MCMC fitting with emcee

7th PYTHON MEETING

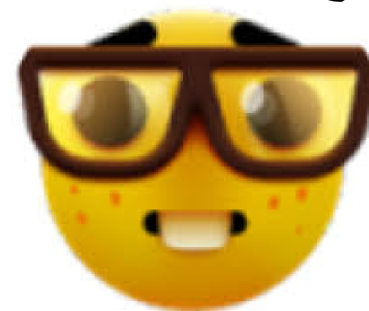
Marcelo Ojeda

What does MCMC mean?

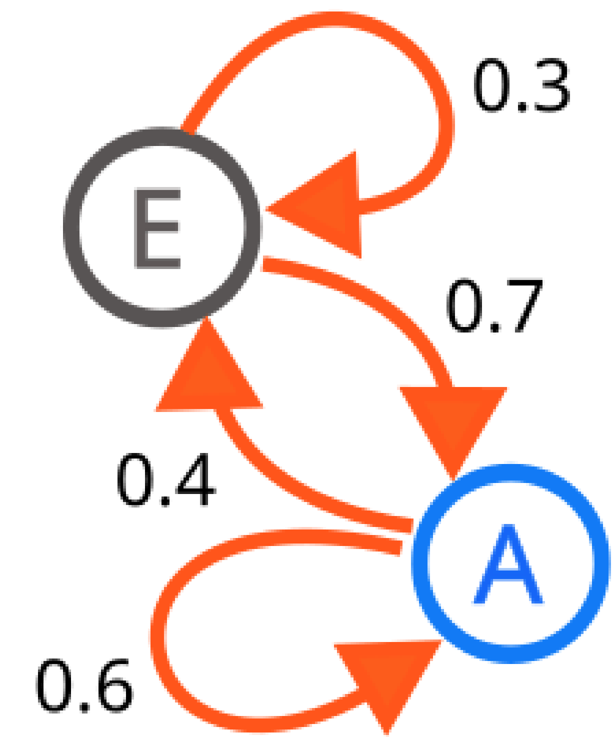
spoiler: Markov Chain Monte Carlo

Markov Chains

Markov chain or Markov process is a stochastic process describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event.



Andrey Markov



Two-state Markov process

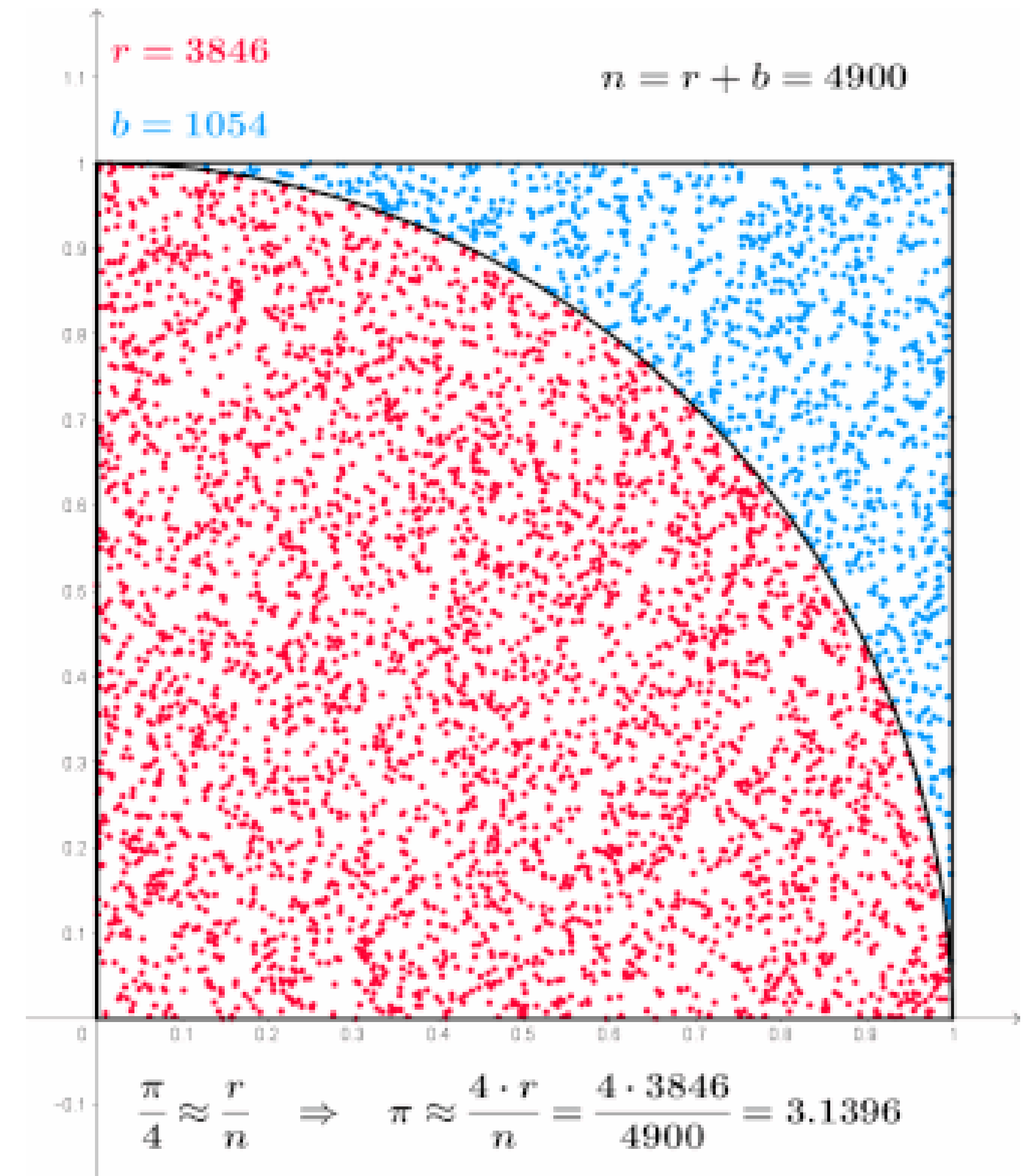
Cool video by Veritasium

*Example of NOT Markov process: Probability of shifting to a specific gear while driving (depends on whether you were accelerating or braking)

Monte Carlo

Broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle.

Monte-Carlo Casino

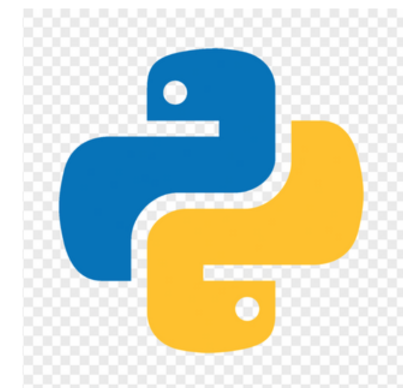


Approximating π using Monte Carlo

Markov Chain Monte Carlo

Remembering previous state + Guessing and exploring

How can we use this to fit models to data?



Installation

Package managers

The recommended way to install the stable version of emcee is using pip

```
python -m pip install -U pip
pip install -U setuptools setuptools_scm pep517
pip install -U emcee
```

or conda

```
conda update conda
conda install -c conda-forge emcee
```

https://emcee.readthedocs.io/en/stable/

The Ensemble Sampler

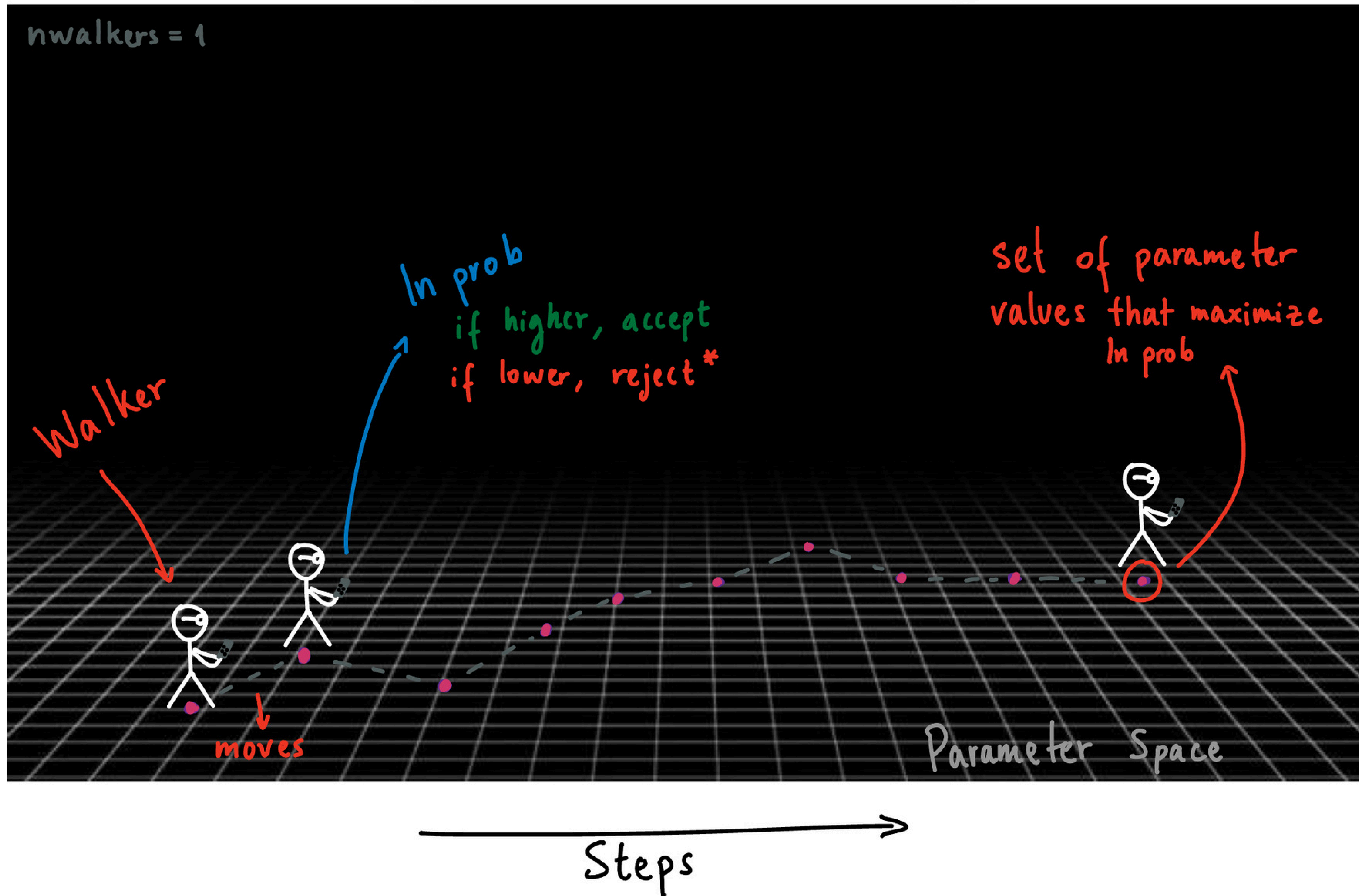
Standard usage of `emcee` involves instantiating an `EnsembleSampler`.

```
class emcee.EnsembleSampler(nwalkers, ndim, log_prob_fn, pool=None, moves=None,
args=None, kwargs=None, backend=None, vectorize=False, blobs_dtype=None,
parameter_names: Dict[str, int] | List[str] | None = None, a=None, postargs=None,
threads=None, live_dangerously=None, runtime_sortingfn=None)
```

Parameters:

- **nwalkers** (*int*) – The number of walkers in the ensemble.
- **ndim** (*int*) – Number of dimensions in the parameter space.
- **log_prob_fn** (*callable*) – A function that takes a vector in the parameter space as input and returns the natural logarithm of the posterior probability (up to an additive constant) for that position.
- **moves** (*Optional*) – This can be a single move object, a list of moves, or a “weighted” list of the form `[(emcee.moves.StretchMove(), 0.1), ...]`. When running, the sampler will randomly select a move from this list (optionally with weights) for each proposal. (default: `StretchMove`)
- **args** (*Optional*) – A list of extra positional arguments for `log_prob_fn`. `log_prob_fn` will be called with the sequence `log_prob_fn(p, *args, **kwargs)`.
- **pool** (*Optional*) – An object with a `map` method that follows the same calling sequence as the built-in `map` function. This is generally used to compute the log-probabilities for the ensemble in parallel.

In a nutshell...



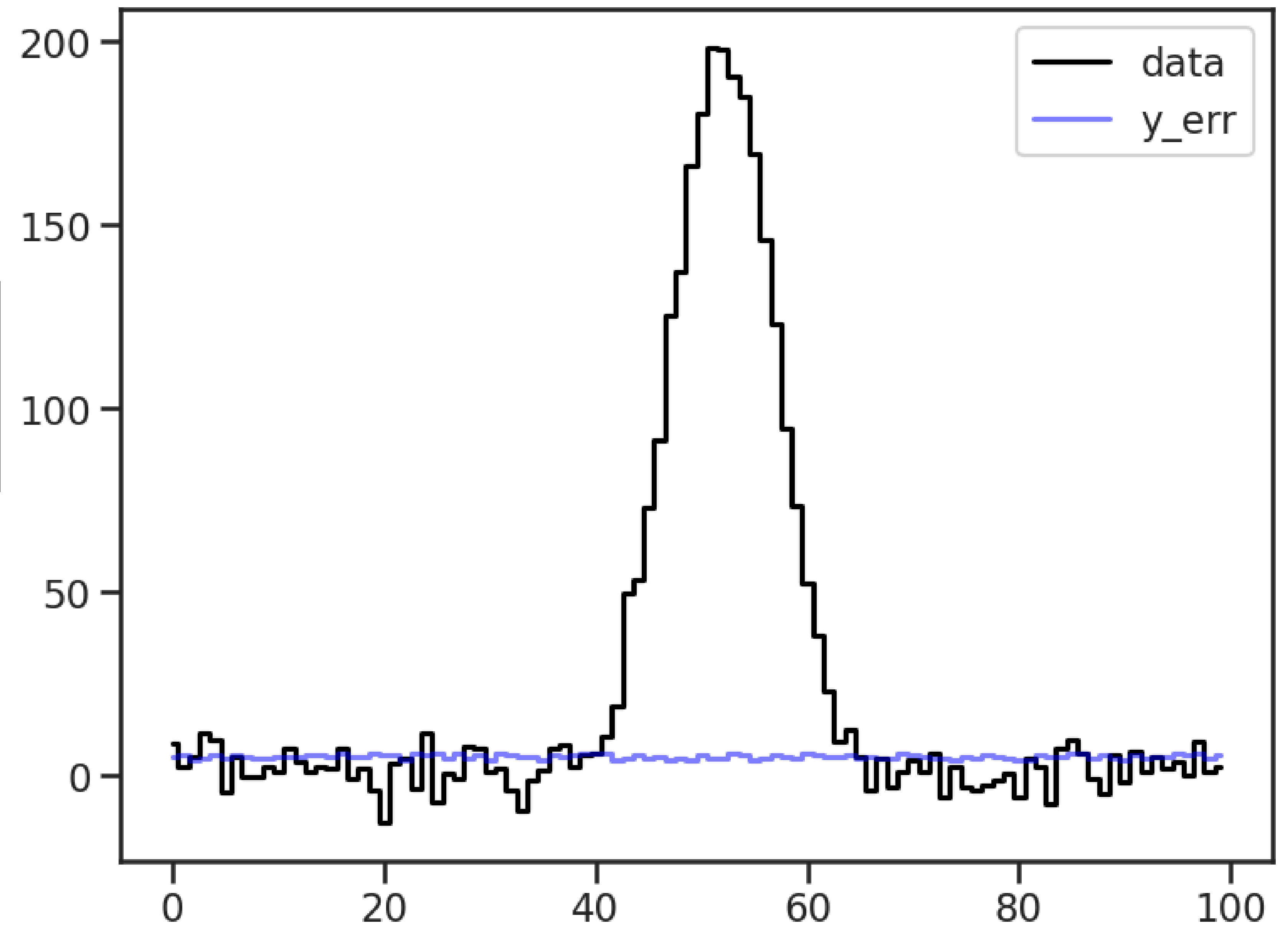
Recipe for fitting with emcee

- Data
- Model
- Logarithm of posterior probability (lnprob)
  - lnprior
  - lnlike (Likelihood)
- Parameters for the Ensemble Sampler
  - nwalkers
  - nsteps
  - ndim
  - pool
  - moves

Example

Fitting a Gaussian profile

```python
def model(x, a, b, c):
    # gaussian
    return a * np.exp(-0.5 * ((x - b) / c) ** 2)
```



Defining lnprob

```python
# theta is a vector that contains the parameters of the model
def lnprior(theta):
    a, b, c = theta

    # flat prior, boundaries
    if 0 < a < 500 and 0 < b < 100 and 2 < c < 7:
        return 0.0
    return -np.inf


def lnlike(theta, x, y, yerr):
    a, b, c = theta
    model_y = model(x, a, b, c)

    # gaussian likelihood
    return -0.5 * np.sum(np.log(2 * np.pi * yerr ** 2) +
                         (y - model_y) ** 2 /
                         yerr ** 2)
```

```python
def lnprob(theta, x, y, yerr):
    lp = lnprior(theta)
    if not np.isfinite(lp):
        return -np.inf

    lnMeasured = lnlike(
        theta,
        x,
        y,
        yerr)
    if not np.isfinite(lnMeasured):
        return -np.inf

    return lp + lnMeasured
```

Parameters for Ensemble Sampler

```
x = df['x'].values
y = df['y'].values
yerr = df['yerr'].values


ndim = 3
nwalkers = 100
nsteps = 1000
nburn = 200  # steps to discard at the beginning of the chain
```

Initialize walkers and Ensemble Sampler

```
starting_guesses = []

for i in range(nwalkers):
    aux = [
        np.random.uniform(100, 150),  # a
        np.random.uniform(50, 55),  # b
        np.random.uniform(3, 4)  # c
    ]
    starting_guesses.append(aux)
```

```
sampler = emcee.EnsembleSampler(
    nwalkers,
    ndim,
    lnprob,
    args=(x, y, yerr)
)
```

Run MCMC (sampler.sample)

```
currenttime = time.time()
Step = 1
for pos, prob, state in sampler.sample(
        starting_guesses, iterations=nsteps):
    print('Step:', Step, '/', nsteps)
    print("Mean acceptance fraction: %f" % (np.mean(
        sampler.acceptance_fraction
    )))
    print("Mean lnprob and Max lnprob values: %f %f" % (
        np.mean(prob), np.max(prob)
    ))
    print(
        "Time to run previous set of walkers (seconds): %f" %
        (time.time() - currenttime))
    currenttime = time.time()
    Step += 1
```

Explore the results

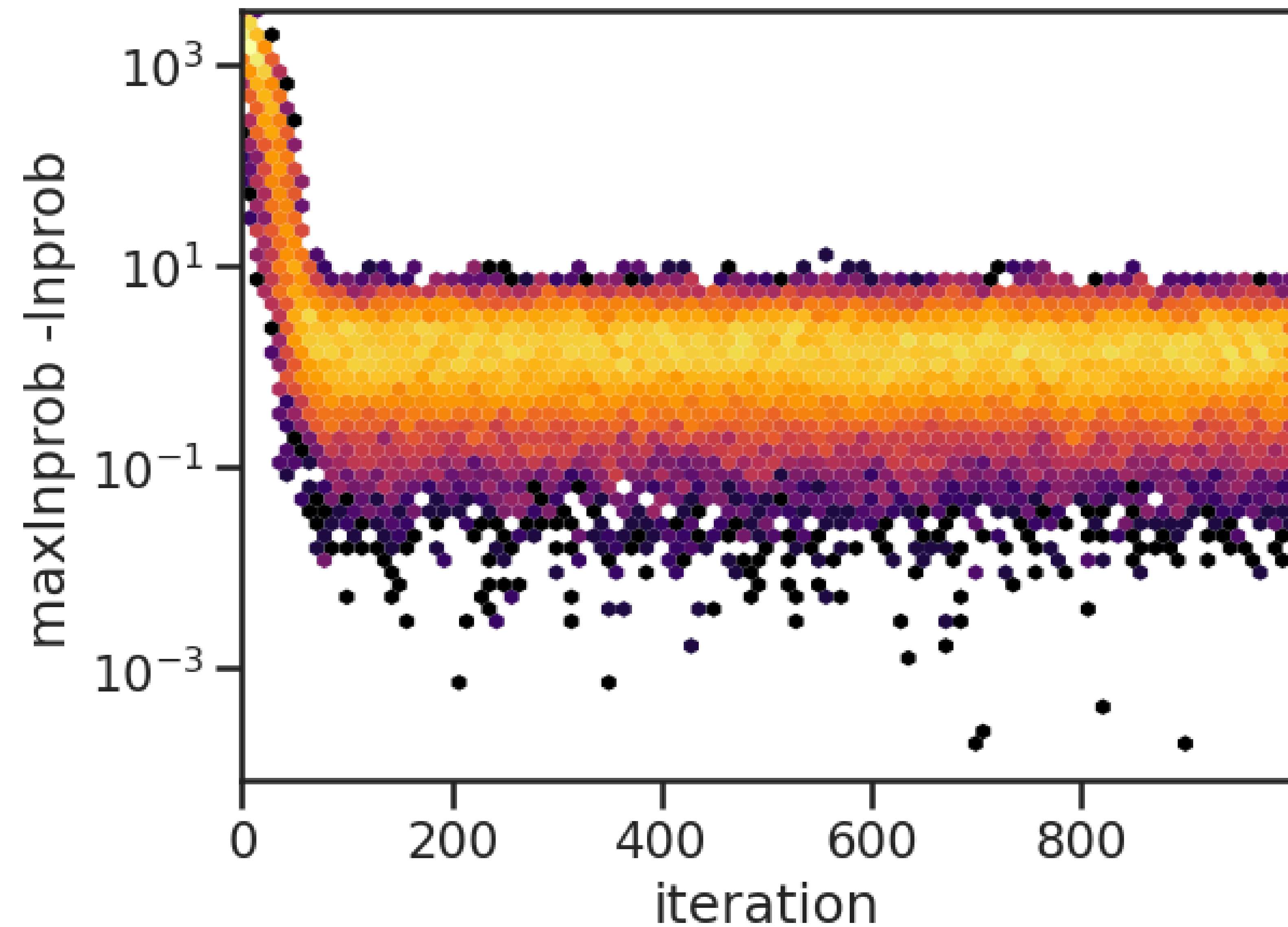
```
emcee_trace = sampler.chain[:, :, :].reshape(
    (-1, ndim))
lnprob = sampler.lnprobability
```

From here we can extract the posterior distributions of the parameters

```
theta = emcee_trace[np.argmax(lnprob)] # best fit parameters
```

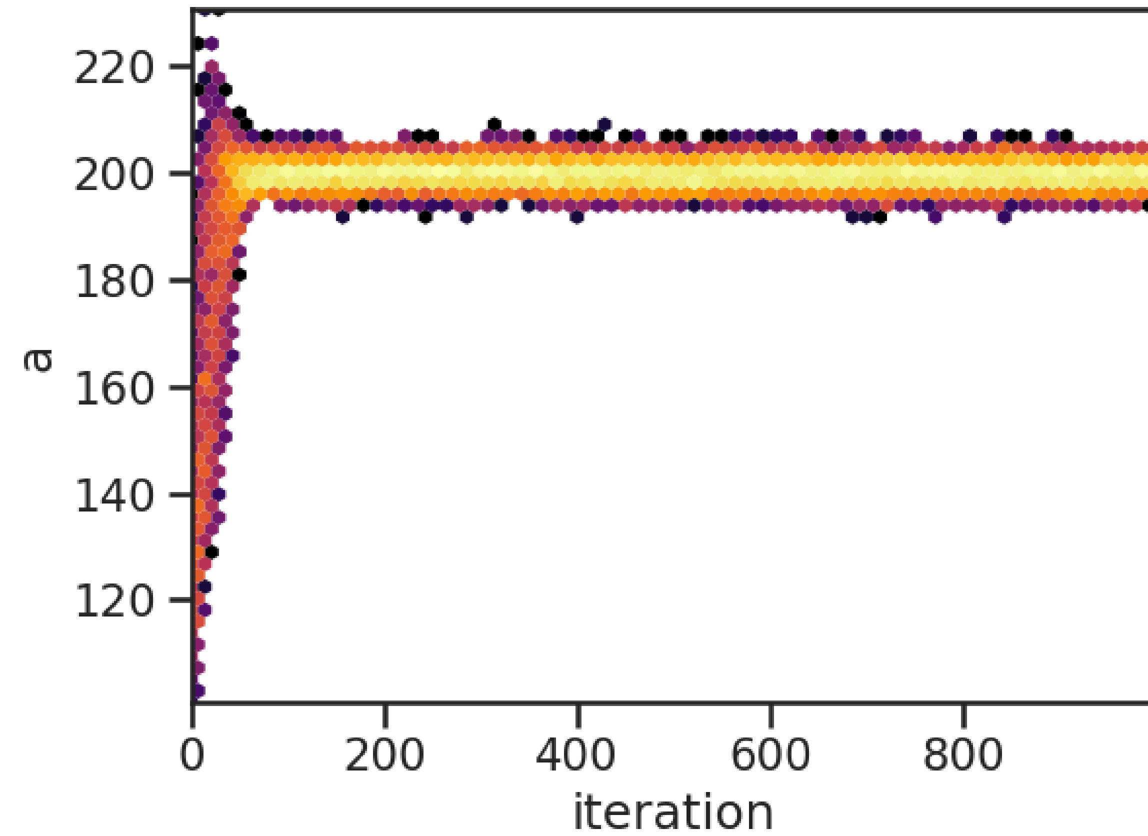
Best fit

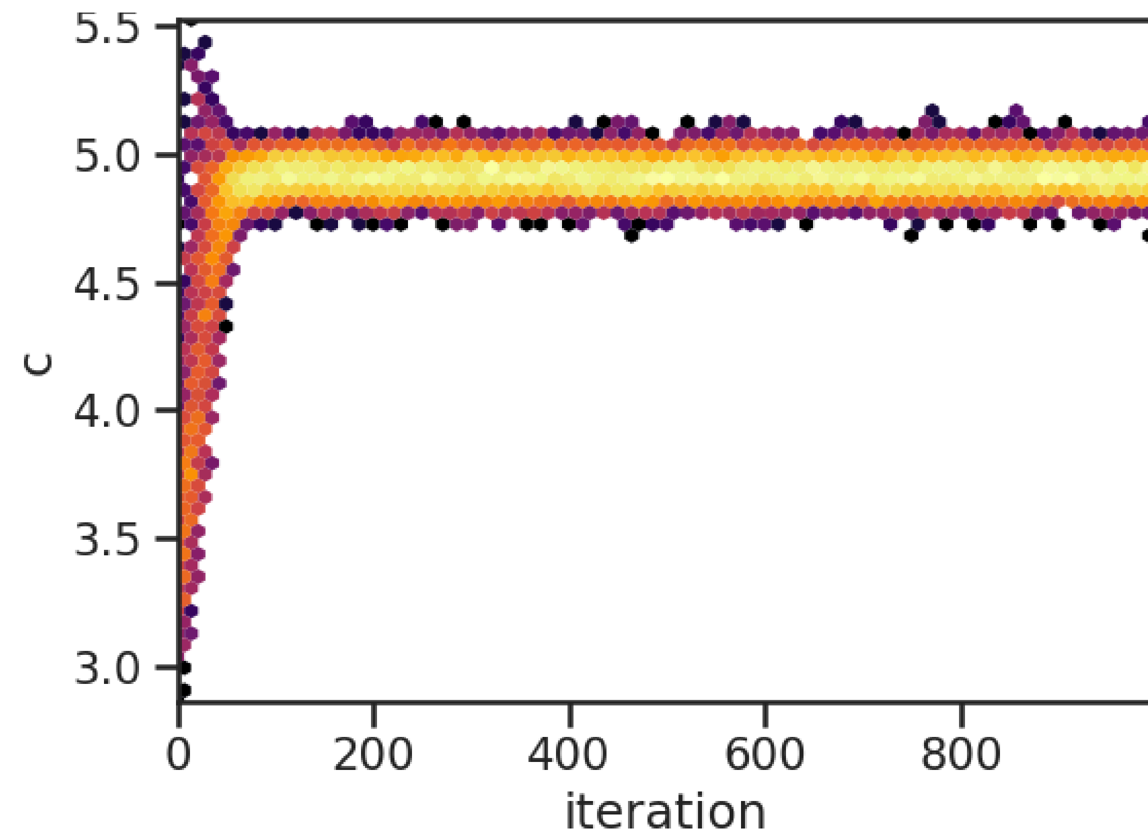
Trace of lnprob

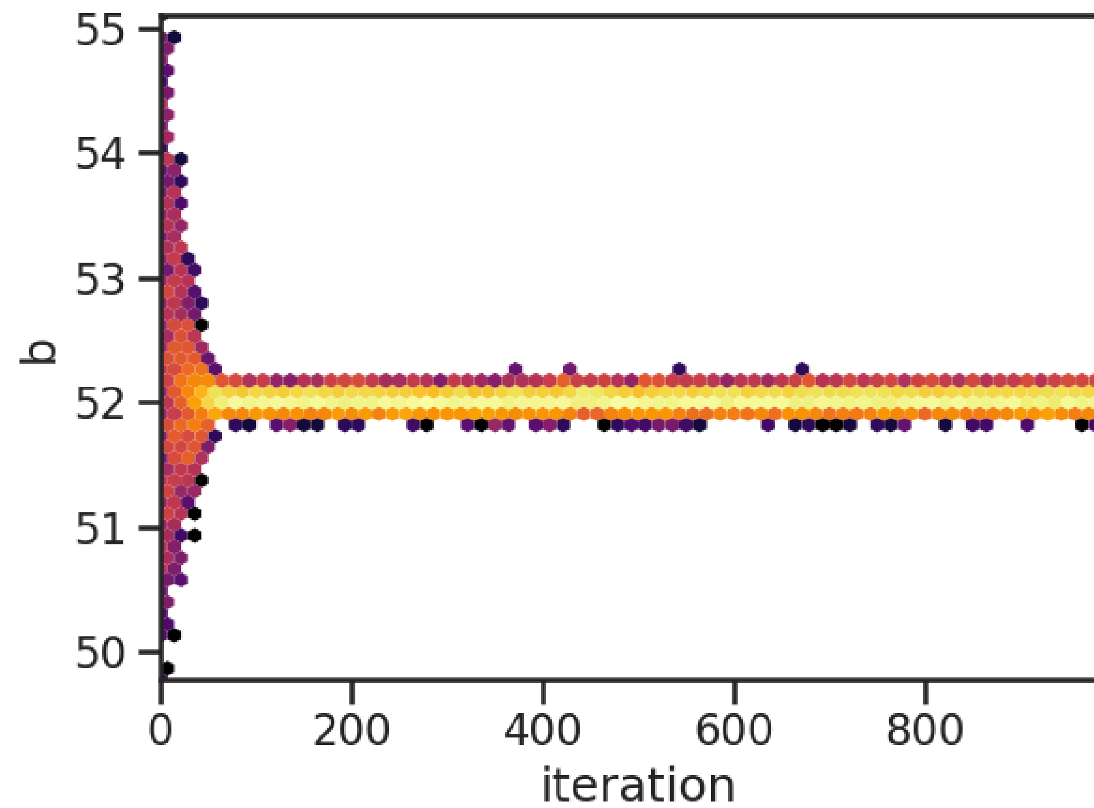



```
x = np.array([])
y = np.array([])
maxlnprob = np.max(lnprob)
for i in range(len(lnprob)):
    x = np.append(x, range(len(lnprob[i])))
    y = np.append(y, maxlnprob - lnprob[i])
plt.figure()
plt.hexbin(
    x[y > 0],
    y[y > 0],
    gridsize=[70, 30],
    cmap='inferno',
    bins='log',
    mincnt=1,
    yscale='log',
    linewidths=0)
plt.ylabel('maxlnprob -lnprob')
plt.xlabel('iteration')
try:
    plt.xlim(min(x), max(x))
    plt.ylim(min(y), max(y))
except Exception:
    print('Negative values in Convergence....')
```

Traces of the parameters



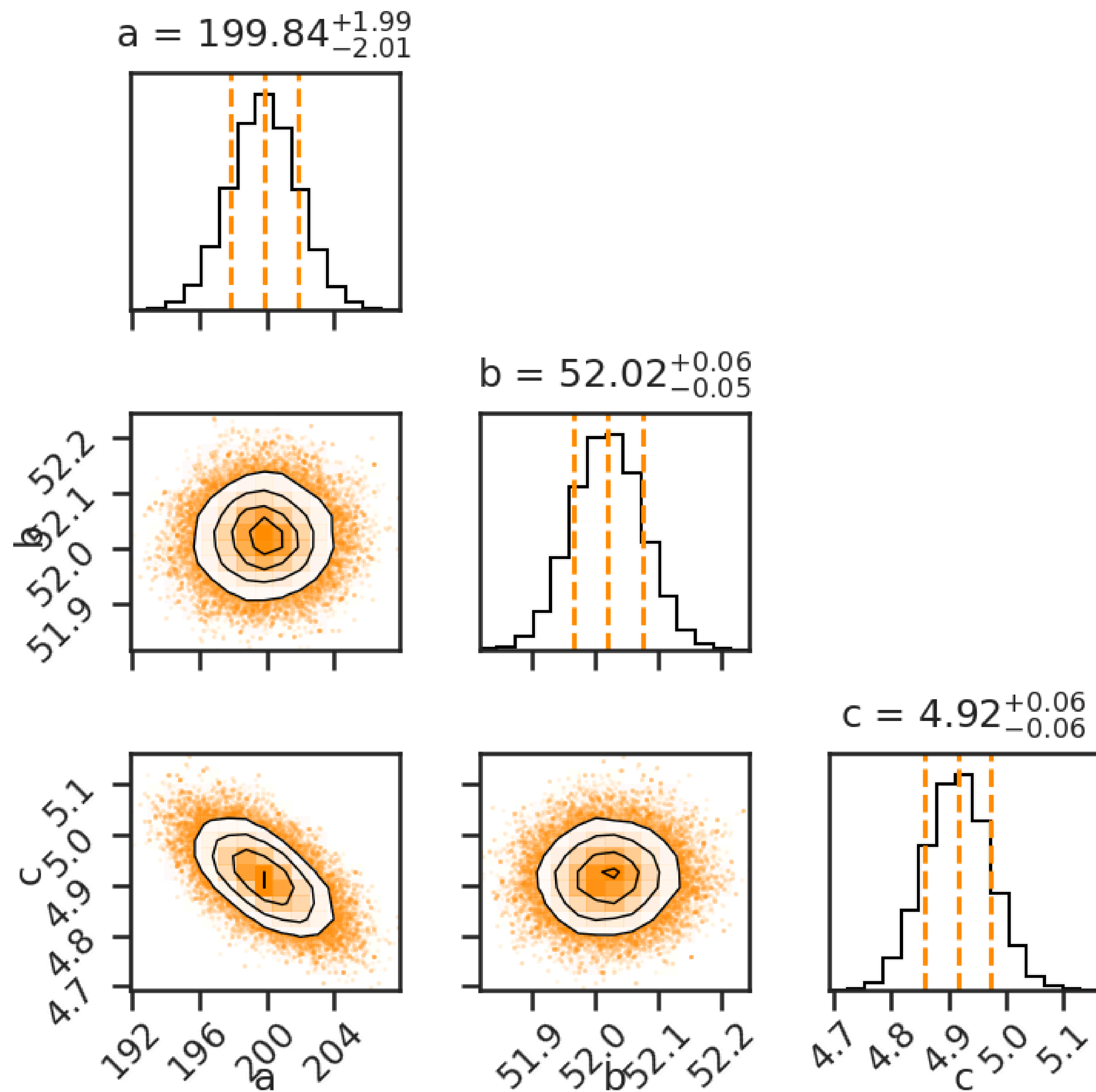





```python
for ID in range(ndim):
    plt.figure()
    x = np.array([])
    y = np.array([])
    for i in sampler.chain:
        x = np.append(x, range(len(i.T[ID])))
        y = np.append(y, i.T[ID])

    plt.figure()
    if (max(y) / min(y)) > 50 and len(y[y < 0]) < 1:
        plt.hexbin(
            x,
            y,
            gridsize=[70, 30],
            cmap='inferno',
            bins='log',
            mincnt=1,
            yscale='log',
            linewidths=0
        )
    else:
        plt.hexbin(
            x,
            y,
            gridsize=[70, 30],
            cmap='inferno',
            bins='log',
            mincnt=1,
            linewidths=0
        )
    plt.ylabel(ll[ID])
    plt.xlabel('iteration')
    plt.xlim(min(x), max(x))
    plt.ylim(min(y), max(y))
```

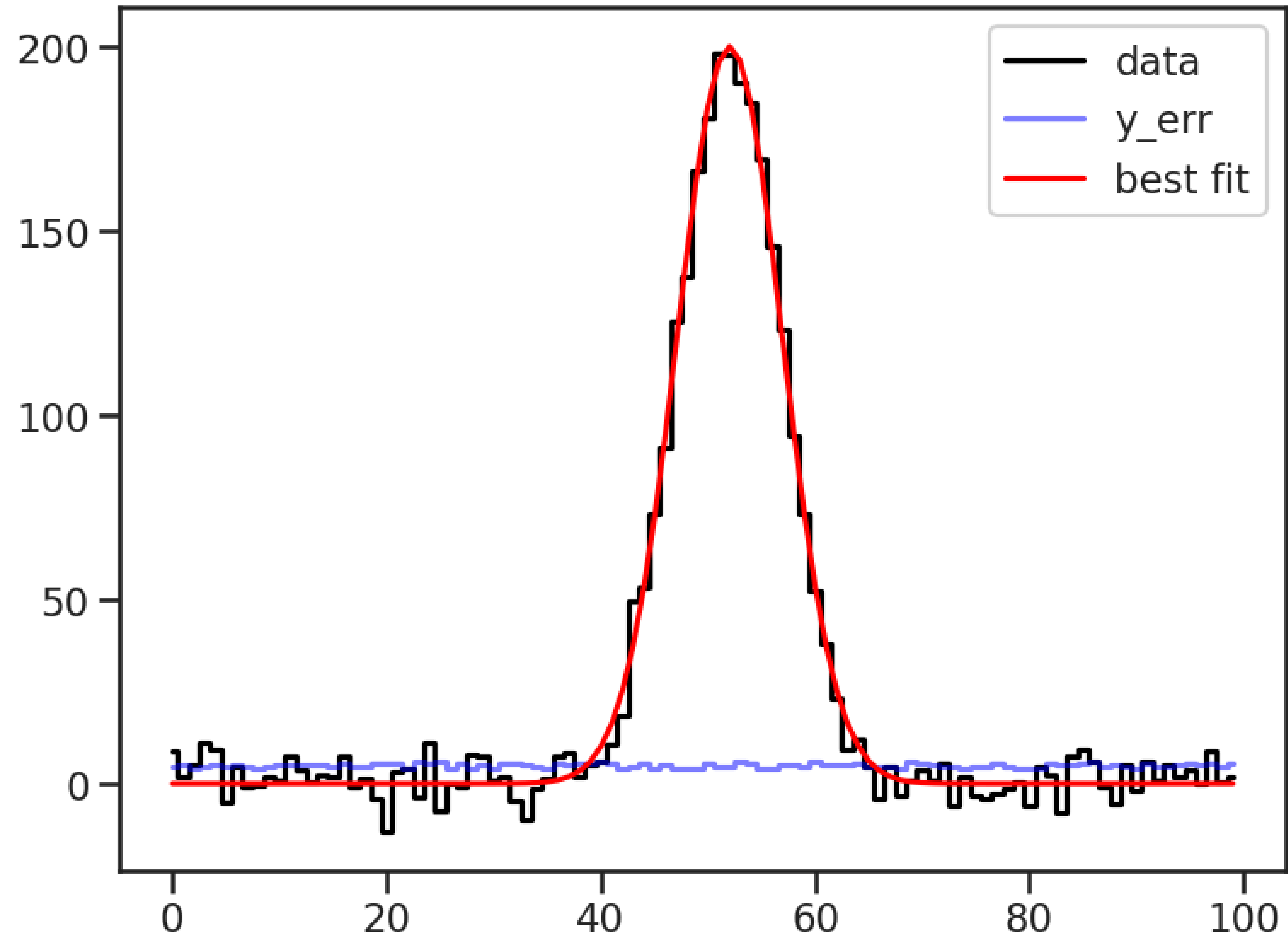
Corner plots




```
samples = sampler.chain[:, nburn:, :].reshape(
    (-1, ndim))

fig = corner.corner(
    samples,
    labels=ll,
    title_kwargs={'y': 1.05},
    title_fmt=".2f",
    use_math_text=True,
    bins=15,
    quantiles=[0.16, 0.5, 0.84],
    show_titles=True,
    color='DarkOrange',
    hist_kwargs={'color': 'black', 'linewidth': 1.5},
    contour_kwargs={'linewidths': 1, 'colors': 'black'}
```

Best fit over data



Percentiles of the posteriors

```python
df_results = pd.DataFrame()

new_row = {}

for i in range(len(ll)):
    new_row[ll[i] + '_bestfit'] = emcee_trace[np.argmax(lnprob)][i]
    new_row[ll[i] + '_16'] = np.percentile(emcee_trace.T[i], 16)
    new_row[ll[i] + '_50'] = np.percentile(emcee_trace.T[i], 50)
    new_row[ll[i] + '_84'] = np.percentile(emcee_trace.T[i], 84)
    new_row[ll[i] + '_mean'] = np.mean(emcee_trace.T[i])
    new_row[ll[i] + '_err'] = np.std(emcee_trace.T[i])

df_results = pd.concat([df_results, pd.DataFrame([new_row])], ignore_index=True)
df_results
```

✓ 0.0s

	a_bestfit	a_16	a_50	a_84	a_mean	a_err	b_bestfit	b_16	b_50	b_84	b_mean	b_err	c_bestfit	c_16	c_50	c_84	c_mean	c_err
0	199.788509	197.508949	199.737729	201.801712	198.486129	8.671935	52.020842	51.961949	52.01973	52.079663	52.026621	0.172289	4.918422	4.84762	4.912324	4.972186	4.882647	0.19456

Why MCMC and not curvefit()?

Depends...

MCMC is much more robust for fitting and determining uncertainties, and more reliable when trying to fit lots of parameters. More flexibility for exploring posterior distributions. Downside: computational cost. curvefit() might be more than enough for models with few parameters and small errors.

Thank you!