THE MARKOV CHAIN MONTE CARLO METHOD

David Kirkby, UC Irvine

LSSTC Data Science Fellows Program
Caltech, January 2017

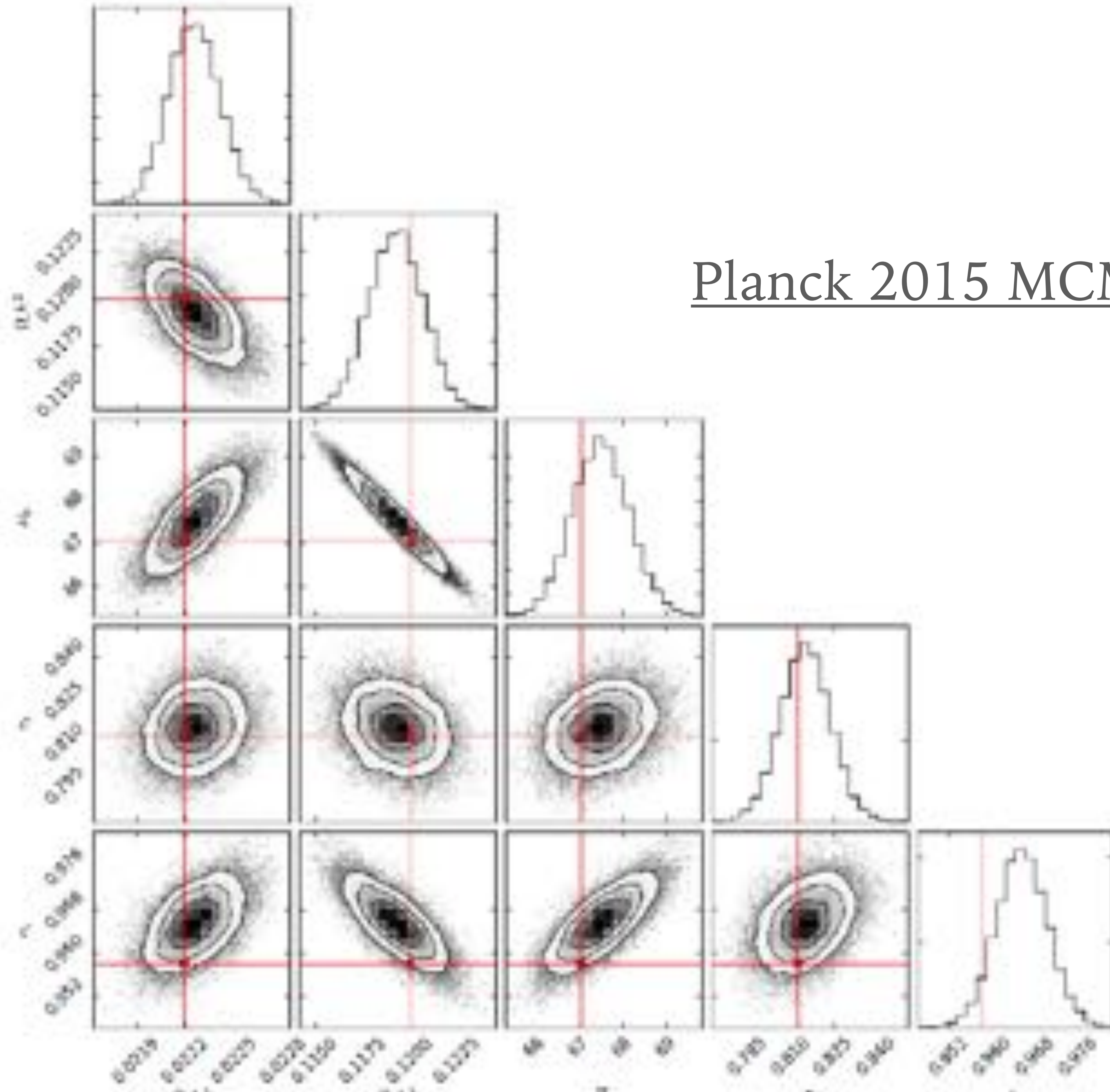
WHAT IS MARKOV CHAIN MONTE CARLO (MCMC)?

- MCMC is a computational tool to solve the following:
  - Given $p(\theta)$ proportional to some prob. density function:
    - $prob(\theta) = p(\theta)/N$
    - $p(\theta) \geq 0$
    - $N = \int p(\theta)\, d\theta \quad , 0 < N < \infty$
  - suppose you have a written a function to evaluate $p(\theta)$
    - `def p(theta): …`
  - generate values $\theta_1, \theta_2, \theta_3, \ldots$ sampled from this p.d.f.
    - `for i in range(10): print generate()`

WHY IS THIS PROBLEM IMPORTANT IN MACHINE LEARNING?

- Central to Bayesian inference:
  - $P(\theta|D) = P(D|\theta)\, P(\theta) \,/\, P(D)$
  - D = data, θ = parameters.
  - Evidence $P(D)$ is usually not practical to calculate.
  - Take $p(\theta) = P(D|\theta)\, P(\theta)$, which is often much easier to calculate.
- Can now use MCMC to generate a random sequence of parameter values.
- MCMC is often the only (or most straightforward) method for doing this.

HOW ARE RANDOM SAMPLES USEFUL?

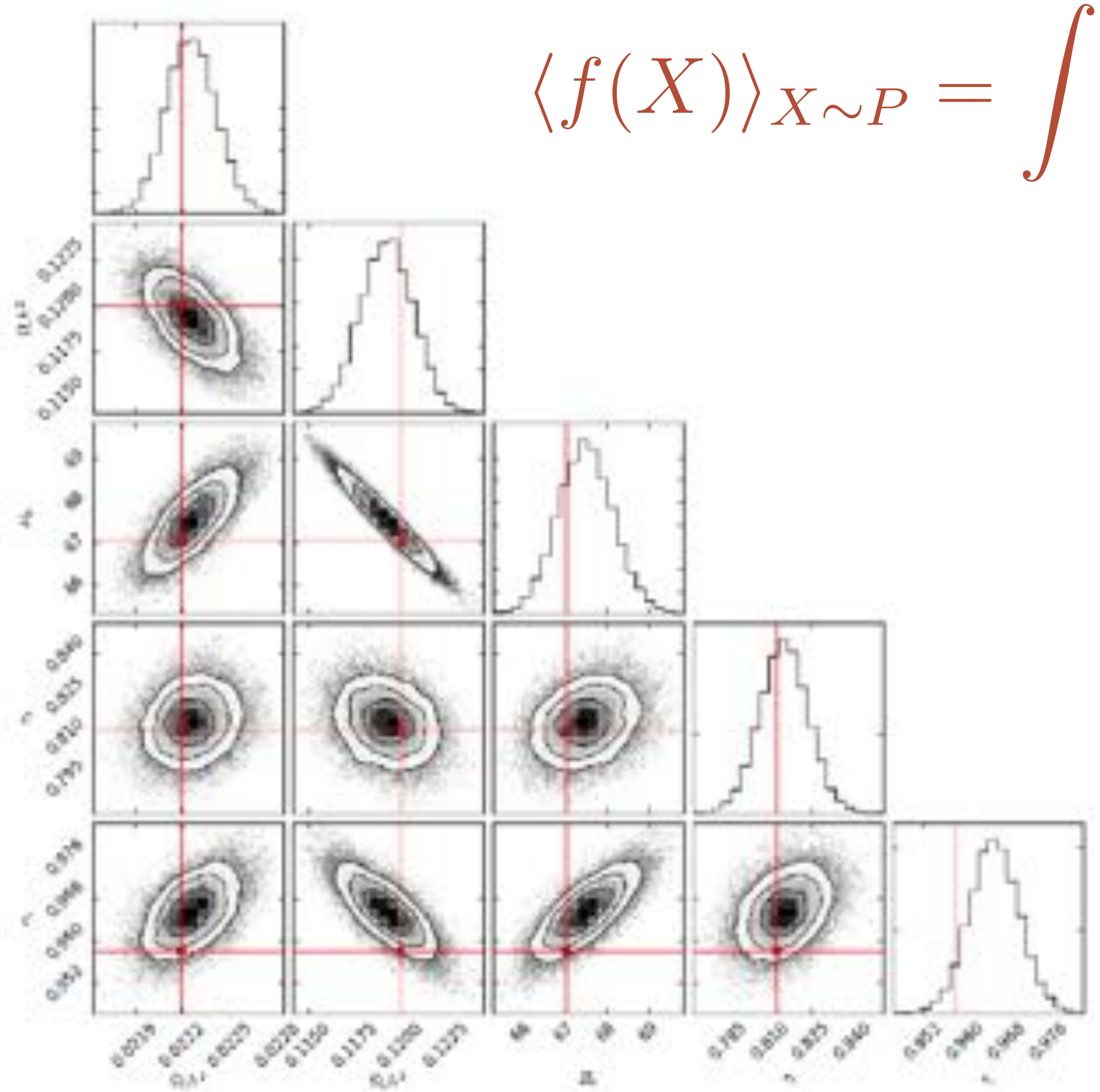


Planck 2015 MCMC Chains

HOW ARE RANDOM SAMPLES USEFUL?

- Can marginalize over any subset of (nuisance) parameters.
- Can estimate distributions of arbitrary functions of params.
- Can perform Monte-Carlo integrations:

$$\langle f(X) \rangle_{X \sim P} = \int f(x) P(x) dx \simeq \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

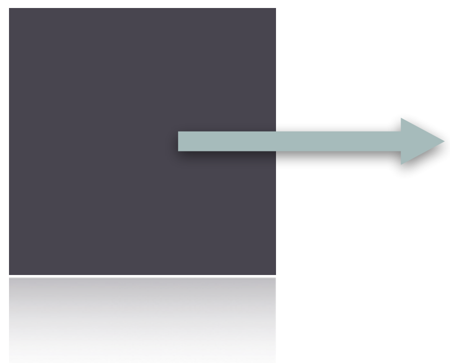


OUTLINE

- Stochastic Processes and Markov Chains
- Markov Chain Monte Carlo
- Practical Advice
- Exercise

STOCHASTIC PROCESSES

- A stochastic process is a black box generator of random samples:



$$X_1,\ X_2,\ \ldots,\ X_n,\ X_{n+1},\ \ldots$$

- In general, the next sample depends on all previous samples, *i.e.*, the samples form a correlated sequence
- For example:

```
history = []
def stochastic():
    history.append(random.uniform())
    return sum(history)
```

MARKOV CHAINS

- A Markov chain is a special type of stochastic process:

$$X_1, X_2, \ldots, X_n, X_{n+1}, \ldots$$

X_{n+1} depends <u>only</u> on X_n
(and not on $X_1, X_2, \ldots, X_{n-1}$)

- An important subset of Markov chains are "stationary":

$$X_1, X_2, \ldots, X_n, X_{n+1}, \ldots, X_m, X_{m+1}, \ldots$$

$$P(X_{n+1} \mid X_n) = P(X_{m+1} \mid X_m)$$

~time invariant

MARKOV CHAINS

- Another important property is “reversibility”:

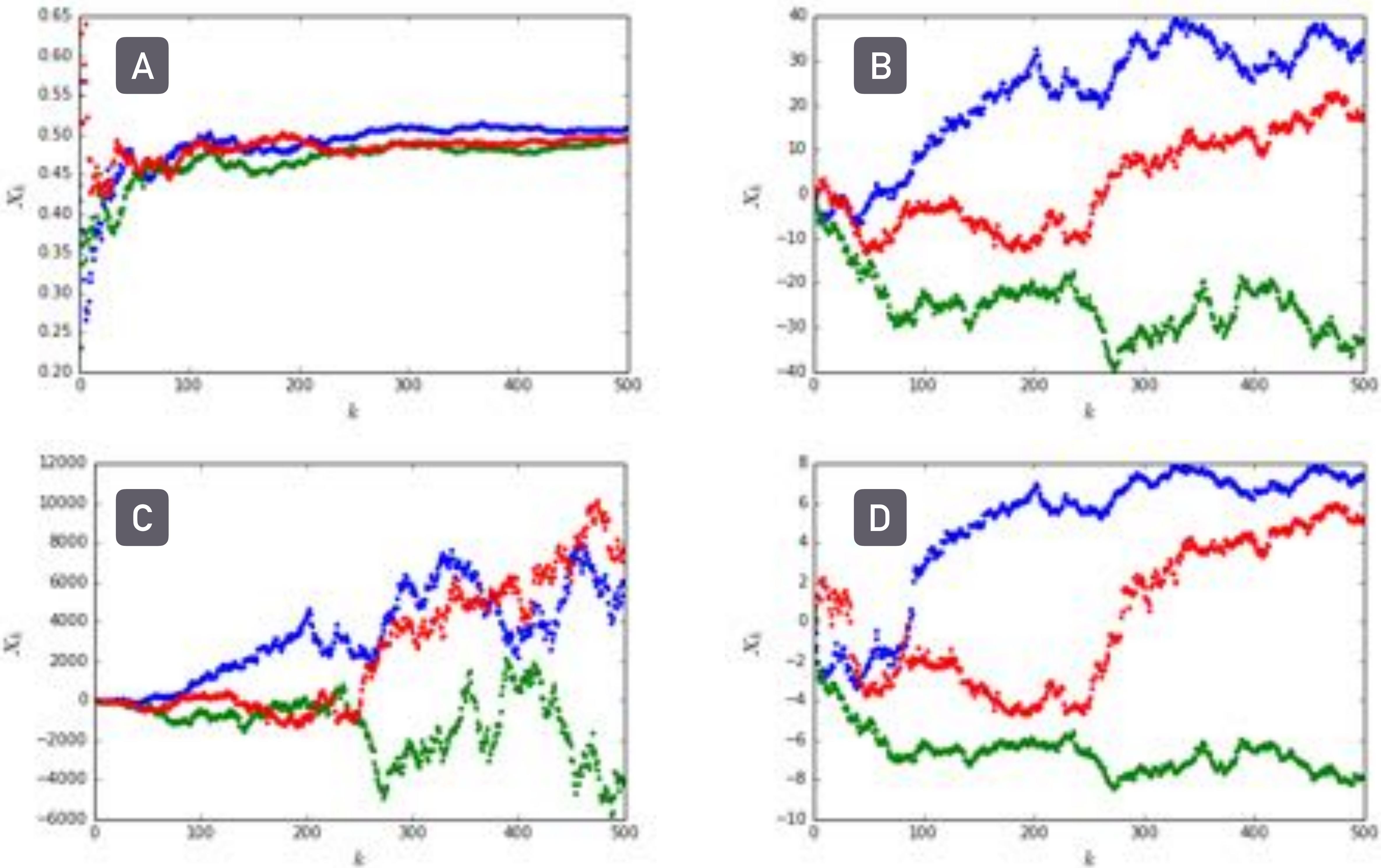
$$X_1,\ X_2,\ \ldots,\ \overbrace{X_n,\ X_{n+1},\ X_n}^{=x},\ \ldots$$

$$P(X_{n+1}\,|\,X_n = x) = P(X_{n+2} = x\,|\,X_{n+1})$$

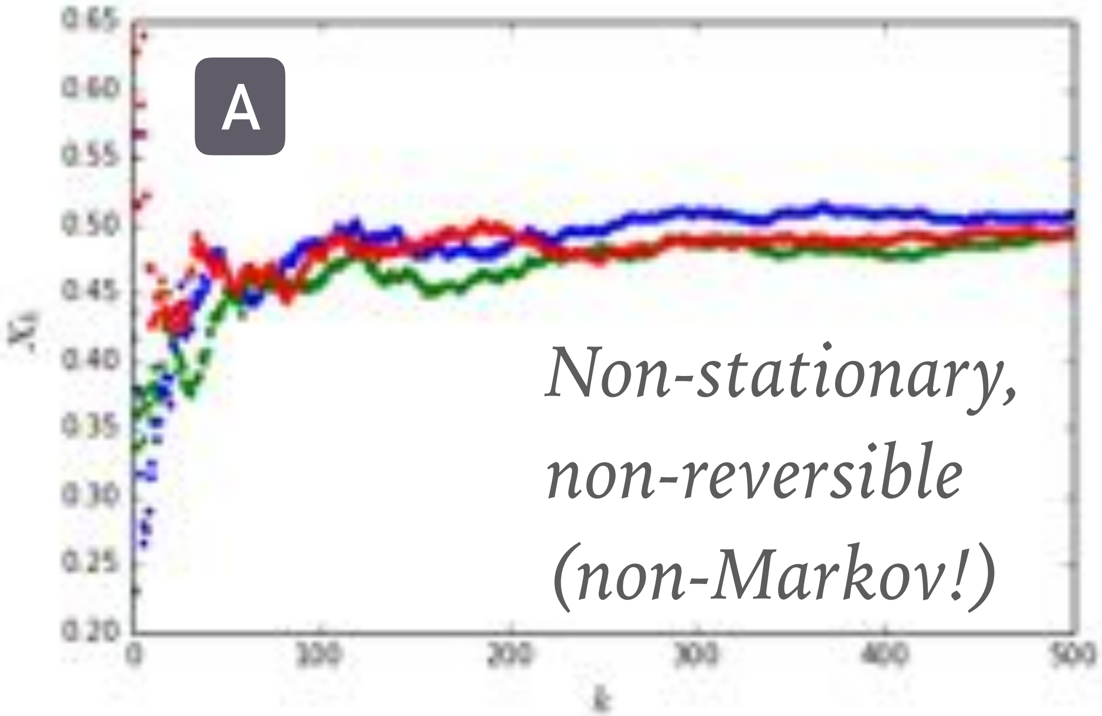
$\sim$*time-reversal invariant*

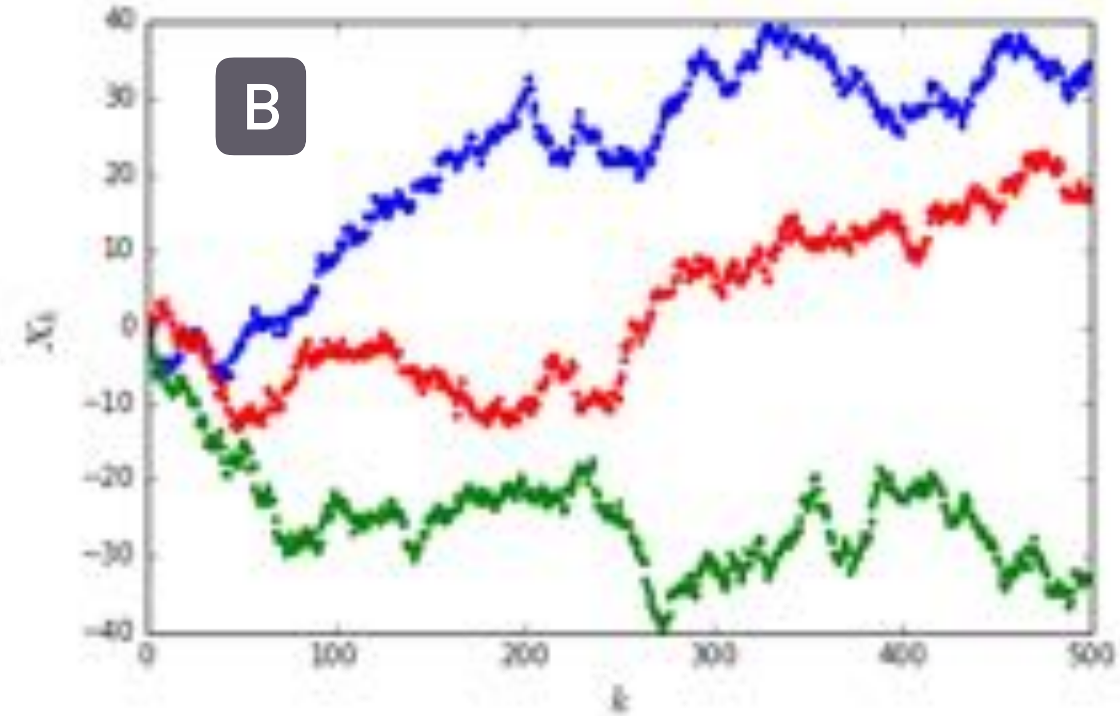
- A <u>reversible</u> chain is <u>stationary</u> but not vice versa.

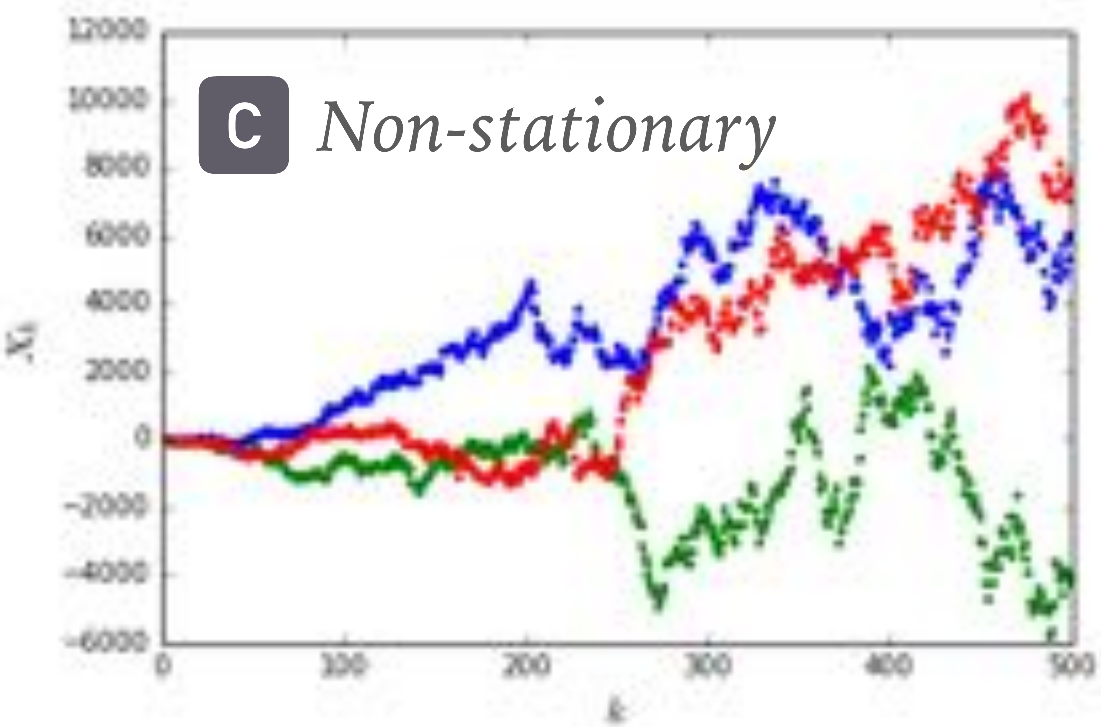
ACTIVITY: STATIONARY / NON-STATIONARY / NON-REVERSIBLE?

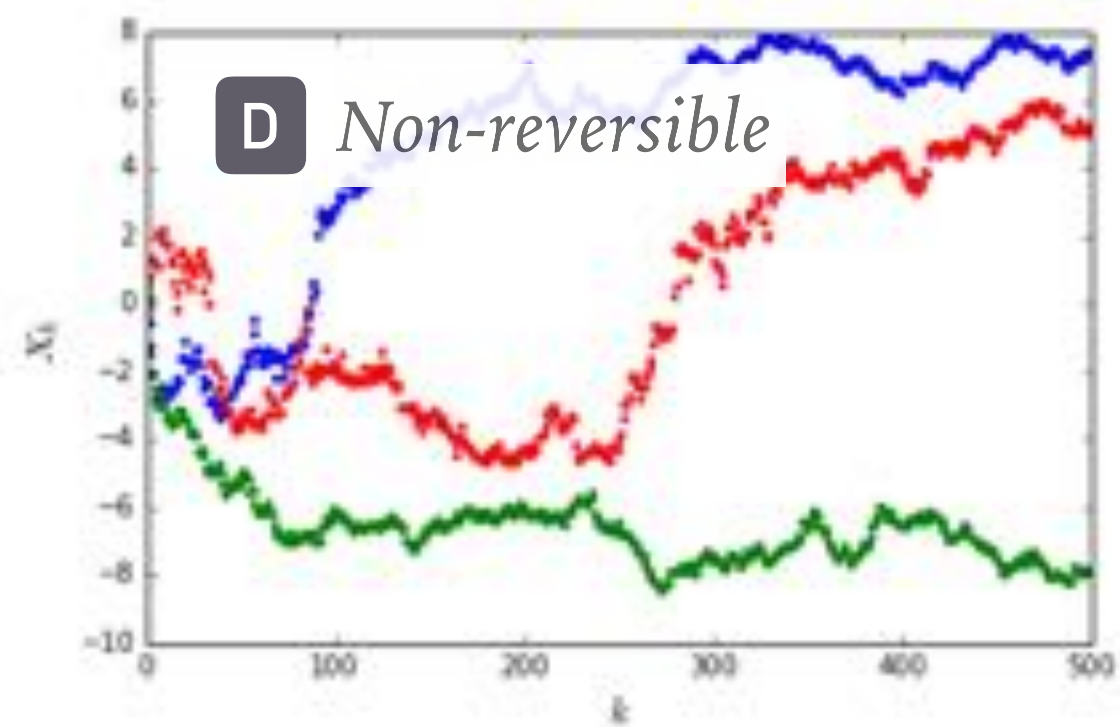


ACTIVITY: STATIONARY / NON-STATIONARY / NON-REVERSIBLE?









ACTIVITY: BUILD YOUR FIRST MARKOV CHAIN

- Build a Markov chain for the weather where you grew up:
    - Identify the main types of weather (raining, windy, …)
    - Write down the probabilities for each possible transition

tomorrow

today

Prob			
	0.50	0.25	0.25
			
			

row sum = 100%

- Enter your probabilities at http://setosa.io/markov/

```
[[0.50, 0.25, 0.25],
[…],
[…]]
```

ACTIVITY: IDENTIFY (APPROXIMATE) MARKOV CHAINS

- Is the sequence of letters in a book a Markov chain?
- Is the sequence of words in a book a Markov chain?

$$X_1, X_2, \ldots, X_n, X_{n+1}, \ldots$$

X_{n+1} depends <u>only</u> on X_n
(and not on $X_1, X_2, \ldots, X_{n-1}$)

- If you answered NO, is it at least approximately YES?
- How could you generate a random book with a Markov chain?

MARKOV CHAINS FOR NATURAL LANGUAGE

- Examples of randomly generated words
- Use a more general stochastic process for better results:
    - for practical implementation, use a “recurrent neural network” with “long-short-term memory”.
    - The unreasonable effectiveness of RNNs

```
static void num_serial_settings(struct tty_struct *tty)
{
  if (tty == tty)
    disable_single_st_p(dev);
  pci_disable_spool(port);
  return 0;
}
```

MARKOV CHAIN STATE SPACE

- The “state space” of a process is the set of all possible values.
- The weather and language examples have a finite state space.
- The possible values in a scientific application are usually real numbers: the state space is (uncountably) infinite.
  - Can no longer represent transition probabilities with a matrix or graph.
- The output values can also be multi-dimensional.
- The essential elements of Markov theory still hold in an infinite multi-dimensional state space.

MARKOV CHAIN DISTRIBUTIONS

- A Markov chain is specified by two distributions:

$$\boxed{1} \longrightarrow X_1, X_2, \ldots, X_n, X_{n+1}, \ldots$$

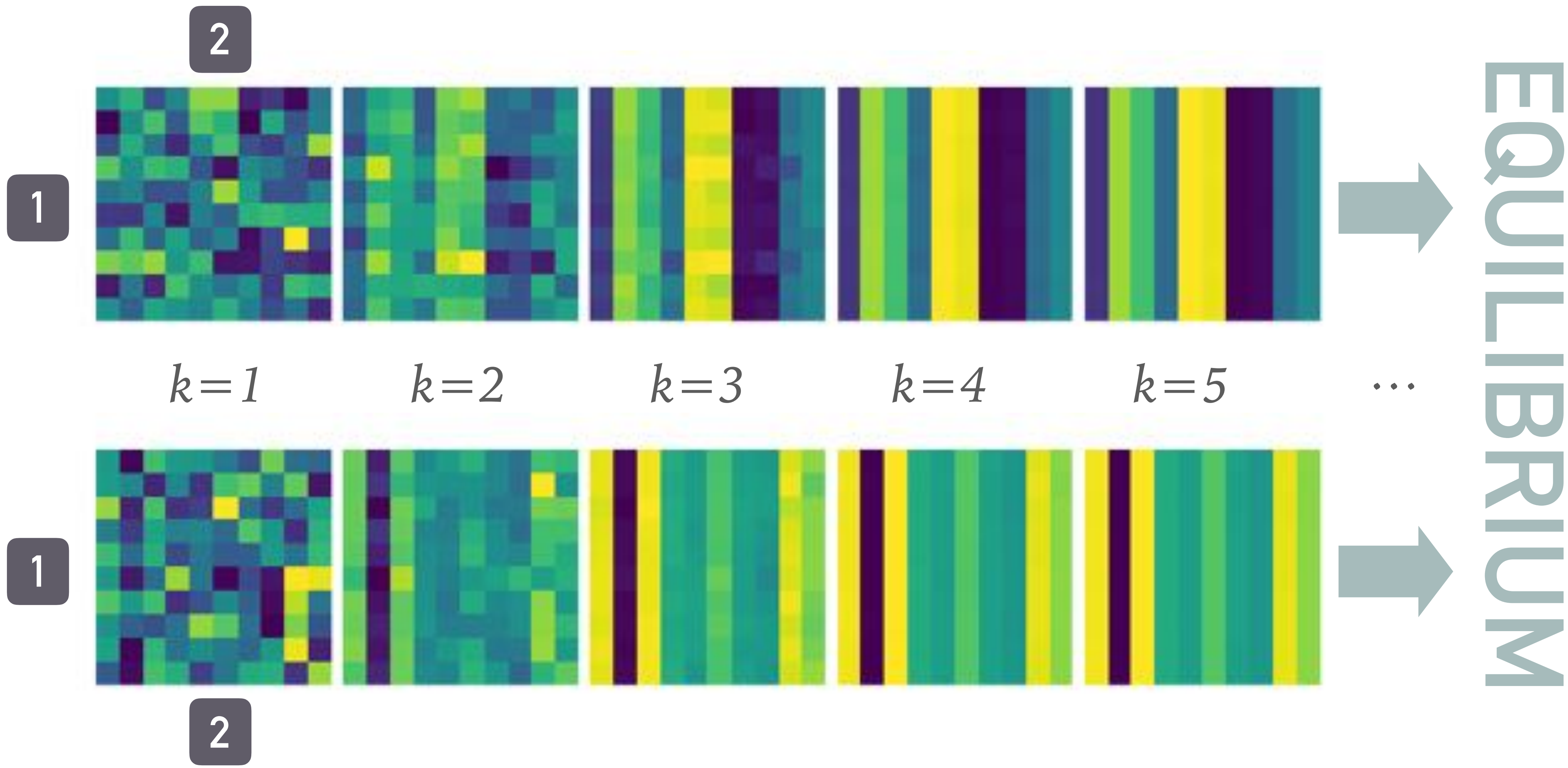
$$X_n \xrightarrow{\boxed{2}} X_{n+1}$$

```
class StationaryMarkov(object):

    def __init__(self):
        random.seed(seed)
        self.x = random.uniform()

    def generate(self):
        self.x += random.normal()
        return self.x
```

1. *Distribution of initial values $P(X_1)$*
2. *Transition probabilities $P(X_{n+1} | X_n)$*

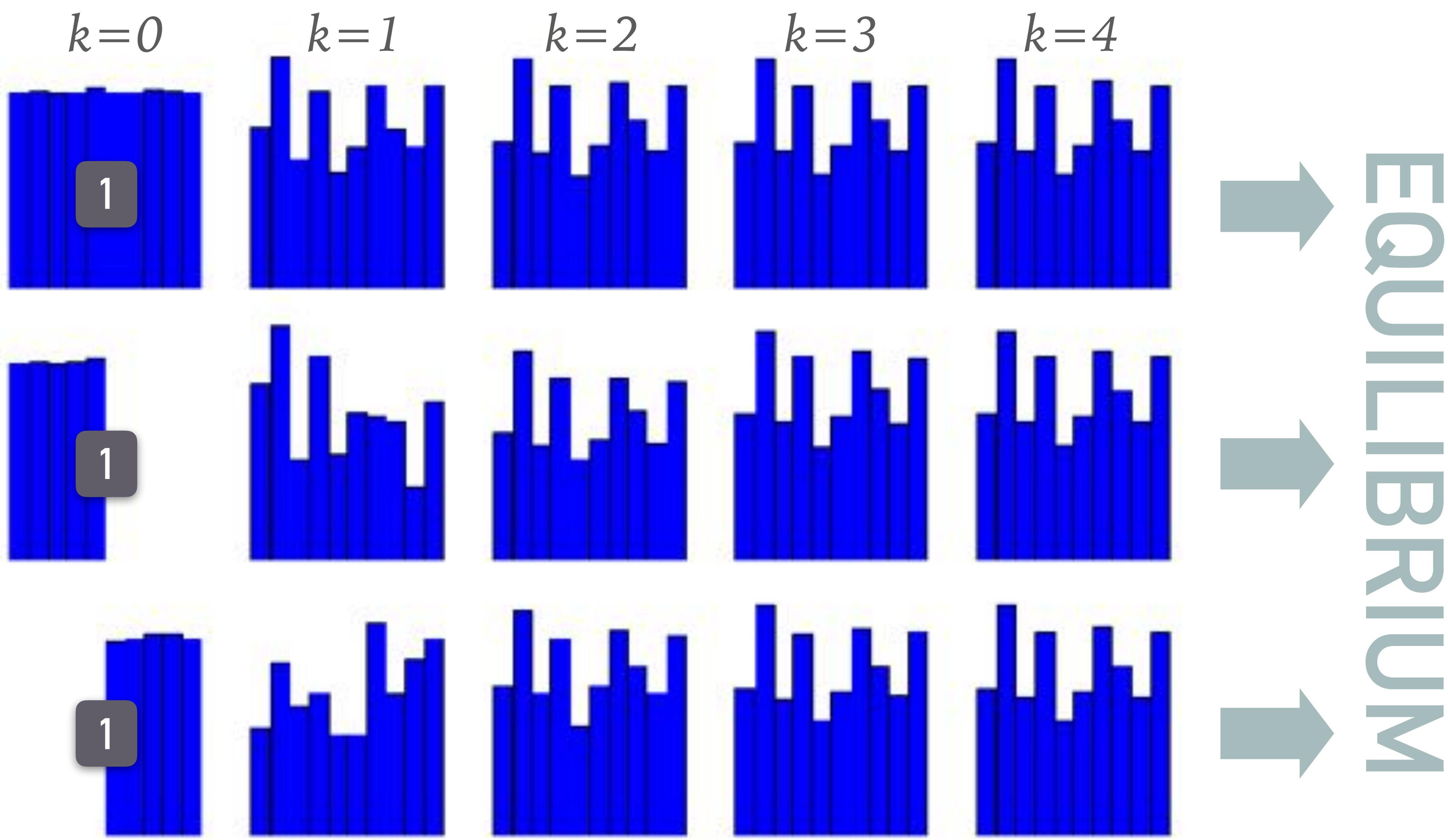
THE EQUILIBRIUM DISTRIBUTION OF A MARKOV CHAIN

- A stationary Markov chain eventually reaches an equilibrium distribution of states:



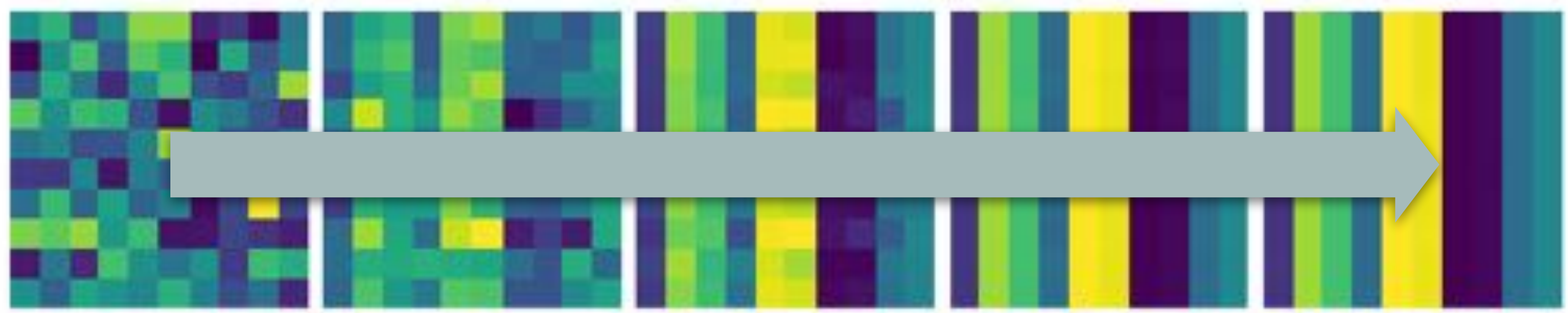
THE EQUILIBRIUM DISTRIBUTION OF A MARKOV CHAIN

- The equilibrium distribution depends only on the transition probabilities, and not on the initial distribution.



THE EQUILIBRIUM DISTRIBUTION OF A MARKOV CHAIN

- Given transition probabilities for a stationary Markov chain, we can generate samples from <u>some</u> distribution.



- To be useful, we want to <u>specify</u> a target distribution.
- This requires solving a difficult inverse problem!
  - Given the target distribution, select appropriate transition probabilities.
  - Transition probabilities are encoded in an "update rule".

MARKOV CHAIN UPDATE METHODS

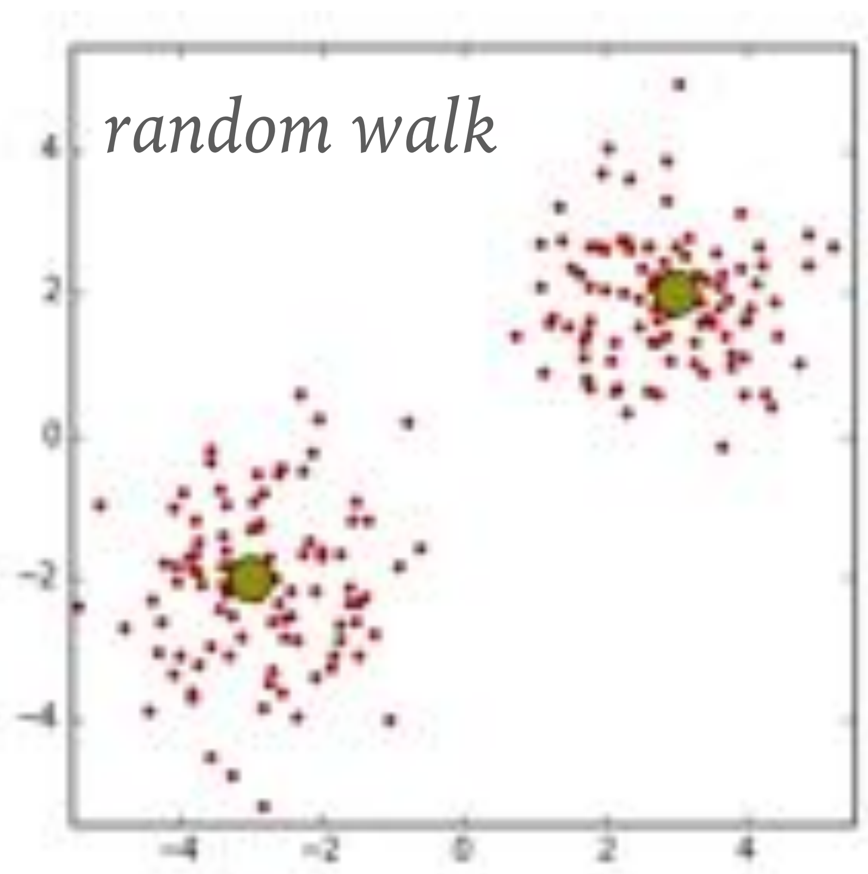
- MCMC algorithms do not use stationary Markov chains!
- Instead, they are carefully designed to have similar properties.
- All practical methods are special cases of the Metropolis-Hastings-Green algorithm:
  - Metropolis-Hastings
    - Metropolis
    - Gibbs
    - Hamiltonian
- The simpler Metropolis-Hastings algorithm contains the essential ideas, so we will focus on that.

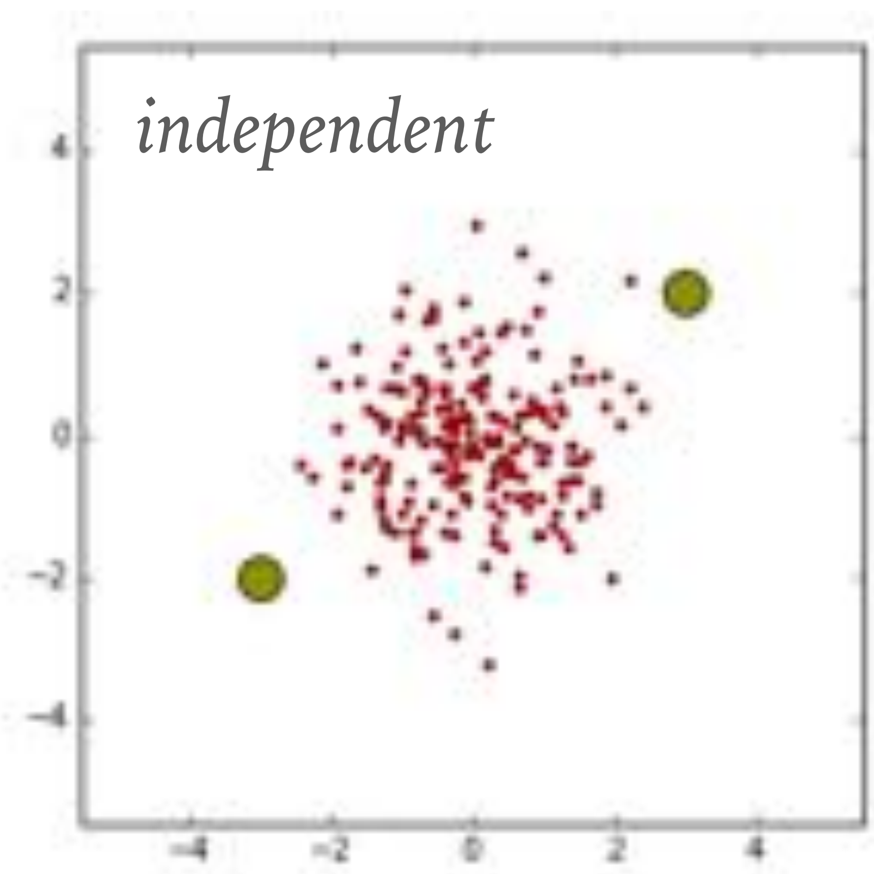
METROPOLIS-HASTINGS UPDATES

- Goal is to sample a target probability density $p(\theta)/N$.
  - N is too expensive to calculate, unlike $p(\theta)$.
  - target is typically a Bayesian posterior, but this is not required.
- Generate proposed updates $\theta_n \rightarrow \theta_{n+1}$ by sampling a distribution $q(\theta_{n+1} | \theta_n)$.
  - Chose q that is easier to sample than p
    (you don't need MCMC if you can sample p directly!)
  - q is often a (multivariate correlated) Gaussian, for convenience.
  - Choice of q affects the algorithm efficiency but not its validity.

METROPOLIS-HASTINGS UPDATES

```
sample = proposal_density.sample()
if mode == 'random-walk':
  new_theta = old_theta + sample
elif mode == 'independent':
  new_theta = sample
```





METROPOLIS-HASTINGS UPDATES

- Calculate the "Hastings ratio":

$$r(\theta_n,\theta_{n+1}) = \frac{p(\theta_{n+1})\; q(\theta_n \,|\, \theta_{n+1})}{p(\theta_n)\;\; q(\theta_{n+1} \,|\, \theta_n)}$$

- Accept the proposed update $\theta_n \rightarrow \theta_{n+1}$ with probability:

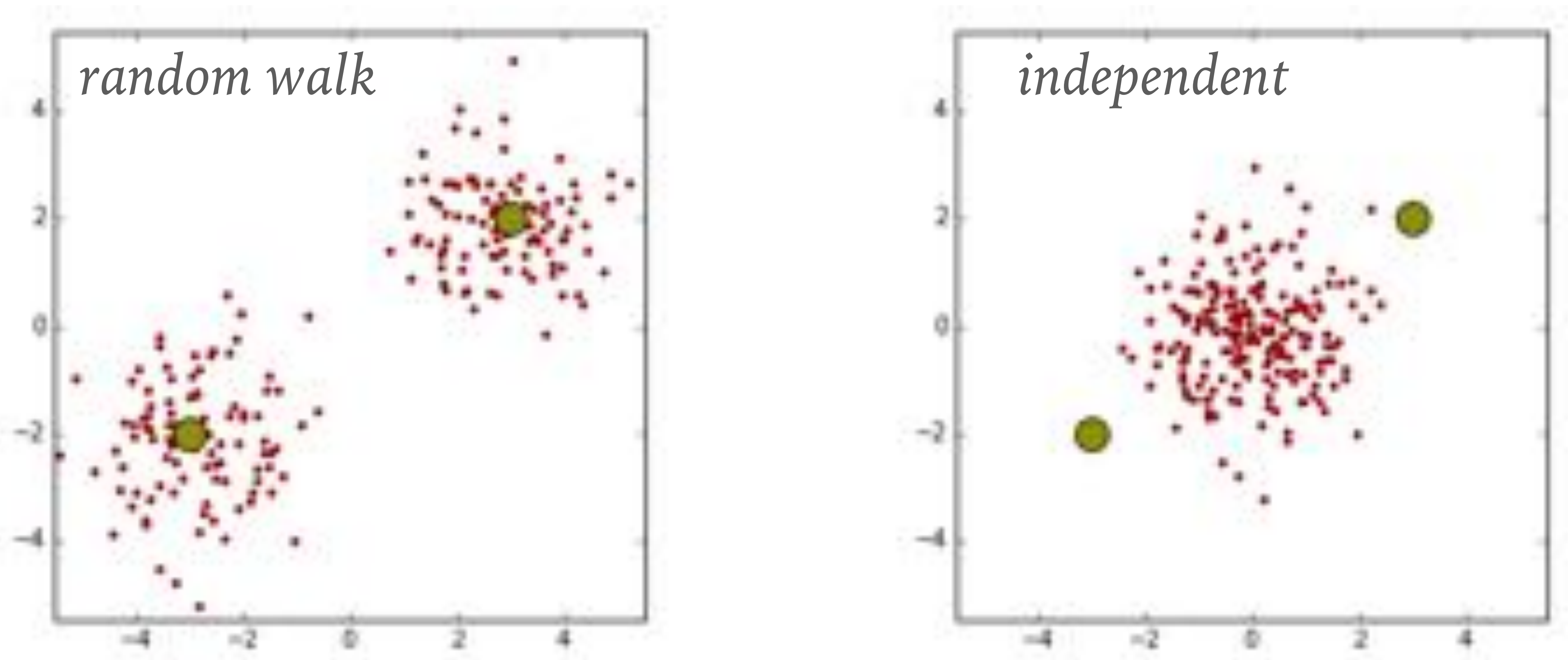
$$P_{acc} = \min(1,\, r(\theta_n,\, \theta_{n+1}))$$

- If the proposed update is not accepted, keep the original value, $\theta_{n+1} = \theta_n$, with probability 1 - P_{acc}.
  - The generated chain will normally have some repeats!

METROPOLIS-HASTINGS UPDATES

```
ratio = p(new) / p(old) * q(old, new) / q(new, old)
accept_prob = min(1, ratio)
```

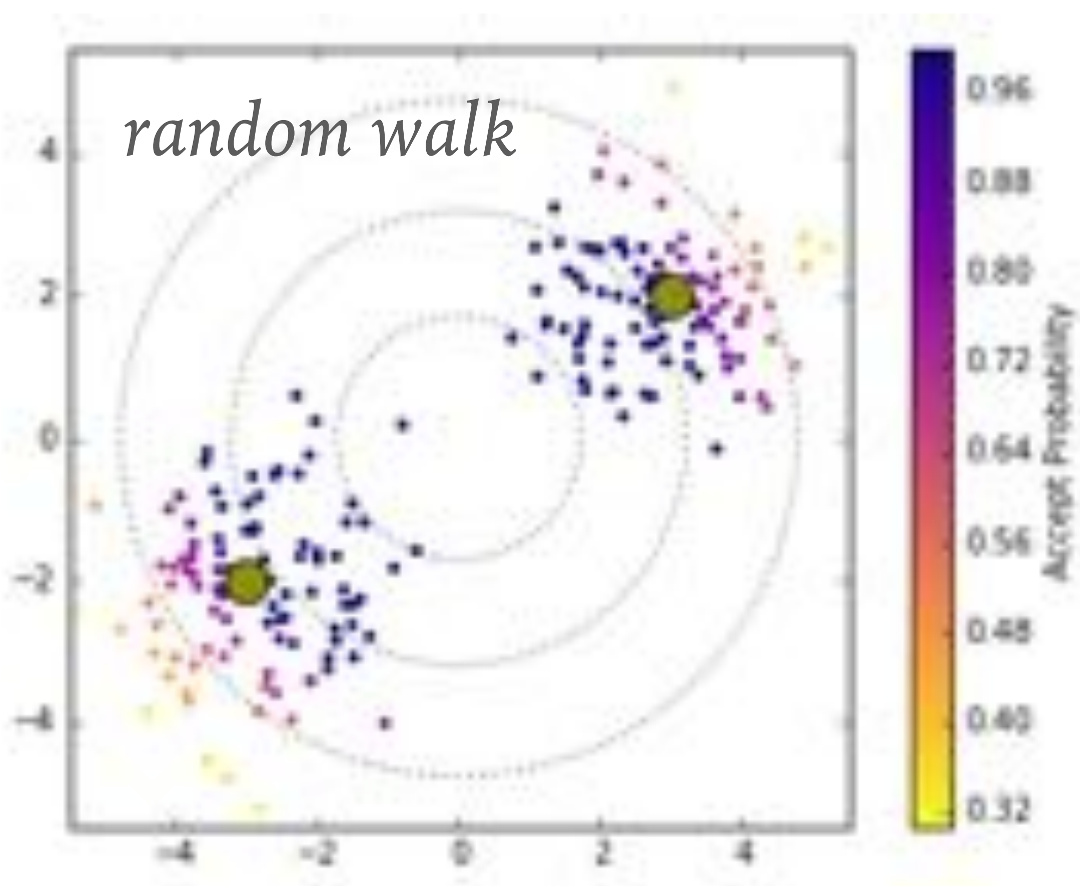
Which points will be favored for these proposal distributions?

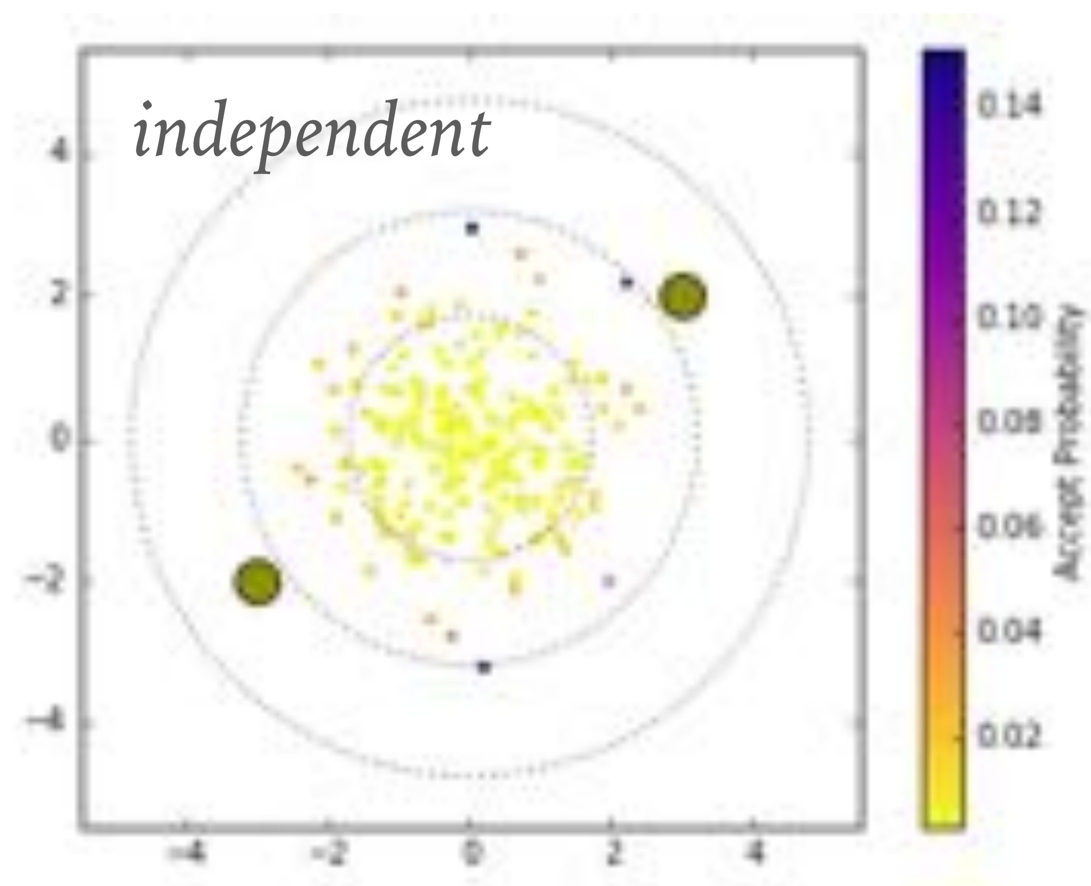


METROPOLIS-HASTINGS UPDATES

```
ratio = p(new) / p(old) * q(old, new) / q(new, old)
accept_prob = min(1, ratio)
```

Which points will be favored for these proposal distributions?





METROPOLIS UPDATES

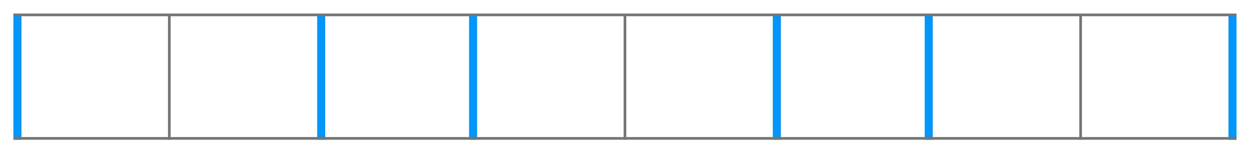
- MH updates reduce to Metropolis updates when the proposal function is reversible, $q(\theta_n | \theta_{n+1}) = q(\theta_{n+1} | \theta_n)$:

$$r(\theta_n, \theta_{n+1}) = \frac{p(\theta_{n+1})\; q(\theta_n | \theta_{n+1})}{p(\theta_n)\; q(\theta_{n+1} | \theta_n)} = \frac{p(\theta_{n+1})}{p(\theta_n)}$$

- Recap of Metropolis-Hastings requirements:
  - Can evaluate target probability density $p(\theta_n)$ up to a constant.
  - Can evaluate proposal density $q(\theta_{n+1} | \theta_n)$ (up to a constant).
  - Can sample from proposal density, $\theta_n \rightarrow \theta_{n+1}$.

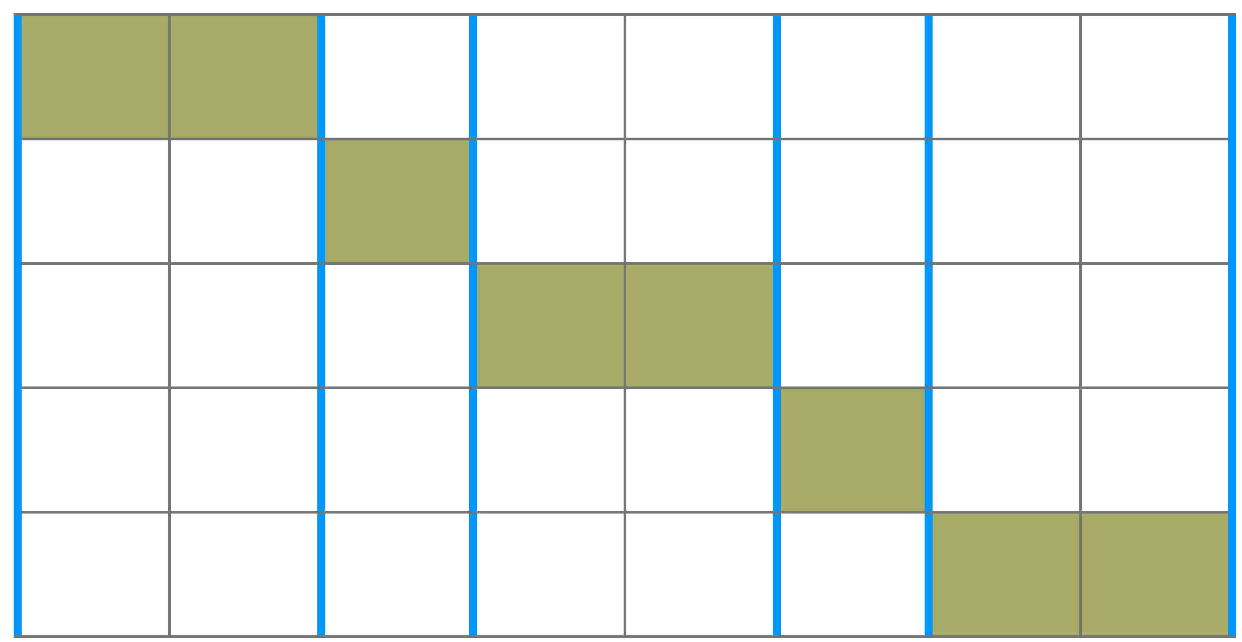
GIBBS UPDATES

1. Decompose state space into subspaces:



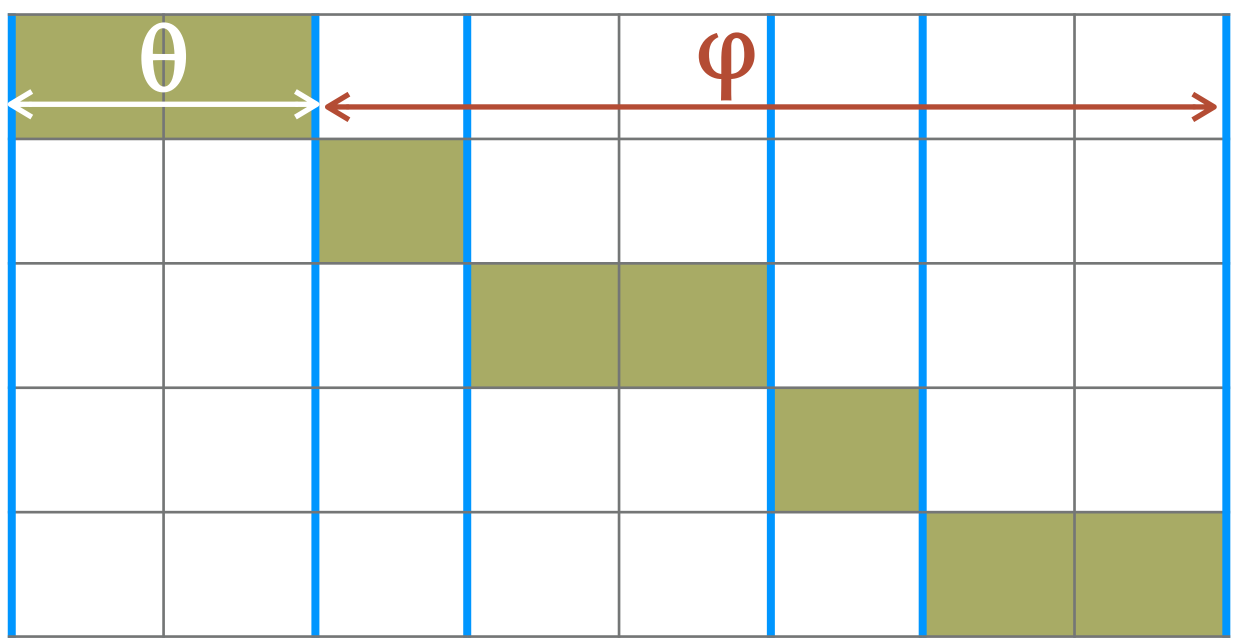
GIBBS UPDATES

2. Update variables of each subspace in consecutive steps:



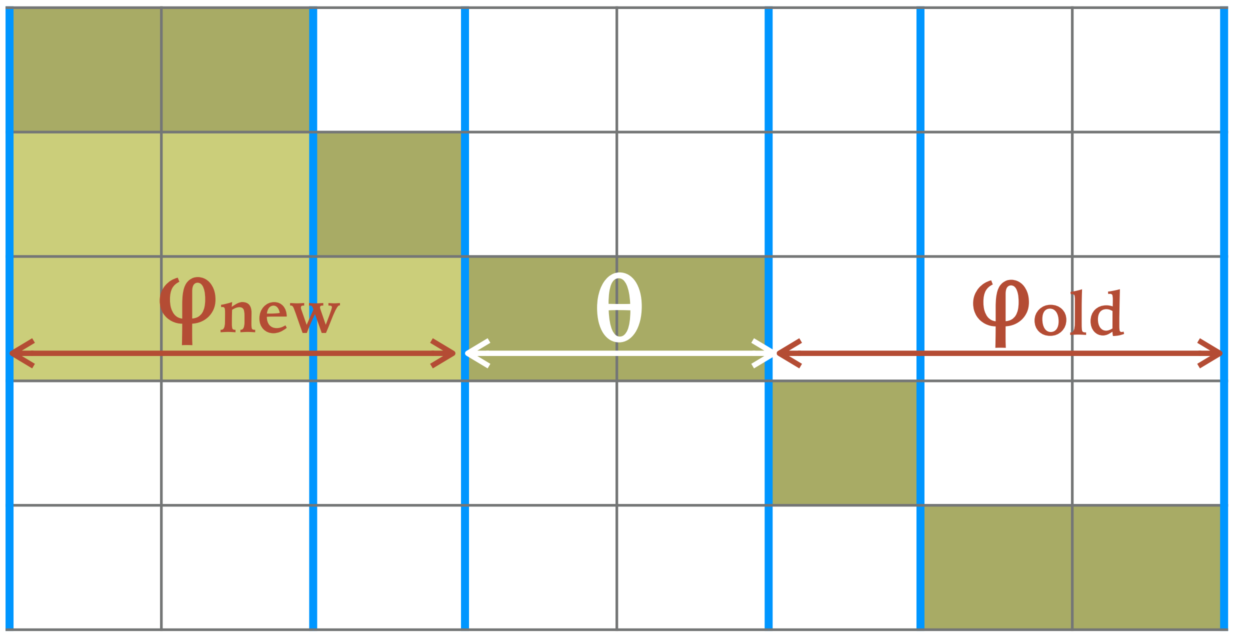
GIBBS UPDATES

3. Sample updated variables from the conditional probability P(θ | φ)



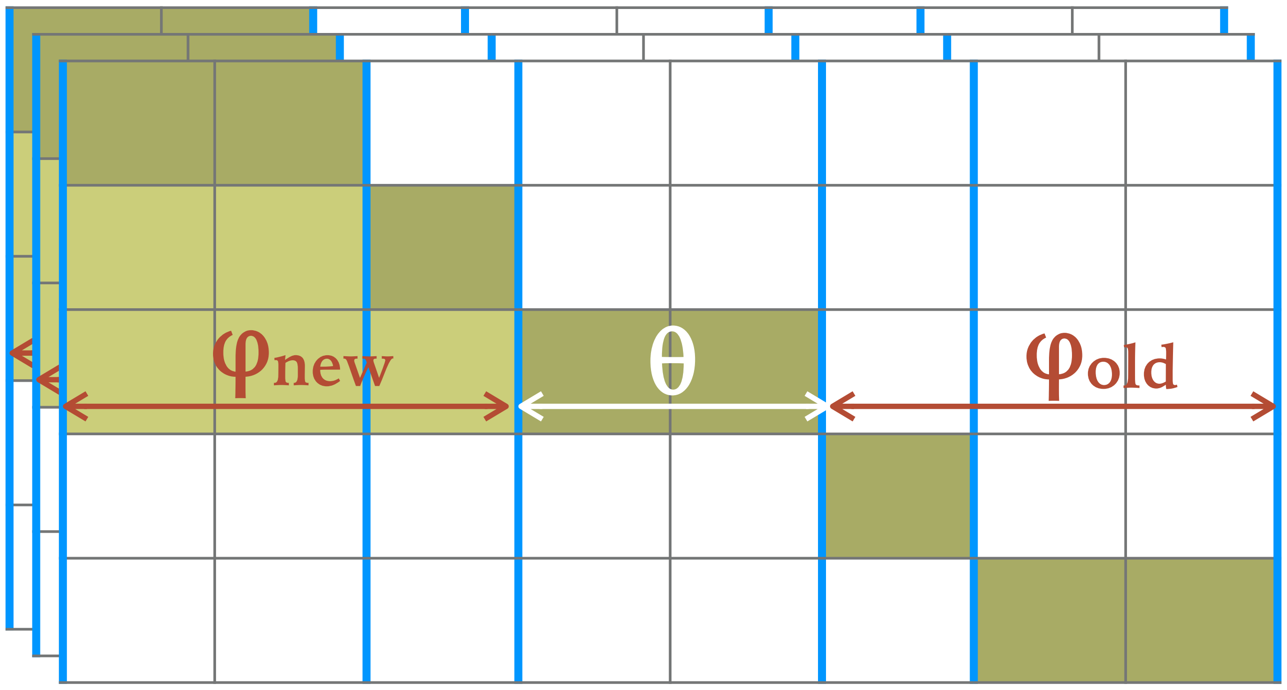
GIBBS UPDATES

4. Condition on new values of earlier subsets P(θ | φ_{new} , φ_{old})



GIBBS UPDATES

5. Start over and repeat this cycle.



GIBBS UPDATES

- Special case of Metropolis Hastings.
- Acceptance probability is always one, by construction.
- Requires that conditional probabilities can be sampled.
- Freedom to choose whatever subsets make this easiest most efficient.

ENSEMBLE SAMPLERS

- Based on the paper:
  - Goodman & Weare, Ensemble Samplers with Affine Invariance
- Implemented in emcee
- Ensemble: many"walkers" simultaneously generating correlated Markov chains.
- Affine invariance: efficiency not affected by any linear (aka "affine") transformation of the parameter space.

ENSEMBLE SAMPLERS

- Each walker performs Metropolis-Hastings updates but using a proposal distribution that depends on the current positions of all other walkers.
- Straightforward to parallelize
- Does not require derivatives

HAMILTONIAN MC

- Relies on a nifty physics analogy.
- "Recall" that the equations of motion for a system with Hamiltonian H are:

$$\frac{dq_i}{dt} = +\frac{\partial H}{\partial p_i} \quad , \quad \frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i}$$

- q_i and p_i are the position and momentum of particle i.
- Can often split H into kinetic and potential terms:

$$H(q,p) = U(q) + K(p)$$

$$K(p) = \sum_i \frac{p_i^2}{2m_i}$$

HAMILTONIAN MC

- This leads to simpler eqns. of motion:

$$\frac{dq_i}{dt} = \frac{p_i}{m_i} \quad , \quad \frac{dp_i}{dt} = -\frac{\partial U}{\partial q_i}$$

- We turn Hamiltonian dynamics into a Markov chain by:
  - Identify positions *q* with the parameters we wish to sample.
  - Create new parameters *p* for the corresponding momenta . We will treat these as nuisance parameters, but this doesn't look promising since we just doubled the dimension of our sampling space!
  - Pick a random starting point then follow its evolution according to Hamiltonian dynamics for some fixed time.

HAMILTONIAN MC

- Each time we repeat the last step, we add a new point to the generated chain.
- Total energy is conserved (by construction) so the distribution of the resulting values is given by the canonical distribution from statistical mechanics:

$$\mathrm{prob}(q) \propto \exp\left(-\frac{U(q)}{kT}\right)$$

- Therefore, pick a "potential energy" to recover the probability distribution P(q) that we actually want (q ~ parameters):

$$U(q) = -\log P(q)$$

HAMILTONIAN MC

- In practice, you can usually set the temperature and all masses equal to 1 and this works surprisingly well!
- The disadvantage is that the method is relatively complex to implement (so let someone else do the work for you!)
- It also requires that you can write a function to evaluate partial derivatives of your logP with respect to each parameter, which is not always feasible.
- Tensorflow and Theano are two (complex) packages that can perform automatic differentiation.

USEFUL LINKS

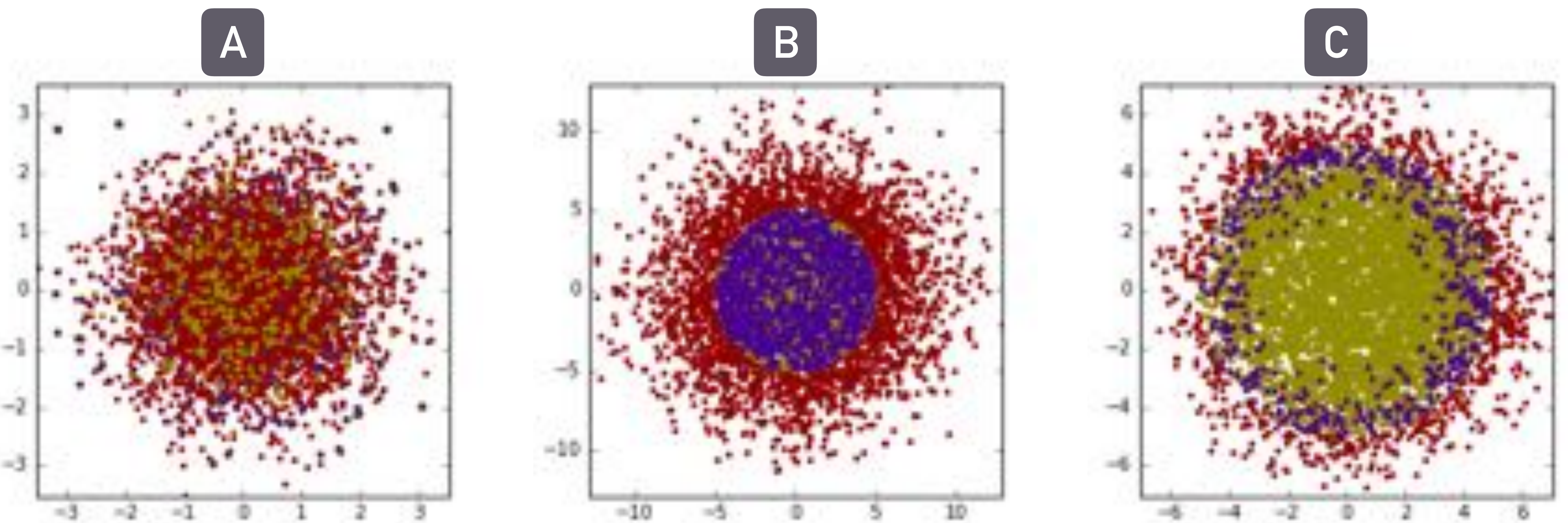
- Visualizations of sampling:
  - MH, Gibbs, NUTS in pymc3
  - pymc3 vs emcee
  - Hamiltonian MC
- Comparison of different packages (emcee, pymc2, pystan)

MCMC METRICS

- Efficiency: what fraction of proposals are accepted?
- Coverage: is the target density fully explored?
- Correlation:
  - how correlated is each sample with previous samples?
  - how independent are the generated samples of the initial starting point?
- Prefer coverage and low correlations over efficiency!

ACTIVITY: VISUAL DIAGNOSTICS

- ➤ Target density is a top-hat (disk) of radius 5.
- ➤ Which samplers are: most efficient? have good coverage?
- ➤ Which samplers use “random-walk” updates?



proposed and accepted | *proposed but not accepted* | *repeated*

PRACTICAL ADVICE

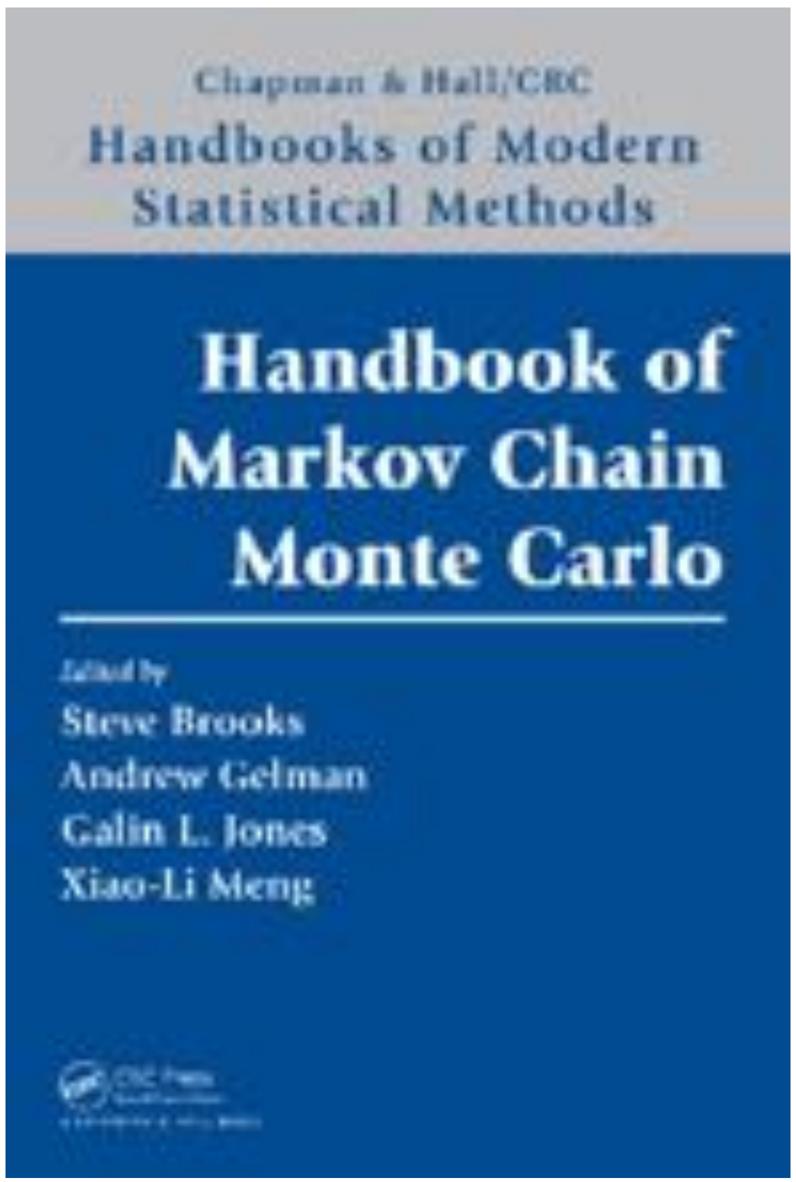
- How long should the chain be?
  - Large multiple of whatever sequence length is necessary for autocorrelations to drop near zero.
  - One long chain is safer than many short chains if you aren't sure.
- Should I remove the initial "burn-in" samples?
  - Mostly harmless, but un-necessary.
  - Instead, start with a value you don't mind including in your generated samples (also, MCMC is a terrible optimizer).
- Should I "thin" the generated samples to reduce correlations?
  - No! You can never get a better answer by throwing away information.

PRACTICAL ADVICE

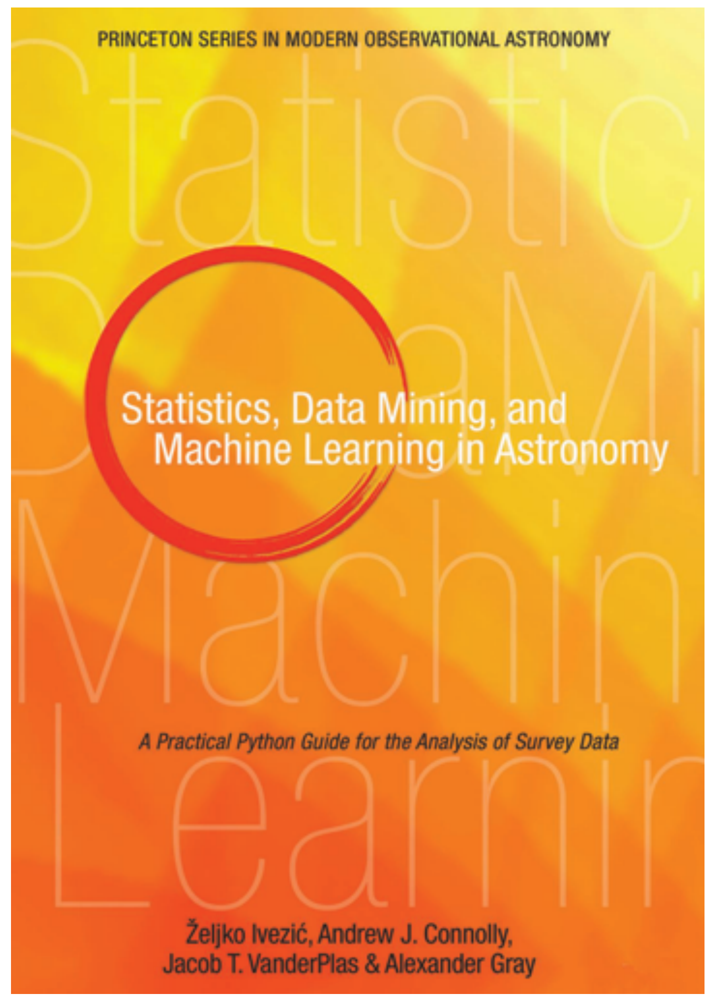
- Which updating algorithm should I use?
  - random walks are robust but inefficient.
  - fancier algorithms suppress random walk behavior to improve efficiency but are generally more fragile.
  - no “best algorithm”: need to benchmark your problem.
- Which MCMC package should I use?
  - Start with emcee.
  - Try pymc3 if you need a fancier updating algorithm (e.g., Hamiltonian).
  - Don’t write your own, except to learn how they work.
- Its still too slow: what do I do?
  - Try variational methods to obtain exact results for an approx. posterior.

REFERENCES

Chapter 1
Chapter 5



Section 5.8



Chapters 29-30