

Working with posteriors and model checking

Checking MCMC outputs

We spent the last two lectures on sampling algorithms. But how do we check that their outputs actually sample from our target distribution?

- Trace plots
- Auto-correlation time/number of effective samples
- If applicable: Gelman-Rubin \hat{R} statistic

Trace plots

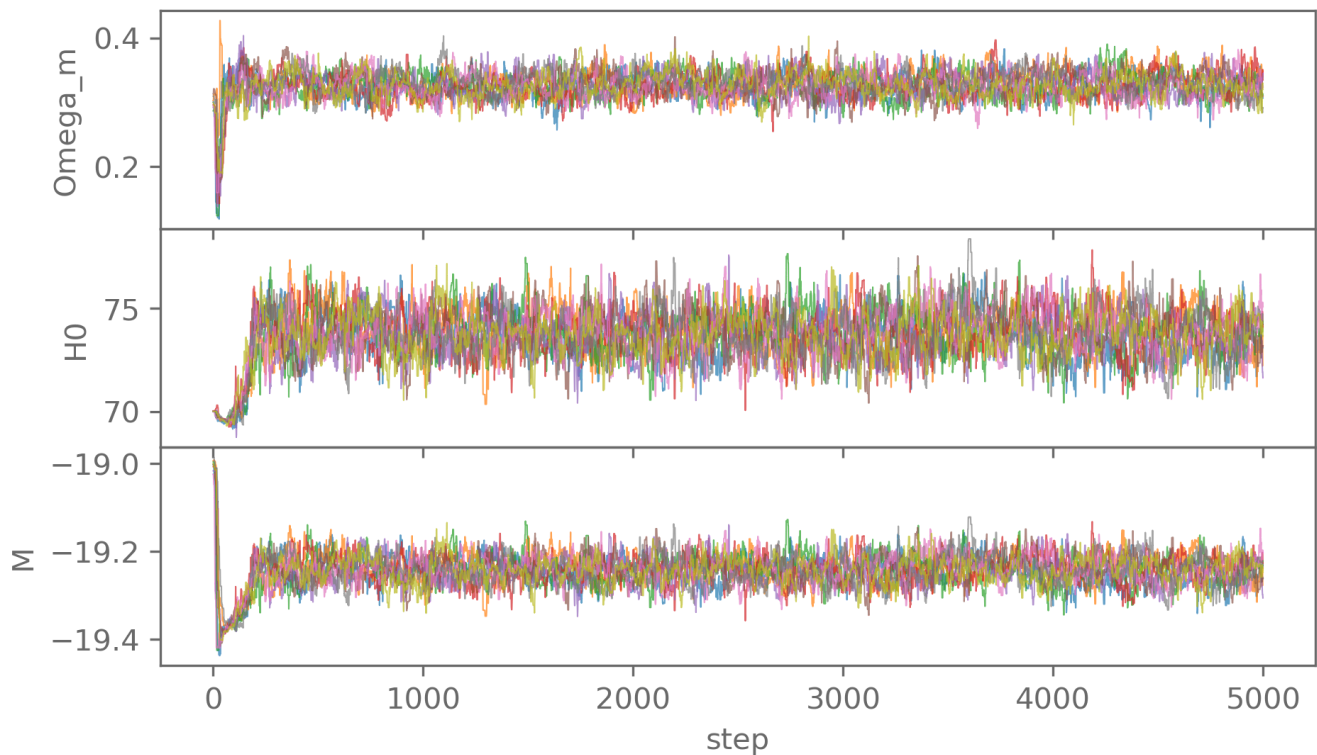
Plot the value of each parameter in the chain(s) vs iteration.

A simple visualisation that can flag many common issues, like burn-in, poor mixing, non-stationarity, etc.

Imagine we ran an MCMC with `emcee` (the super nova example in this case):

```
sampler = emcee.EnsembleSampler(  
    nwalkers=n_walker, ndim=n_param,  
    ...  
)  
state = sampler.run_mcmc(...)  
chain_per_walker = state.get_chain()
```

The trace plot of this chain looks like this:



The first couple hundred steps are clearly affected by burn-in.

Otherwise the chain looks good:

- The mean and variance do not vary with time, indicating that the distribution is stationary.
- The different walkers all explore the distribution and do not get stuck in a subspace.

Auto-correlation/effective number of samples

A more quantitative check is the (integrated) auto-correlation time τ , which tells us how long the chain takes to "forget" a previous position.

The `emcee` documentation has an excellent write-up on this:

<https://emcee.readthedocs.io/en/stable/tutorials/autocorr/>

In short, if the chain has n samples in it, the number of effective independent samples is $n_{\text{eff}} \approx \frac{n}{\tau}$.

One thing to keep in mind is that for $n \lesssim 50\tau$, the estimate for τ will be unreliable.

```
# If the chain is an array (for example from a different MCMC algorithm)
tau = emcee.autocorr.integrated_time(chain_per_walker)

# If the algorithm does use walkers, set the has_walkers=False argument
# tau = emcee.autocorr.integrated_time(chain_per_walker[:, 0], has_walkers=False)

# Or directly from emcee:
# tau = sampler.get_autocorr_time()
```

```
print("Integrated auto-correlation time")
for name, iat in zip(param_names, tau):
    print(f"{name}: {iat:.1f}")
```

```
Integrated auto-correlation time
Omega_m: 46.0
H0: 79.8
M: 48.8
```

This indicates that the chain needs 50-80 steps to produce a new independent sample.

To get rid of the burn-in phase, it is good to discard a few (5-10) auto-correlation times.

This agrees with what we could estimate from the trace plots earlier.

Since many of the samples in the chain are correlated, we can thin down the chain by a factor of $\approx \tau$.

```
max_autocorr = max(tau)
burn_in = int(5*max_autocorr)
thin = int(max_autocorr/2)

chain = chain_per_walker[burn_in::thin].reshape(-1, n_param)
# Directly from an emcee sampler:
# chain = sampler.get_chain(discard=burn_in, thin=thin, flat=True)
```

Interpreting posteriors

We have evaluated and sampled a few posteriors so far. As Bayesians, that is the end-goal, since the full posterior distribution is what we care about.

In practice, we need to summarise the posterior in a form that allows it to be represented on a sheet of paper.

- Mode of the posterior
 - pro: summary of the full n -dimensional joint posterior
 - con: point estimate, uncertainty is hard to define in practice
- Credible intervals
 - pro: intuitive interpretation, can be put in a table
 - con: only applies to 1D (marginal) distributions, can suffer from projection effects
- Corner plots
 - pro: more information than credible intervals due to showing 2D marginals as well
 - con: still only marginals, can suffer from projection effects

Credible interval or region

A credible region for a probability distribution $p(x)$ is a set $\Omega(\alpha)$ such that the probability enclosed within that region is $1 - \alpha$:

$$\int_{\Omega(\alpha)} p(x) dx = 1 - \alpha.$$

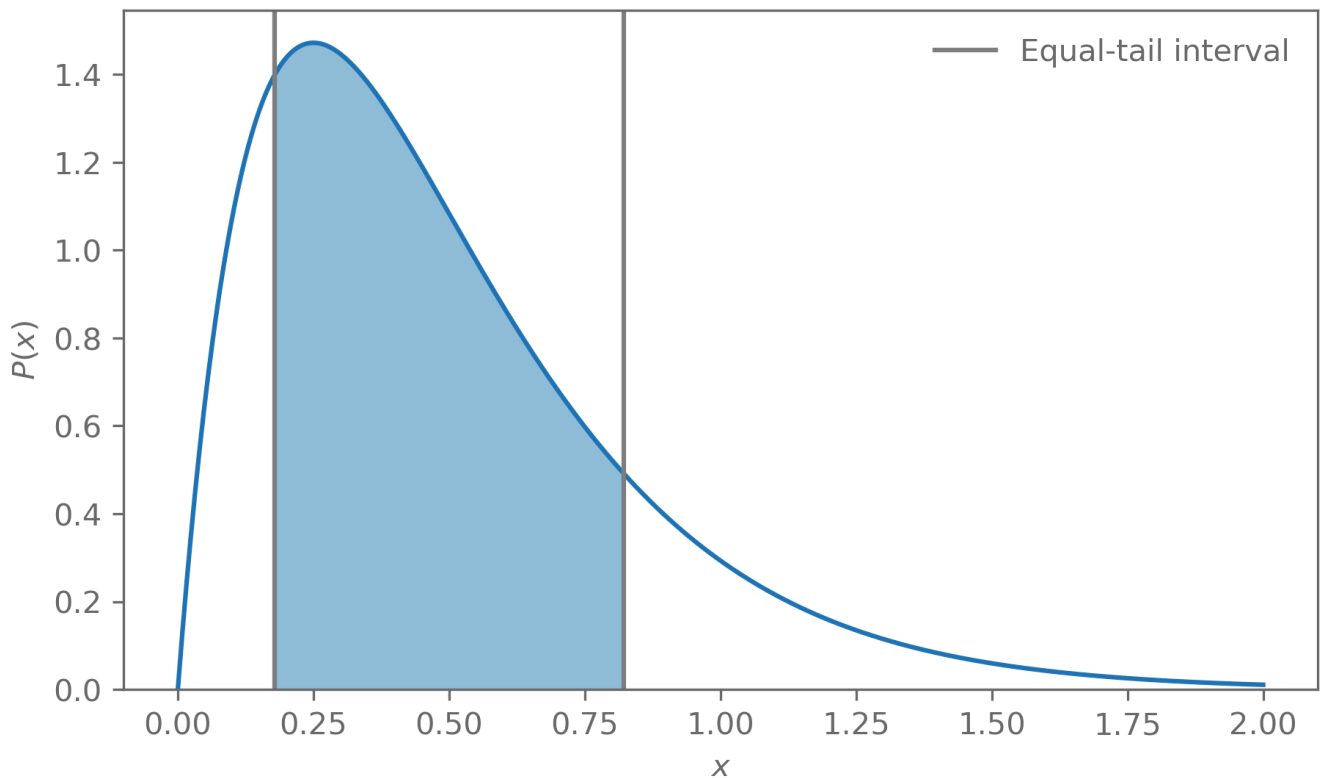
There are infinitely many ways to define such regions but two common ones are

- equal-tail intervals
- highest posterior density intervals

The equal-tail interval (x_L, x_R) is defined by requiring the tails of the distribution to each contain $\frac{\alpha}{2}$ of the total probability.

$$\text{CDF}(x_L) = \frac{\alpha}{2}, \quad 1 - \text{CDF}(x_R) = 1 - \frac{\alpha}{2}$$

The equal-tail interval is centred on the median of $p(x)$. The equal-tail interval has the advantage that it is easy to compute, for example from samples from a posterior.



The highest-density interval (or region) is defined as the region for which $p(x)$ is above a threshold c_α , such that

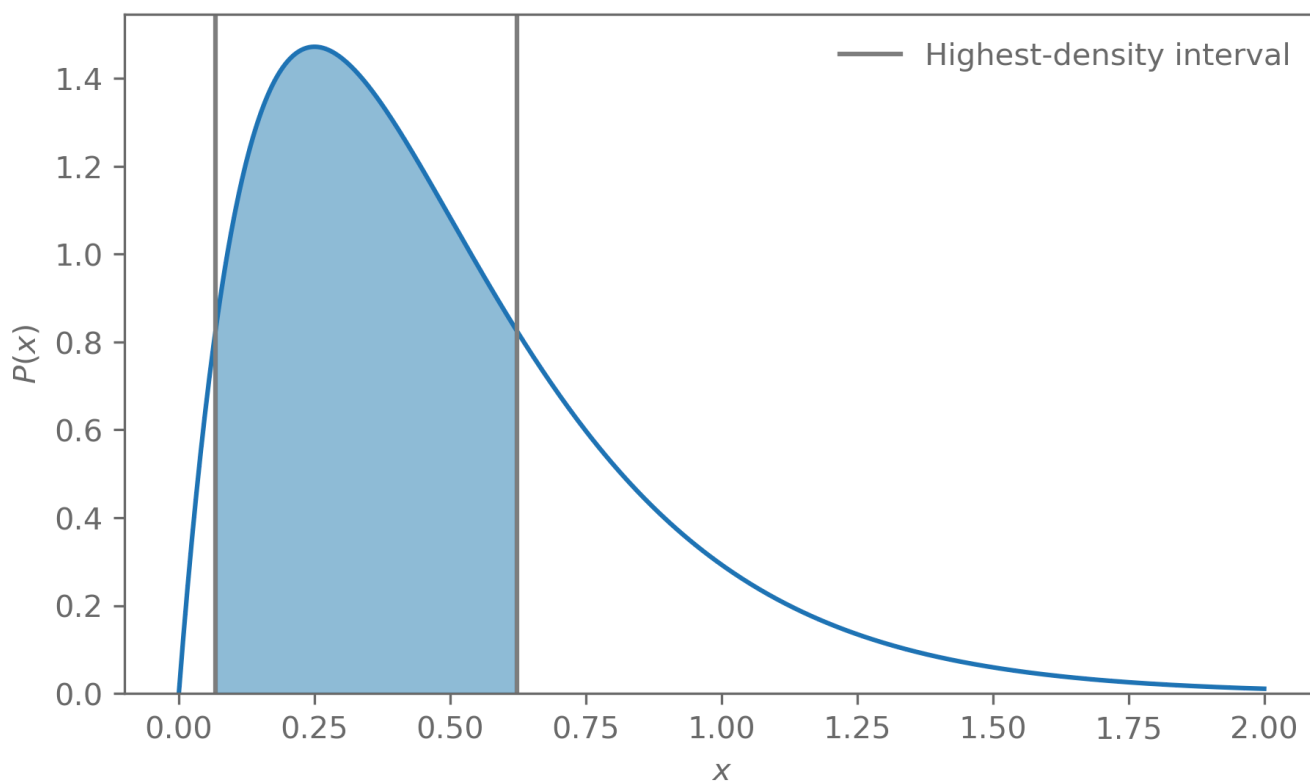
$$\int_{\Omega(\alpha)} p(x) dx = 1 - \alpha,$$

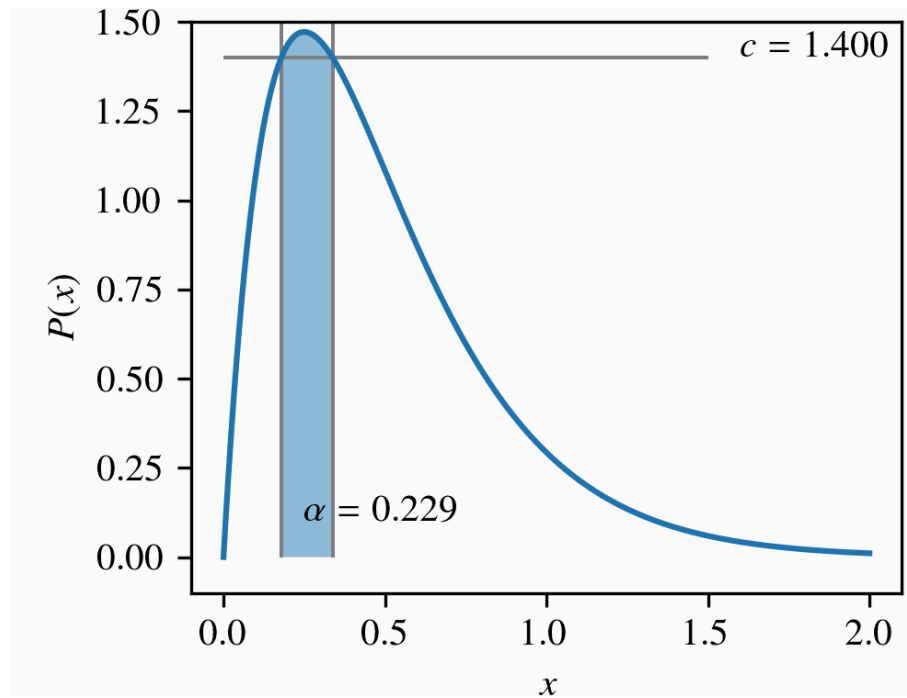
where

$$\Omega(\alpha) = \{x | p(x) > c_\alpha\}.$$

The highest-density regions are the smallest possible credible region that can be defined. It is centred on the mode of $p(x)$.

The downside is that it is more complicated to compute (even in 1D).

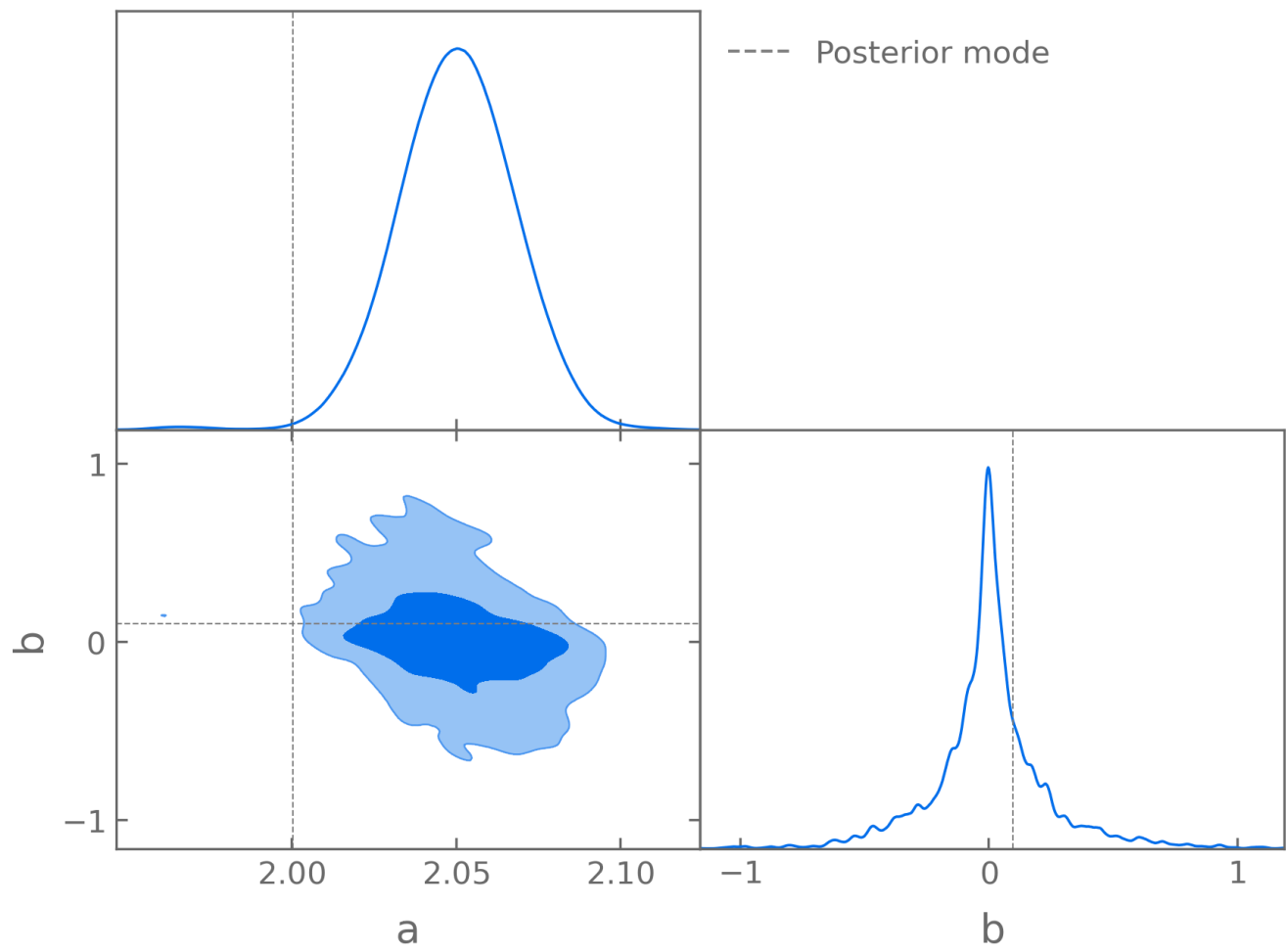


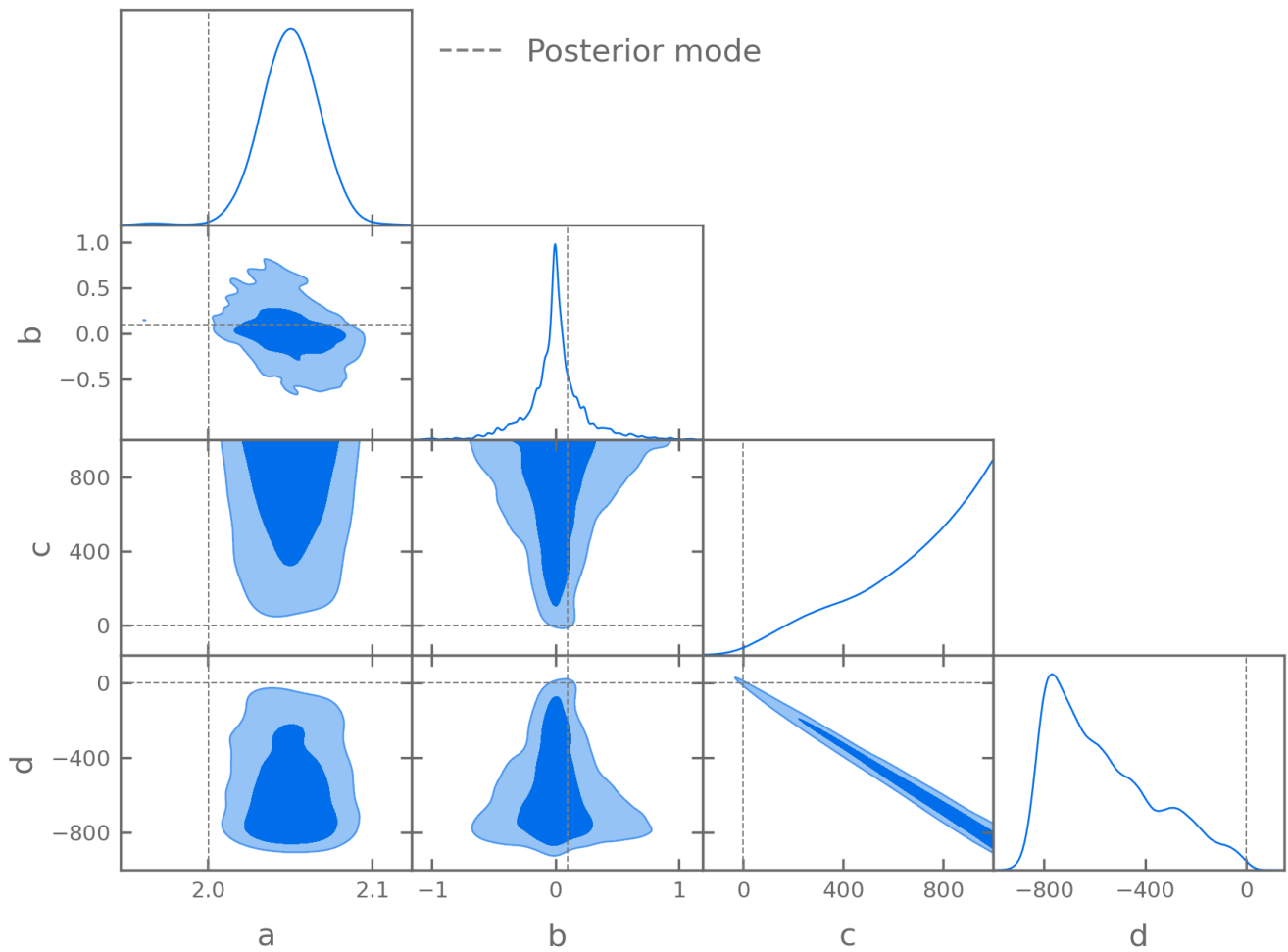


Projection effects

Once we have more than 2 dimensions in our posterior, we can only visualise marginal distributions of it (unless you use a fancy 3D plotting library).

These marginal distributions can behave in unintuitive ways when the posterior is not symmetric and Gaussian.





The model here is a Gaussian likelihood with mean

$$\mu = a + bx \left(\frac{1+x}{1+x_0} \right)^c \left(\frac{1+x^2}{1+x_0^2} \right)^d,$$

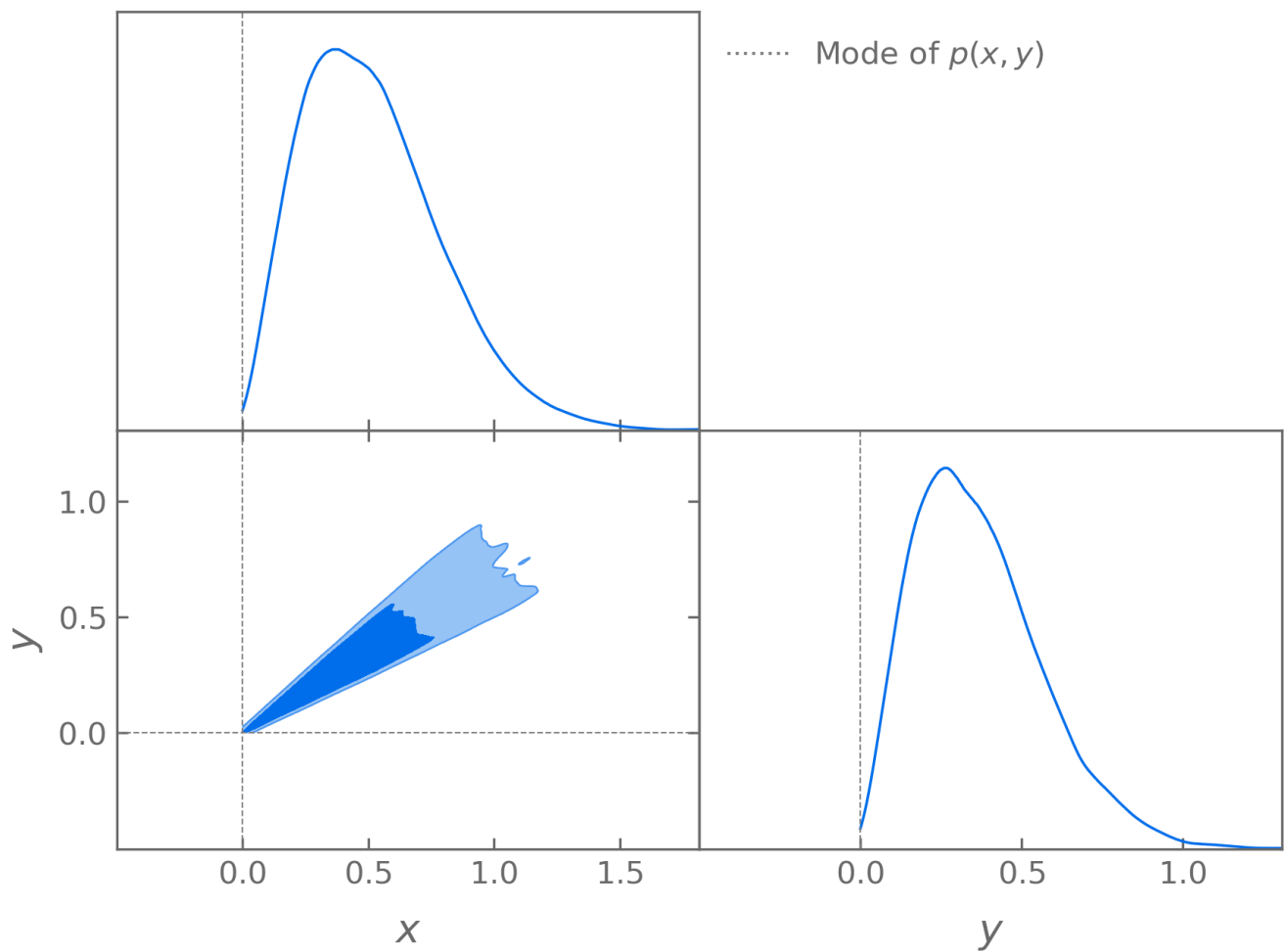
for parameters a, b, c, d , with $x_0 = 0.5$.

When $b \approx 0$, the terms involving c and d do not contribute to the model anymore and c and d are not constrained.

Depending on how the prior is chosen, this can lead to large volumes of posterior space where $b \approx 0$.

When projecting the posterior into marginal distributions, this leads to marginals that are not centred on the mode of the joint posterior.

This can be seen with very skewed distributions even in 2 dimensions:



Clicker

What causes volume or projection effects?

- The projection of high dimensions onto marginal distributions.
- Unconstrained regions in the parameter space due to degeneracy between parameters.
- The influence of the prior distribution on the effective number of parameters.
- Data rescaling on the posterior distribution.

The Bayesian data analysis workflow

1. Model the data generating process probabilistically.
2. Condition the generative process on the observed data.
3. Check that the results fit the observed data and improve the model.
 - Compare different models.

Model checking

A first step to check your model is to plot it against the data.

This can be plotting the best-fit model, but better are the posterior predictive distributions.

If there are obvious issues, such as outliers or misspecified errors, making some plots can already flag these problems.

Our eyes can be easily deceived though. For example, strongly correlated data can have very unintuitive behaviour of the best-fitting model.

Chi-square goodness-of-fit

If the data \vec{y} are Gaussian distributed

$$\vec{y} \sim \mathcal{N}(\vec{\mu}(\vec{\theta}), \Sigma(\vec{\theta}))$$

then a traditional test of how well our model fits the data is the chi-square test.

Let us assume we have a set $\vec{\theta}^*$ of "best" model parameters. For example the MAP or mean of the posterior.

If these are the true parameters that generated the data and the data are Gaussian distributed, then the quantity

$$\chi^2(\vec{y}, \vec{\theta}^*) = (\vec{y} - \vec{\mu}(\vec{\theta}^*))^T \Sigma(\vec{\theta}^*)^{-1} (\vec{y} - \vec{\mu}(\vec{\theta}^*))$$

is distributed as χ^2_ν .

With this we can calculate the probability that the χ^2 statistic would take a higher value in imaginary repeated experiments than the one we have observed

$$\text{Probability to exceed (PTE)} \chi^2(\vec{y}, \vec{\theta}^*) = 1 - \text{CDF}(\chi^2(\vec{y}, \vec{\theta}^*)).$$

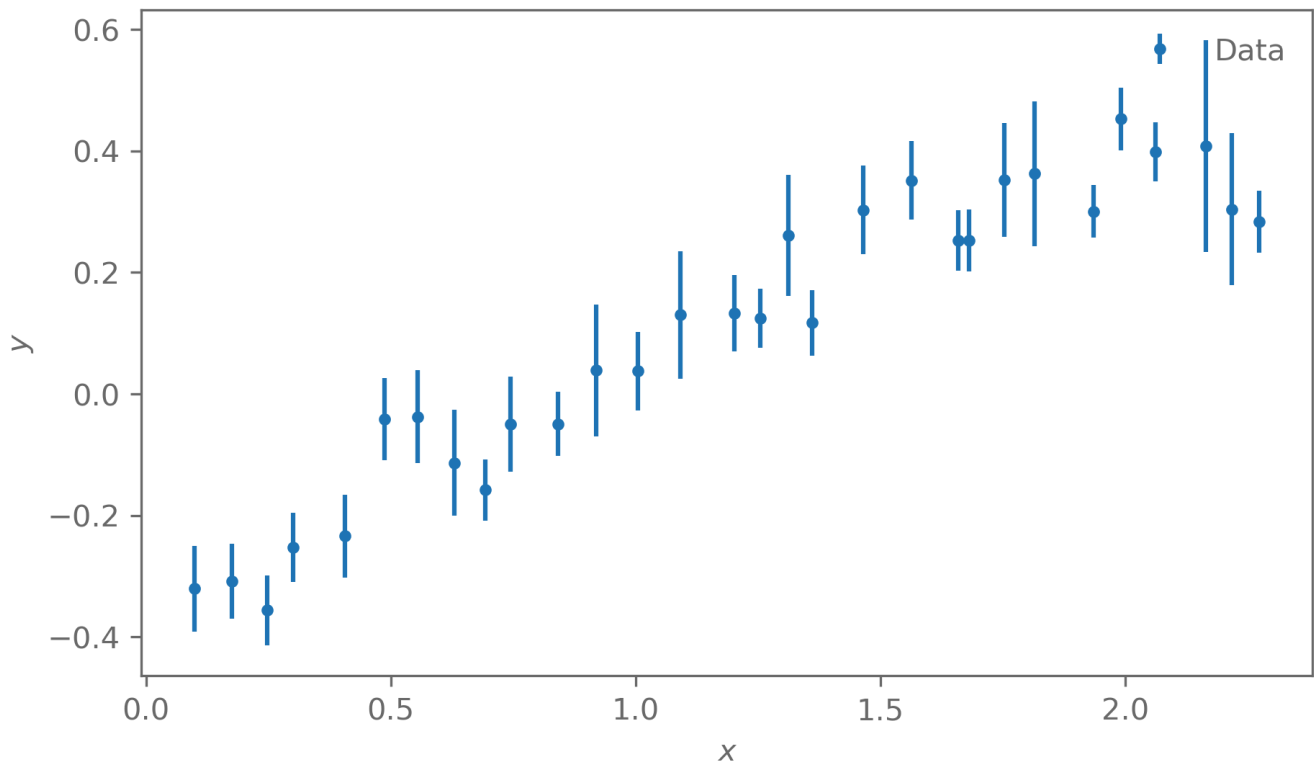
If the PTE is small, it is unlikely that an experiment would see a χ^2 statistic larger than what we observed. This indicates that our model is a poor fit to the data.

The number of degrees of freedom is $\nu = n_{\text{data}} - n_{\text{param}}$, where n_{data} is the dimensionality of the data and n_{param} is the number of constrained parameters.

By fitting the model to the data, we reduce the scatter in the residuals $\vec{y} - \vec{\mu}(\theta)$, so the number of degrees of freedom is reduced from n_{data} by n_{param} .

Finding n_{param} in a Bayesian context can be challenging because some parameters might be constrained by the prior, instead of by the likelihood. In this case, they do not reduce the degrees of freedom.

We use the data in "data/linear_fits/data_2.txt" has an example and fit both a linear and quadratic model to it.



```
def linear_model(theta, x):
    m, b = theta
    return m*x + b

def log_prior_linear_model(theta):
    m, b = theta
    # Unnormalised uniform prior  $m \sim U(-2, 2)$ ,  $b \sim U(-3, 3)$ 
    if -2 < m < 2 and -3 < b < 3:
        return 0
    else:
        return -np.inf

log_likelihood_lin, log_posterior_lin, predict_lin = \
    make_gaussian_likelihood_components(
        linear_model, log_prior_linear_model
    )
```

```
def quadratic_model(theta, x):
    a, b, c = theta
    return a*x**2 + b*x + c

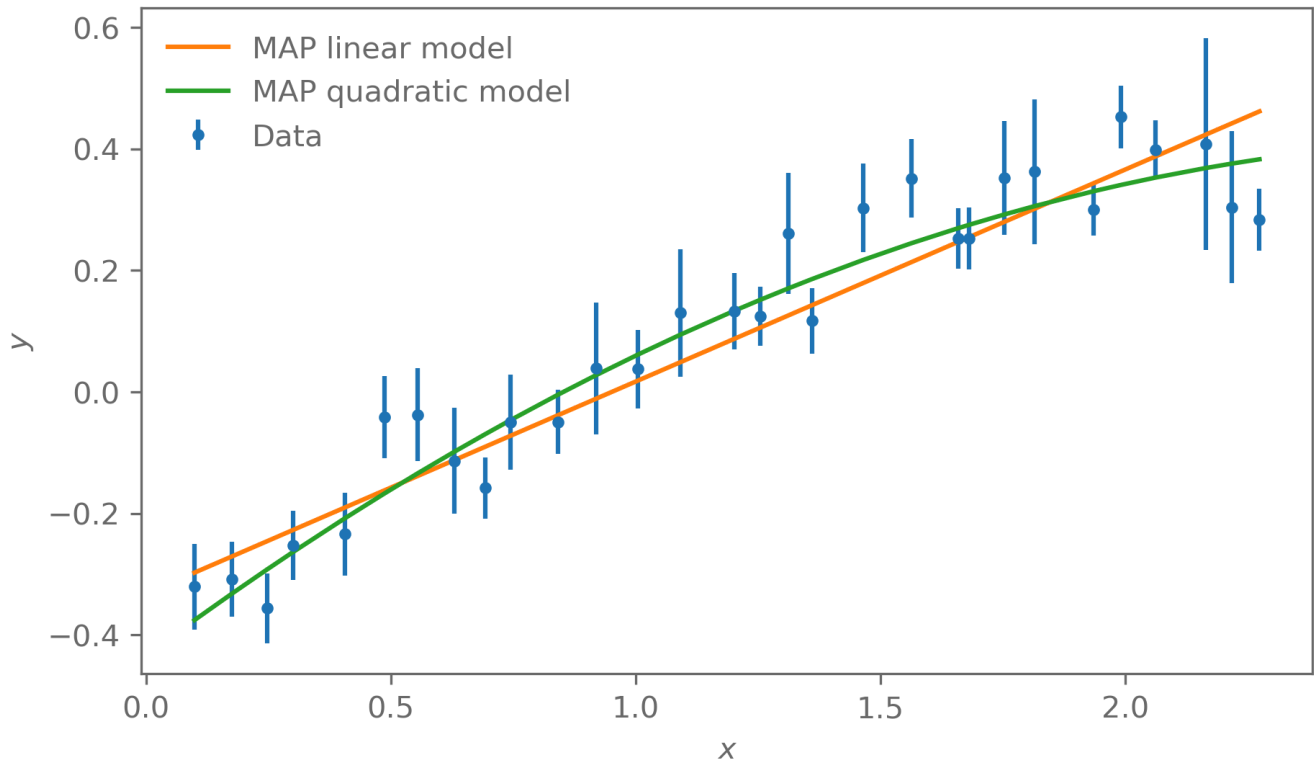
def log_prior_quadratic_model(theta):
    a, b, c = theta
    # Unnormalised uniform prior  $m \sim U(-2, 2)$ ,  $b \sim U(-3, 3)$ 
    if -1 < a < 1 and -2 < b < 2 and -3 < c < 3:
        return 0
    else:
```

```

    return -np.inf

log_likelihood_quad, log_posterior_quad, predict_quad = \
    make_gaussian_likelihood_components(
        quadratic_model, log_prior_quadratic_model
    )

```



Implement the χ^2 statistic:

```

def chi_squared(y, sigma_y, mu):
    return np.sum((y - mu)**2/sigma_y**2)

chi_squared_linear = chi_squared(
    y, y_err,
    mu=linear_model(results_linear["MAP"], x)
)
chi_squared_quadratic = chi_squared(
    y, y_err,
    mu=quadratic_model(results_quadratic["MAP"], x)
)

```

```

n_data = len(y)
n_param_lin = 2
n_param_quad = 3

PTE_lin = scipy.stats.chi2(df=n_data - n_param_lin).sf(chi_squared_linear)
PTE_quad = scipy.stats.chi2(df=n_data - n_param_quad).sf(chi_squared_quadratic)

print(f"Linear:  $\chi^2$ ={chi_squared_linear:.1f}, "
      f"ndof={n_data}-{n_param_lin}, PTE={PTE_lin:.3f}")

```

```
print(f"Quadratic:  $\chi^2$ ={{chi_squared_quadratic:.1f}}, "  
      f"ndof={{n_data}}-{{n_param_quad}}, PTE={{PTE_quad:.3f}}")
```

Linear: $\chi^2=40.8$, ndof=30-2, PTE=0.056
Quadratic: $\chi^2=29.3$, ndof=30-3, PTE=0.347

The quadratic model fits much better to the data. Note that for the linear model, the probability of getting data that has an even higher χ^2 than we observed is quite low, indicating that the linear model is a poor fit.

Posterior predictive checks

For more details, see chapter 6 in Bayesian Data Analysis

The chi-squared test has some downsides:

- It assumes the data are Gaussian distributed.
- It depends on the effective number of constrained parameters, which can be challenging to define.
- It uses a point estimate for the parameters $\vec{\theta}^*$ for which to calculate the goodness-of-fit. As Bayesians, we want to avoid point estimates and instead account for the full uncertainty in the posterior.

We use the posterior predictive distribution (PPD) to assess how well our model fits the data.

We define a test statistic $T(y, \theta)$ of the data and the parameters. For example, we could choose the χ^2 statistic

$$T(\vec{y}, \vec{\theta}) = \left(\vec{y} - \vec{\mu}(\vec{\theta}) \right)^T \Sigma(\vec{\theta})^{-1} \left(\vec{y} - \vec{\mu}(\vec{\theta}) \right) .$$

The χ^2 statistic is often convenient but other choice for $T(y, \theta)$ are possible too.

We then sample replicates \vec{y}^{rep} of the data from the PPD $p(\vec{y}^{\text{rep}}|\vec{y})$. Remember,

$$p(\vec{y}^{\text{rep}}|\vec{y}) = \int p(\vec{y}^{\text{rep}}, \vec{\theta}|\vec{y}) d\theta = \int p(\vec{y}^{\text{rep}}|\vec{\theta}) p(\vec{\theta}|\vec{y}) d\theta .$$

Then compute the probability

$$\Pr \left(T(\vec{y}^{\text{rep}}, \vec{\theta}) \geq T(\vec{y}, \vec{\theta}) | \vec{y} \right) .$$

In practice:

1. Sample S samples $\vec{\theta}_i$ from the posterior $p(\vec{\theta}|\vec{y})$
2. Sample one \vec{y}_i^{rep} from the likelihood $p(\vec{y}^{\text{rep}}|\vec{\theta}_i)$ for each θ_i
3. Count the fraction of samples where $T(\vec{y}_i^{\text{rep}}, \vec{\theta}_i) \geq T(\vec{y}, \vec{\theta}_i)$

```

def test_statistic(y, theta, x, sigma_y, model):
    mu = model(theta, x)
    t = chi_squared(y, sigma_y, mu)
    return t

def ppd_model_check(test_statistic, y, ppd, ppd_params):
    t_data = []
    t_rep = []
    for y_rep, theta in zip(ppd, ppd_params):
        t_data.append(test_statistic(y, theta))
        t_rep.append(test_statistic(y_rep, theta))

    t_data = np.array(t_data)
    t_rep = np.array(t_rep)

    pte = (t_rep >= t_data).sum()/len(t_data)
    return pte, t_rep, t_data

```

```

PPD_PTE_lin, t_rep_lin, t_data_lin = ppd_model_check(
    test_statistic=lambda y, theta: test_statistic(
        y, theta, x, y_err, linear_model),
    y=y,
    ppd=results_linear["PPD"],
    ppd_params=results_linear["PPD_params"]
)

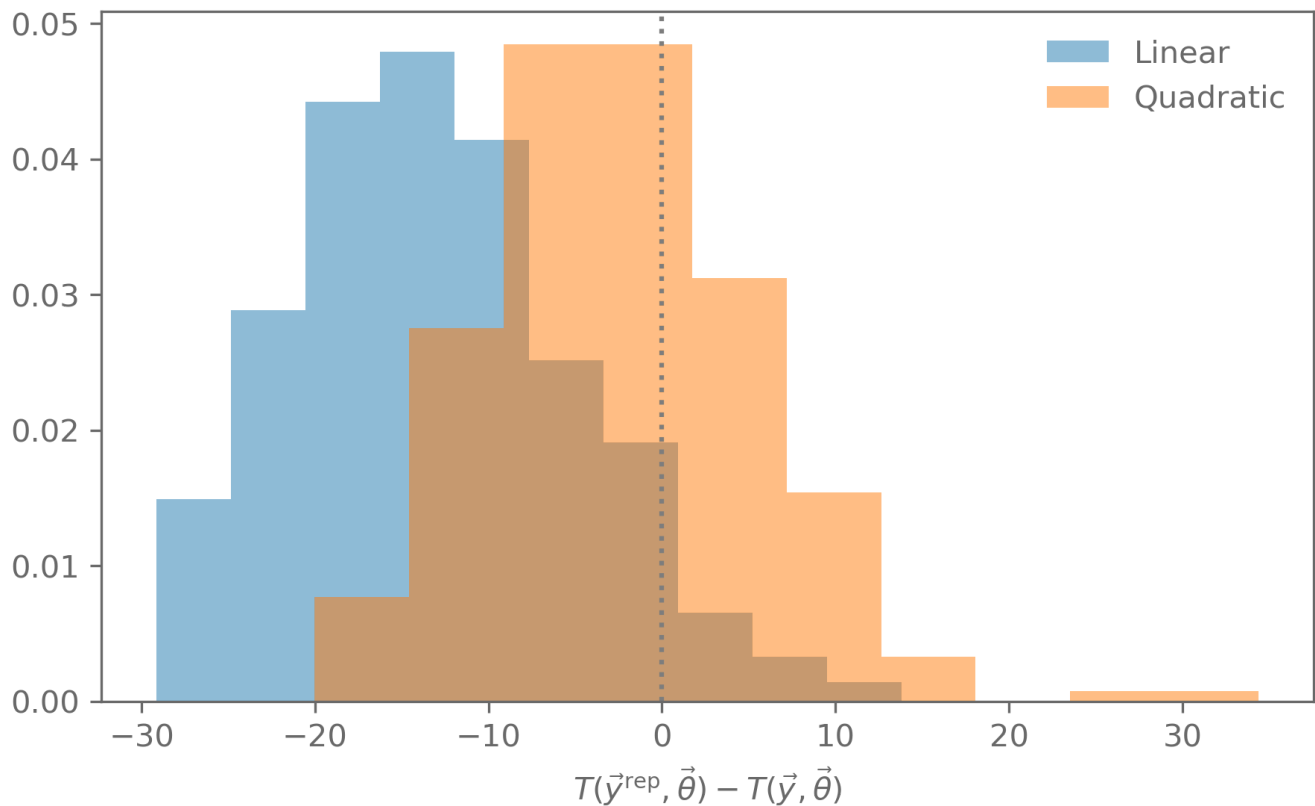
PPD_PTE_quad, t_rep_quad, t_data_quad = ppd_model_check(
    test_statistic=lambda y, theta: test_statistic(
        y, theta, x, y_err, quadratic_model),
    y=y,
    ppd=results_quadratic["PPD"],
    ppd_params=results_quadratic["PPD_params"]
)

print(f"Linear: PPD PTE={PPD_PTE_lin:.3f}")
print(f"Quadratic: PPD PTE={PPD_PTE_quad:.3f}")

```

Linear: PPD PTE=0.064
 Quadratic: PPD PTE=0.372

We get quite similar results to the χ^2 test just from samples from the posterior, without further assumptions or point estimates.



Clicker

What is the difference between a chi-square test and a posterior predictive check with a chi-square test statistic?

- A chi-square test measures predictive accuracy, while a posterior predictive check measures overfitting.
- In the chi-square test, the chi-square statistic always decreases with more parameters, but in a posterior predictive check, it increases as parameters are added.
- The chi-square test assumes Gaussianity whereas the PPC with a chi-square test statistic doesn't.
- The chi-square test uses a point estimate for model parameters, while the posterior predictive check accounts for the full uncertainty in the posterior.

Model comparison

After we have checked that the model is describing the data well, we might want to compare different models.

We have already discussed the Bayes ratio $\frac{p(y|M_1)}{p(y|M_2)}$ as a tool to decide between models.

Here we look at two other approaches to compare models that do not need computing the evidence.

Deviance information criterion (DIC)

The deviance information criterion (DIC) is defined as

$$\text{DIC} = -2 \left(\log p(\vec{y}|\vec{\theta}^*) - p_D \right) ,$$

with the number of effective model parameters

$$p_D = 2 \left(\log p(\vec{y}|\vec{\theta}^*) - E_{p(\theta|\vec{y})}[\log p(\vec{y}|\vec{\theta})] \right) ,$$

where the last term is the mean of the log likelihood over the posterior.

An alternative estimate of the number of effective model parameters is

$$p_V = 2 \text{Var}_{p(\theta|\vec{y})}[\log p(\vec{y}|\vec{\theta})] .$$

$$\text{DIC} = -2 \left(\log p(\vec{y}|\vec{\theta}^*) - p_D \right) ,$$

Interpretation: log-likelihood at the point of the best-fit parameter $\vec{\theta}^*$ minus the number of parameters in the model.

The better the fit, the higher the likelihood. The DIC is lower for better models, but adding more parameters that do not significantly improve the likelihood are penalised.

Why are p_D and p_V estimates of the effective number of model parameters?

Let us consider the case where the likelihood, posterior, and prior are approximately Gaussian, with d model parameters.

Define $D(\theta) = -2 \log p(\vec{y}|\vec{\theta})$. First, expand around the best-fit $\vec{\theta}^*$:

$$D(\theta) \approx D(\theta^*) + (\theta - \theta^*) D'_{\theta^*} + \frac{1}{2} (\theta - \theta^*)^T D''_{\theta^*} (\theta - \theta^*) ,$$

where D'_{θ^*} and D''_{θ^*} are the first and second derivatives of D with respect to $\vec{\theta}$ at $\vec{\theta}^*$.

Then take the expectation with respect to the posterior:

$$\begin{aligned} E[D(\theta)] &\approx D(\theta^*) + E[(\theta - \theta^*)^T D''_{\theta^*} (\theta - \theta^*)] \\ &= D(\theta^*) + \text{tr}(D''_{\theta^*} E[(\theta - \theta^*)(\theta - \theta^*)^T]) \\ &= D(\theta^*) + \text{tr}(D''_{\theta^*} \Sigma_{p(\theta|y)}) , \end{aligned}$$

where $\Sigma_{p(\theta|y)}$ is the covariance of the posterior.

Remember the definition of p_D :

$$p_D = D(\theta^*) - \mathbb{E}[D(\theta)] \approx \text{tr}(D''_{\theta^*} \Sigma_{p(\theta|y)})$$

We can write $D(\theta)$ in terms of the posterior and prior using Bayes' theorem:

$$D(\theta) = -2 \log p(\vec{y}|\vec{\theta}) = -2 \log \left(\frac{p(\theta|\vec{y})}{p(\vec{\theta})} p(\vec{y}) \right).$$

The Hessian of D''_{θ^*} can now be written as

$$D''_{\theta^*,ij} = -2 \frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\theta|\vec{y}) + 2 \frac{\partial^2}{\partial \theta_i \partial \theta_j} \log p(\theta) \approx \Sigma_{p(\theta|y)}^{-1} - \Sigma_{p(\theta)}^{-1},$$

where we assumed the posterior and prior are Gaussian, with covariances $\Sigma_{p(\theta|y)}$ and $\Sigma_{p(\theta)}$.

Putting things together, we find the effective number of model parameters

$$p_D \approx \text{tr}(D''_{\theta^*} \Sigma_{p(\theta|y)}) \approx d - \text{tr}(\Sigma_{p(\theta)}^{-1} \Sigma_{p(\theta|y)})$$

This behaves as we would hope:

- In the case where the prior is very uninformative ($\Sigma_{p(\theta)} \rightarrow \infty$) all parameters are constrained by the data: $p_D = d$.
- In the case where the prior is the same as the posterior ($\Sigma_{p(\theta)} = \Sigma_{p(\theta|y)}$), none of the parameters are constrained by the data: $p_D = 0$.

Implement the DIC:

```
def DIC(theta_star, theta_samples, log_likelihood):
    # Compute log likelihood at theta_star and the samples theta_i
    log_likelihood_star = log_likelihood(theta_star)
    log_likelihood_samples = np.array(
        [log_likelihood(theta) for theta in theta_samples]
    )
    p_D = 2*(log_likelihood_star - np.mean(log_likelihood_samples))
    p_V = 2*np.var(log_likelihood_samples)
    return -2*(log_likelihood_star - p_D), p_D, p_V
```

```
DIC_lin, p_D_lin, p_V_lin = DIC(
    theta_star=results_linear["MAP"],
    theta_samples=results_linear["PPD_params"],
    log_likelihood=lambda theta: log_likelihood_lin(y, theta, x, y_err)
)

DIC_quad, p_D_quad, p_V_quad = DIC(
    theta_star=results_quadratic["MAP"],
    theta_samples=results_quadratic["PPD_params"],
    log_likelihood=lambda theta: log_likelihood_quad(y, theta, x, y_err)
)
```

```
print(f"Linear: DIC = {DIC_lin:.1f}, p_D = {p_D_lin:.1f}, p_V = {p_V_lin:.1f}")
print(f"Quadratic: DIC = {DIC_quad:.1f}, p_D = {p_D_quad:.1f}, p_V = {p_V_quad:.1f}")
```

Linear: DIC = -60.2, p_D = 1.9, p_V = 2.0
 Quadratic: DIC = -69.4, p_D = 3.1, p_V = 2.9

The quadratic model has a lower DIC and is therefore preferred. The much better fit is not offset by the larger number of parameters.

Clicker

How does the Bayes factor conceptually differ from DIC when comparing models?

- The Bayes factor has a probabilistic interpretation whereas the DIC hasn't, it is just a quantitative criterion.
- The Bayes factor is more sensitive to the number of data points than DIC.
- The Bayes factor incorporates prior information, while DIC relies purely on the likelihood.
- DIC focuses on the posterior distribution, while the Bayes factor focuses on point estimates of parameters.

WAIC

The DIC is easy to calculate but has the downside that it relies on a point estimate for $\vec{\theta}^*$. A more Bayesian information criterion is the Watanabe-Akaike or widely applicable information criterion (WAIC).

It assumes we can partition the data into M partitions, for example the n entries y_i in the data vector \vec{y} . Then the log pointwise predictive density (lppd) is

$$\text{lppd} = \sum_{i=1}^n \log E[p(y_i|\vec{\theta})] = \sum_{i=1}^n \log \int p(y_i|\vec{\theta})p(\vec{\theta}|\vec{y})d\theta^n$$

The number of parameters are estimated as

$$p_{\text{WAIC}} = \sum_{i=1}^n \text{Var}[\log p(\vec{y}_i|\vec{\theta})]$$

Finally, the WAIC is calculated as

$$\text{WAIC} = -2 (\text{lppd} - p_{\text{WAIC}})$$

```
def WAIC(theta_samples, log_likelihood, y_partitions, x_partitions, y_err_partitions):
    # Compute the log likelihood for each partition separately
    pointwise_log_likelihood_samples = np.array(
        [[log_likelihood(y_partitions[i], theta, x_partitions[i], y_err_partitions[i])
          for i in range(len(y_partitions))]]
        for theta in theta_samples]
```

```

)

# Compute the lppd and p_waic for each partition
lppd = np.log(np.mean(np.exp(pointwise_log_likelihood_samples), axis=0))
p_waic = np.var(pointwise_log_likelihood_samples, axis=0)
# Check if the any of the terms in p_waic are too large, which indicates
# a problem
if np.any(p_waic > 0.4):
    print(f"Warning: Var[log p(y_i|theta)] > 0.4 for data points "
          f"{np.argwhere(p_waic > 0.4)}. p_WAIC unreliable!")
# Sum up the partitions
lppd = np.sum(lppd)
p_waic = np.sum(p_waic)

return -2*(lppd - p_waic), p_waic, pointwise_log_likelihood_samples

```

```

WAIC_lin, p_WAIC_lin, pointwise_log_likelihood_samples_lin = WAIC(
    theta_samples=results_linear["PPD_params"],
    log_likelihood=log_likelihood_lin,
    y_partitions=y,
    x_partitions=x,
    y_err_partitions=y_err
)

WAIC_quad, p_WAIC_quad, pointwise_log_likelihood_samples_quad = WAIC(
    theta_samples=results_quadratic["PPD_params"],
    log_likelihood=log_likelihood_quad,
    y_partitions=y,
    x_partitions=x,
    y_err_partitions=y_err
)

print(f"Linear: WAIC = {WAIC_lin:.1f}, p_WAIC = {p_WAIC_lin:.1f}")
print(f"Quadratic: WAIC = {WAIC_quad:.1f}, p_WAIC = {p_WAIC_quad:.1f}")

```

```

Warning: Var[log p(y_i|theta)] > 0.4 for data points [[ 2]
[29]]. p_WAIC unreliable!
Warning: Var[log p(y_i|theta)] > 0.4 for data points [[25]
[29]]. p_WAIC unreliable!
Linear: WAIC = -57.7, p_WAIC = 4.0
Quadratic: WAIC = -67.7, p_WAIC = 4.2

```

The WAIC also prefers the quadratic model. Warnings and oddly high number of parameters should give us pause, however.

Cross-validation

Both the DIC and WAIC are approximations to leave-one-out cross-validation (LOO-CV).

The idea behind cross-validation is to split the data into a training set y_{train} and a holdout set y_{holdout} .

We then evaluate the PPD (using the posterior $p(\theta|y_{\text{train}})$) at y_{holdout} .

This checks if the model can predict each data point given all the other data points.

This is a very clean approach to model checking but it usually assumes that there are n i.i.d. data points, or at least that the data can be partitioned as such. For correlated data, that might not be the case.

More specifically, we split the data into n partitions and for each compute

$$\log \int p(y_i|\theta)p(\theta|y_{(-i)})d\theta,$$

where y_i is the partition i of the data and $y_{(-i)}$ is the data without the partition i .

This becomes computationally expensive quickly, since for each partition we need to create samples from the posterior $p(\theta|y_{(-i)})$. It also explicitly requires the data to be able to be split into partitions, which might be difficult for structured or dependent data.

A sophisticated approximation to LOO-CV that avoids recalculating the posterior for the partitions is implemented in the package `arviz`. It also implements WAIC and other statistical diagnostics, for example for MCMC.

```
import arviz

inference_data_lin = arviz.from_dict(
    posterior={"lin": results_linear["PPD_params"][None, ...]},
    log_likelihood={"lin": pointwise_log_likelihood_samples_lin[None, ...]},
)

inference_data_quad = arviz.from_dict(
    posterior={"quad": results_quadratic["PPD_params"][None, ...]},
    log_likelihood={"quad": pointwise_log_likelihood_samples_quad[None, ...]},
)

arviz.compare({"lin": inference_data_lin, "quad": inference_data_quad}, ic="loo", scale="deviance")
```

	rank	elpd_loo	p_loo	elpd_diff	weight	se	dse	warning	scale
quad	0	-66.855284	4.660655	0.000000	0.793203	8.325738	0.000000	False	deviance
lin	1	-57.497176	4.131351	9.358108	0.206797	14.849195	9.453856	False	deviance

Clicker

How does LOO-CV conceptually differ from WAIC or DIC when comparing models?

- LOO-CV is less computationally intensive than WAIC or DIC, making it more suitable for large datasets.
- WAIC and DIC rely on resampling methods, while LOO-CV only uses a point estimate from the posterior distribution.

- LOO-CV penalizes models more heavily for complexity than WAIC or DIC, which tend to overfit the data.
- LOO-CV directly measures how well the model predicts unseen data by holding out one data point at a time, while WAIC and DIC estimate predictive performance using the entire dataset.

Supernovae

Let us apply the goodness-of-fit tests and DIC to the supernova example!

First, the chi-squared test:

```
n_data = len(data)
n_param_flat_lcdm = 3
n_param_lcdm = 4

chi_squared_flat_lcdm = -2*log_likelihood_flat_LCDM(theta_MAP_flat_lcdm)
chi_squared_lcdm = -2*log_likelihood_LCDM(theta_MAP_lcdm)

PTE_flat_lcdm = scipy.stats.chi2(df=n_data - n_param_flat_lcdm).sf(chi_squared_linear)
PTE_lcdm = scipy.stats.chi2(df=n_data - n_param_lcdm).sf(chi_squared_quadratic)

print(f"Flat LCDM:  $\chi^2$ ={{chi_squared_flat_lcdm:.2f}}, "
      f"ndof={{n_data}}-{{n_param_flat_lcdm}}, PTE={{PTE_flat_lcdm:.3f}}")

print(f"LCDM:  $\chi^2$ ={{chi_squared_lcdm:.2f}}, "
      f"ndof={{n_data}}-{{n_param_lcdm}}, PTE={{PTE_lcdm:.3f}}")

Flat LCDM:  $\chi^2$ =1441.69, ndof=1580-3, PTE=1.000
LCDM:  $\chi^2$ =1441.41, ndof=1580-4, PTE=1.000
```

The χ^2 values are too low for the number of data points. This suggests that the data uncertainties are overestimated. Let us run an MCMC chain (with `zeus` in this case) and look at the PPD test statistic.

```
import zeus

n_param = 4
n_walker = 12
n_step = 100

start_point = np.random.normal(size=(n_walker, n_param), loc=theta_MAP_lcdm, scale=(0.1, 0.1))
sampler_lcdm = zeus.EnsembleSampler(nwalkers=n_walker, ndim=n_param, logprob_fn=log_prob)
sampler_lcdm.run_mcmc(start=start_point, nsteps=n_step, progress=True)

print("Integrated autocorrelation time:", zeus.AutoCorrTime(sampler_lcdm.get_chain()).mean())

chain_lcdm = sampler_lcdm.get_chain(discard=50, flat=True)
```

```
Initialising ensemble of 12 walkers...
Sampling progress : 100%|██████████| 100/100 [00:28<00:00, 3.50it/s]
```

Integrated autocorrelation time: [5.44045274 4.71651602 5.50330958 7.05414514]

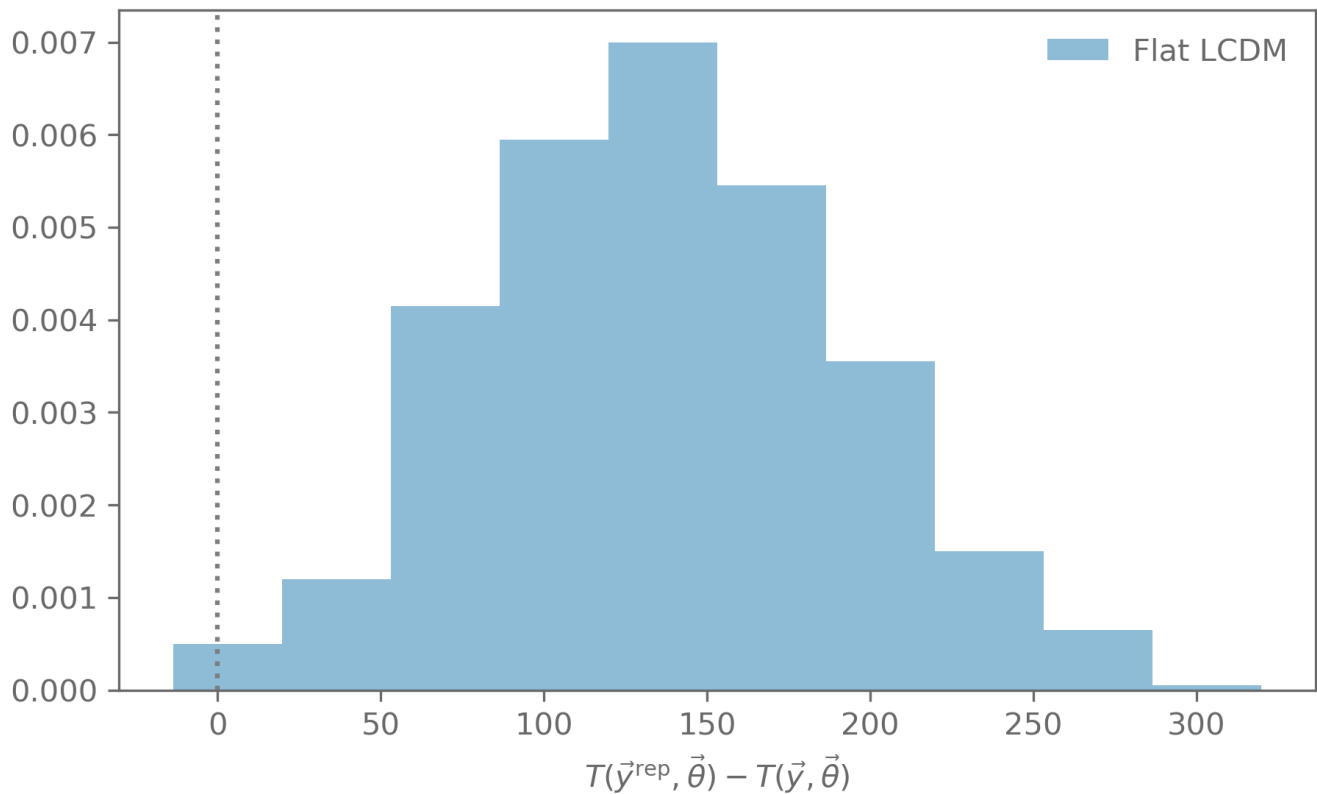
We use the χ^2 as the test statistic again:

```
def test_statistic(data, mu):
    r = data - mu
    return r @ inv_covariance @ r

T_rep_flat_lcdm = np.array([
    test_statistic(
        data=sample_from_likelihood_flat_LCDM(p),
        mu=flat_LCDM_distance_modulus_model(p, z_data)
    ) for p in chain_flat_lcdm
])
T_flat_lcdm = np.array([
    test_statistic(
        data=data,
        mu=flat_LCDM_distance_modulus_model(p, z_data)
    ) for p in chain_flat_lcdm
])

ppd_pte_flat_lcdm = (T_rep_flat_lcdm > T_flat_lcdm).sum()/T_rep_flat_lcdm.size
print(f"Flat LCDM PPD PTE: {ppd_pte_flat_lcdm:.3f}")
```

Flat LCDM PPD PTE: 0.995



This also indicates that the observed data do not scatter as much as our model suggests.

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

We already compared the flat and non-flat Λ CDM models using the Bayes' ratio. Now we see what the DIC tells us:

```
DIC_flat_lcdm, p_D_flat_lcdm, p_V_flat_lcdm = DIC(
    theta_star=theta_MAP_flat_lcdm,
    theta_samples=chain_flat_lcdm,
    log_likelihood=log_likelihood_flat_LCDM
)
DIC_lcdm, p_D_lcdm, p_V_lcdm = DIC(
    theta_star=theta_MAP_lcdm,
    theta_samples=chain_lcdm,
    log_likelihood=log_likelihood_LCDM
)

print(f"Flat LCDM: DIC = {DIC_flat_lcdm:.1f}, p_D = {p_D_flat_lcdm:.1f}, p_V = {p_V_flat_lcdm:.1f}")
print(f"LCDM: DIC = {DIC_lcdm:.1f}, p_D = {p_D_lcdm:.1f}, p_V = {p_V_lcdm:.1f}")

Flat LCDM: DIC = 1446.1, p_D = 2.2, p_V = 2.0
LCDM: DIC = 1447.5, p_D = 3.1, p_V = 3.5
```

The DIC slightly prefers the simpler, flat Λ CDM model.

We also see that the estimates of the number of model parameters makes sense: the flat Λ CDM model has 3 parameters (H_0 , Ω_m , M) but H_0 is completely constrained by the prior. So effectively, the data can only inform ≈ 2 parameters.

The non-flat model has 4 parameters but again, H_0 is completely constrained by the prior, so we are left with ≈ 3 effective free parameters.

Exercise

Reproduce the chi-squared and PPD-based test with your own analysis code.

Show that for a Gaussian posterior with a wide and uniform prior, the estimate p_V of the effective number of free parameters reduces to the dimensionality of the Gaussian.

Try a cubic model. What are the DIC, WAIC, and Bayes factor?