500
thin = 30

chain_per_walker = sampler.get_chain(discard=burn_in, thin=thin)
chain = sampler.get_chain(discard=burn_in, thin=thin, flat=True)
```

Make a trace plot to check for weird chain behaviour.



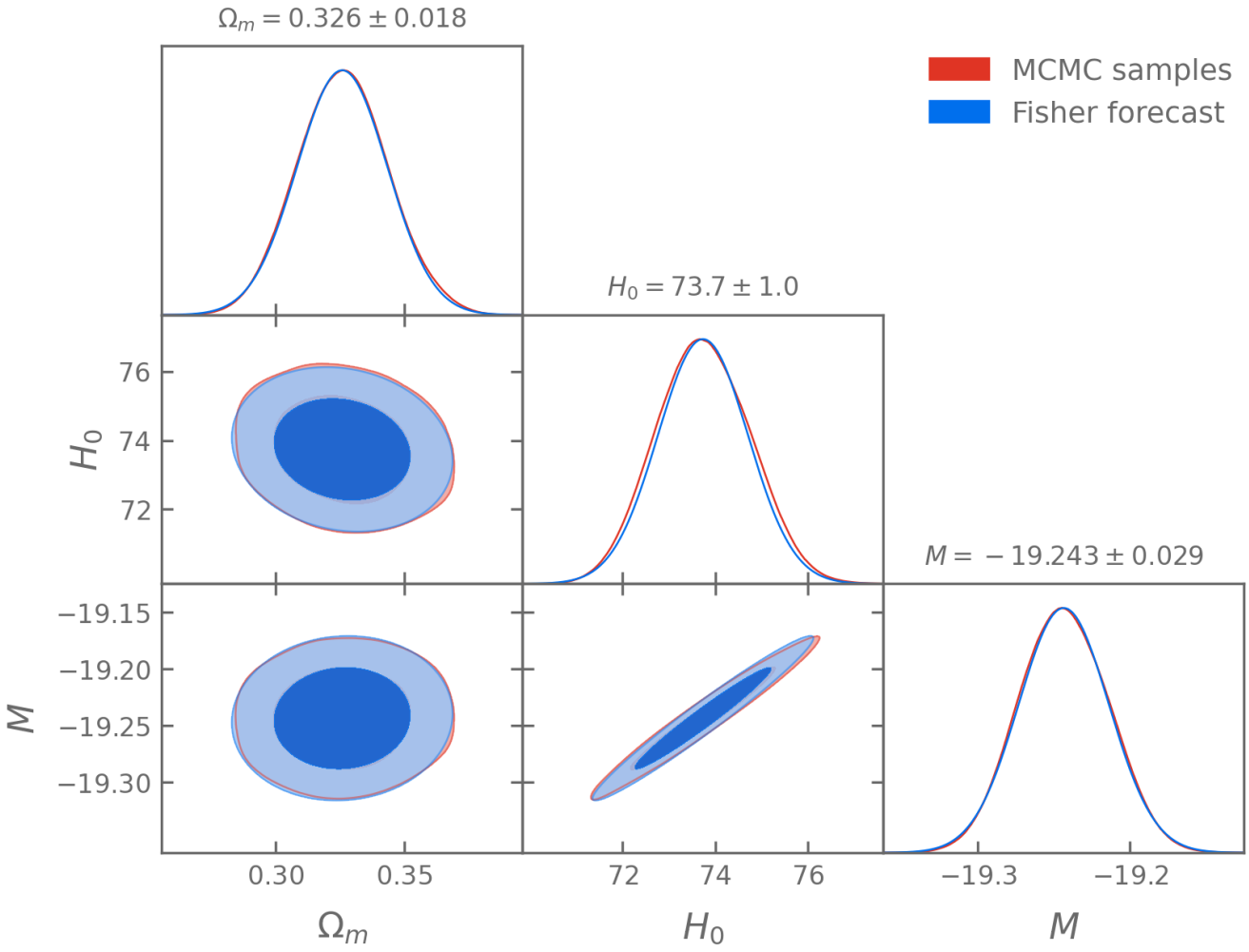
The chain looks good, so now to plot the chain and get summary statistics on the posterior.

We use `getdist` here, since it has a bit more features than `corner` and makes prettier plots.

```
getdist_samples = getdist.MCSamples(
    samples=chain,
    names=param_names,
    labels=param_labels,
    # We tell getdist what the ranges of the parameters are to avoid plotting
    # artefacts at the boundaries
    ranges={"Omega_m": (0.1, 0.9),
            "H0": (55, 91),
            "M": (-20, -18)},
    label="MCMC samples"
)
```

```
Removed no burn in
```

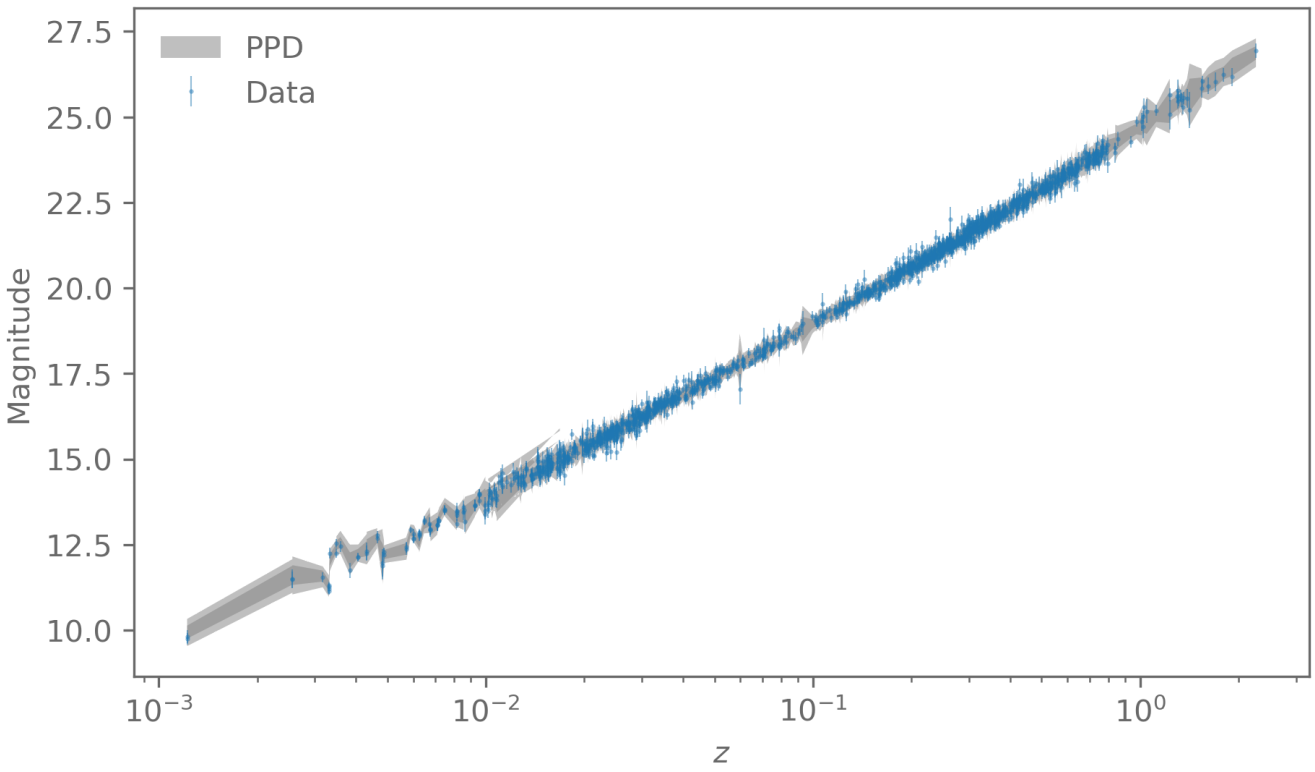
```
<Figure size 1800x1350 with 0 Axes>
```



We need to check that our model fits the data.

We first create samples from the posterior predictive distribution.

```
# Use 500 samples from the chain
posterior_predictive_samples = chain[np.random.choice(chain.shape[0], 500)]

# Define function that samples the likelihood given a set of parameter
def sample_ppd(params, seed):
    return create_likelihood_distribution(
        dict(Omega_m=params[0], H0=params[1], M=params[2])
    ).sample(seed=seed)

keys = random.split(random.PRNGKey(42), posterior_predictive_samples.shape[0])

# Use JAX vmap to get PPD samples for all the parameter samples
ppd = jit(vmap(sample_ppd))(posterior_predictive_samples, keys)
```



Now check the goodness of fit by comparing a test statistic over the PPD to that to the observed data.

```
# We use a statistic that basically the chi-square statistic
def test_statistic(data, params):
    chi2 = -2*create_likelihood_distribution(
        dict(Omega_m=params[0], H0=params[1], M=params[2])
    ).log_prob(data)
    return chi2

# vmap makes it easy to get the our samples of the test statistic
t_rep = jit(vmap(test_statistic))(ppd, posterior_predictive_samples)
t_data = jit(vmap(test_statistic, in_axes=[None, 0]))(
    pantheon_sh0es.data, posterior_predictive_samples
)
```

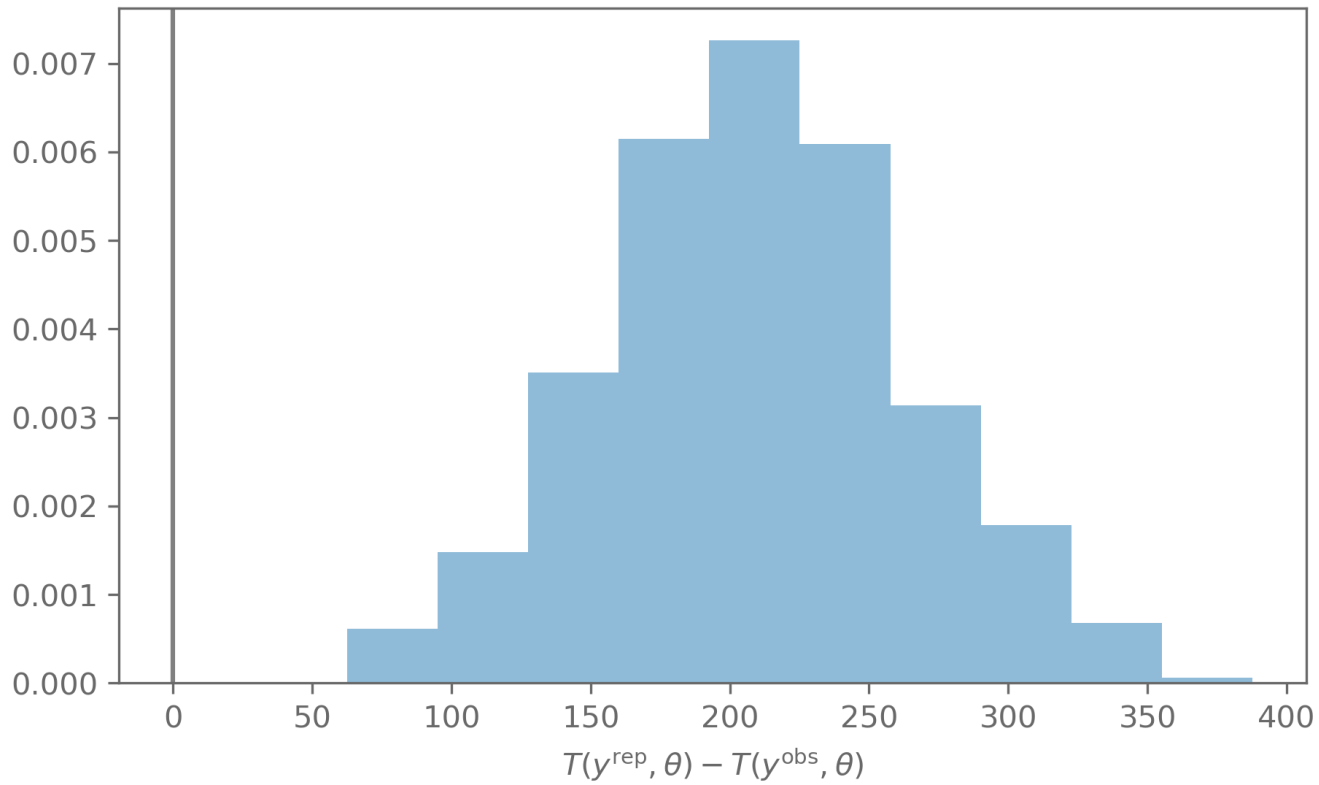
Find the fraction of where the test statistic on the PPD is larger than on the observed data.

```
print(f"PTE PPD = {(t_rep > t_data).sum()/t_rep.shape[0]:.3f}")
```

```
PTE PPD = 1.000
```

That does not look too good. The observed data is too likely in a sense.

Let us check the histogram of $T(y^{\text{rep}}, \theta) - T(y^{\text{obs}}, \theta)$.



This is suspicious. Let us check the χ^2 statistic to see if that gives us a similar result.

```
import scipy.stats

mu = pantheon_sh0es.model(MAP_params)
r = pantheon_sh0es.data - mu

chi2 = r @ pantheon_sh0es.inverse_covariance @ r

print(f"PTE chi-square: {scipy.stats.chi2(df=pantheon_sh0es.data.shape[0]-3).sf(chi2)
```

```
PTE chi-square: 0.9999
```

Something is not right. The first suspect is the covariance.

We used the STAT+SYS covariance here. That means the covariance includes the systematic uncertainty on top of the statistical uncertainty. This could explain the very high PTE on the test statistic.

Doing the analysis with the STATONLY covariance lowers the PTE a little bit but it is still much too high.

That means we need to go back and have a detailed look at the data and how the covariance is estimated.

For that we might want to focus on different parts of the data. For example, the current data vector has parts where the distances are assumed to be known (the Cepheid calibrators), while for the rest the distances are computed based on the parameters and redshifts.

The data also come from different surveys. We could look at if any of the surveys has data that looks suspicious and does not fit into the model.

Once we have fixed the covariance and get acceptable fits, we can then move on the model comparison.

For example, here we assumed the Universe is spatially flat and that dark energy is a cosmological constant.

We can analyse the same data but with different models to assess whether a Universe with curvature or a form of dark energy with a different equation of state is preferred by the data.

For that we would use nested sampling to get the evidences, and cross-check with the DIC and WAIC information criteria.