Lecture 8:

Monte Carlo Methods

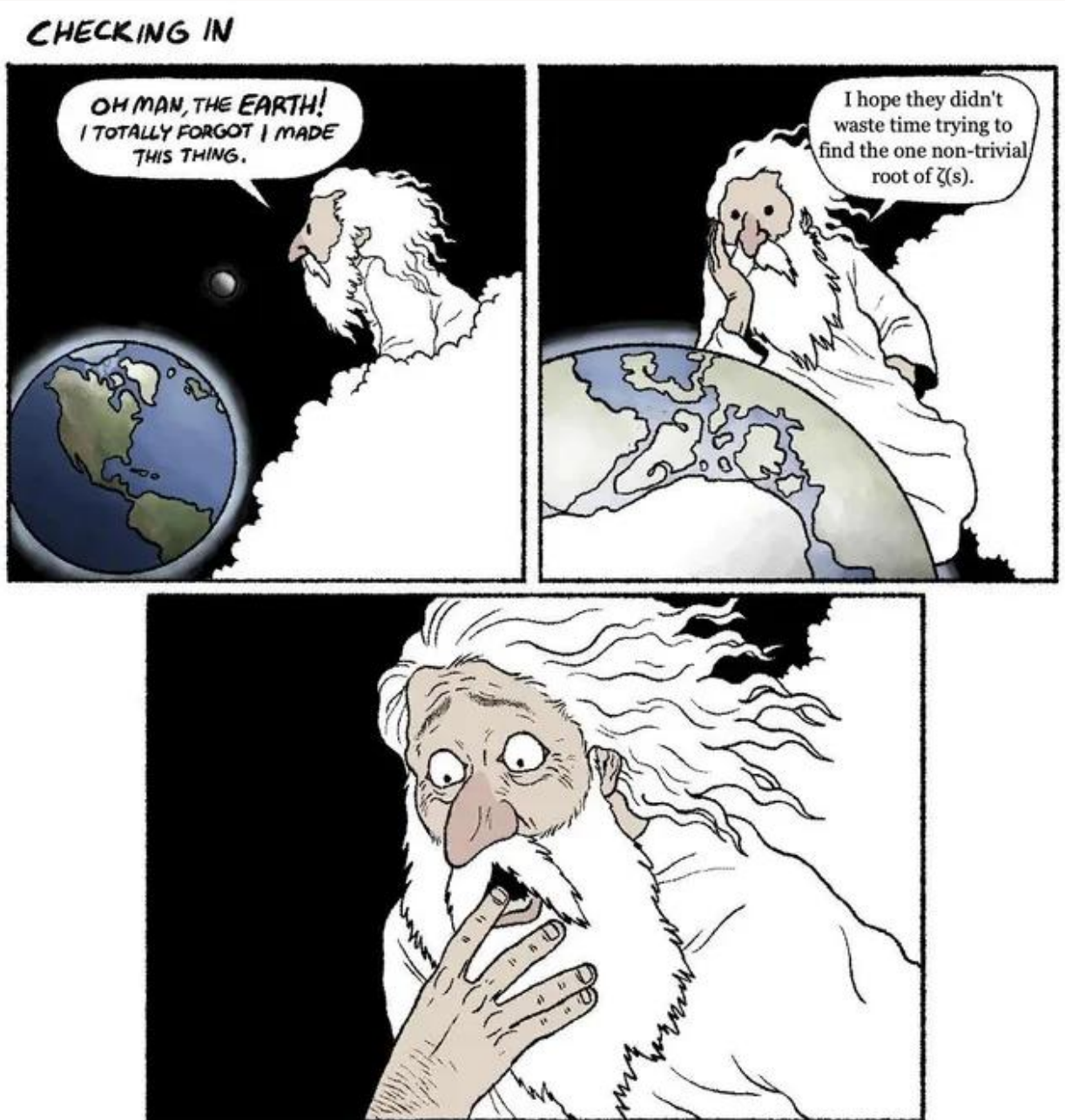
Markus Hohle

University California, Berkeley

Bayesian Data Analysis and Machine Learning for Physical Sciences

Course Map

Module 1	Maximum Entropy and Information, Bayes Theorem
Module 2	Naive Bayes, Bayesian Parameter Estimation, MAP
Module 3	MLE, Lin Regression
Module 4	Model selection I: Comparing Distributions
Module 5	Model Selection II: Bayesian Signal Detection
Module 6	Variational Bayes, Expectation Maximization
Module 7	Hidden Markov Models, Stochastic Processes
Module 8	**Monte Carlo Methods**
Module 9	Machine Learning Overview, Supervised Methods
Module 10	Unsupervised Methods
Module 11	ANN: Perceptron, Backpropagation
Module 12	ANN: Basic Architecture, Regression vs Classification, Backpropagation again
Module 13	Convolution and Image Classification and Segmentation
Module 14	Graphs and GNNs
Module 15	RNNs and LSTMs
Module 16	Transformer and LLMs



Outline

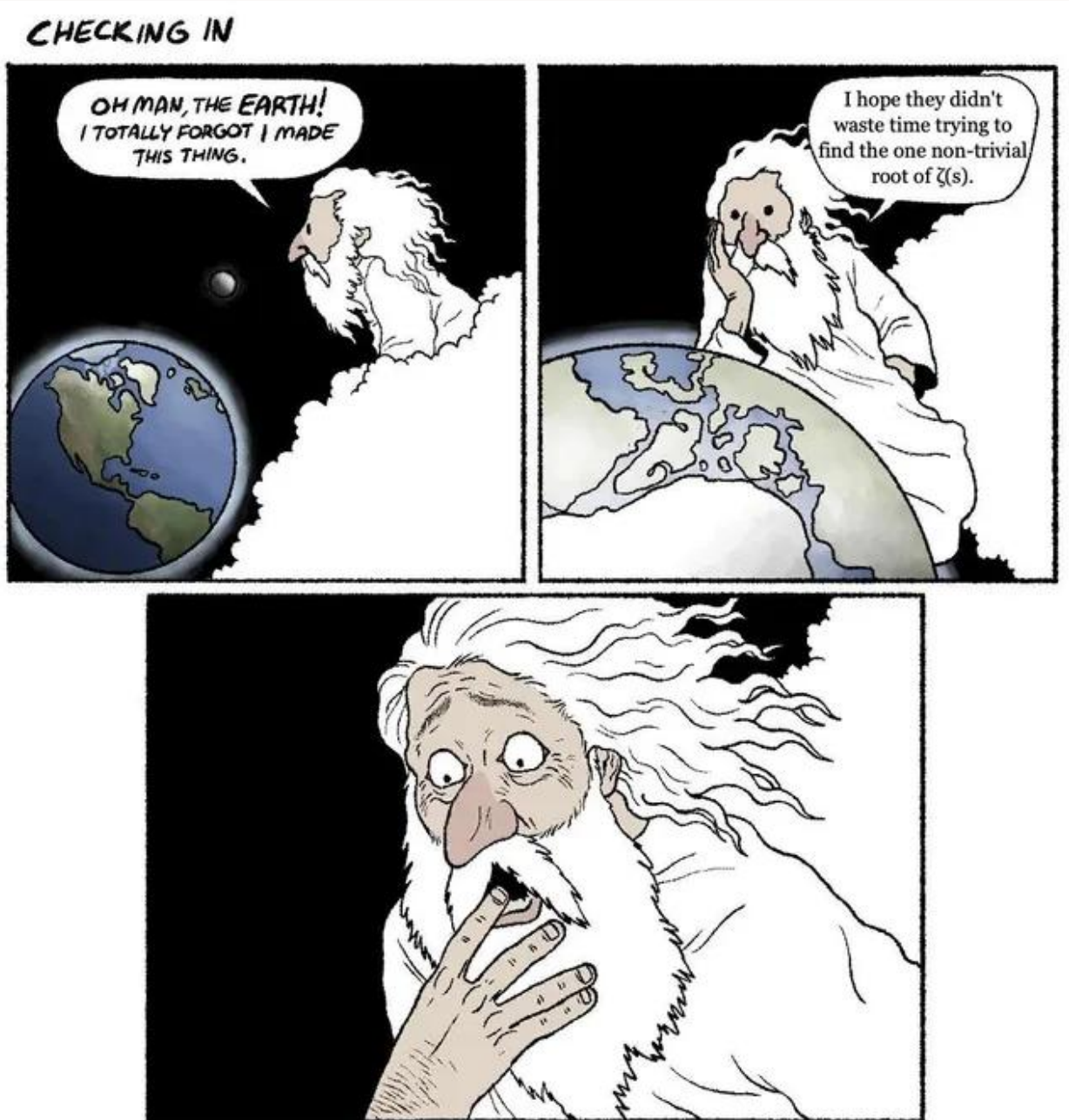
Basic Idea & Finding Pi

Mapping Distributions & Gibbs Sampling

Gillespie Algorithm

Metropolis (- Hastings) Algorithm

Bootstrapping



Outline

Basic Idea & Finding Pi

Mapping Distributions & Gibbs Sampling

Gillespie Algorithm

Metropolis (- Hastings) Algorithm

Bootstrapping

Monte Carlo Methods

idea: generating a set of values randomly → **M**onte **C**arlo method
i.e. repeated random sampling

pros: for many sample repetitions → the actual probability density function emerges
pretty **simple** set up & easy to implement, **easy to parallelize**

cons: not directed like e.g. gradient descent (see later)

applications: - numerical evaluation of complicated integrals

$$\rho = \frac{P(\mathrm{M_A}|D,I)}{P(\mathrm{M_B}|D,I)} = \frac{P(M_A)}{P(M_B)} \cdot \frac{\int P(D|\{\alpha\}_A, M_A, I)\, d\alpha_{Aj}}{\int P(D|\{\alpha\}_B, M_B, I)\, d\alpha_{Bj}} \cdot \frac{\prod_j \boldsymbol{\alpha_{jB}}(max) - \boldsymbol{\alpha_{jB}}(min)}{\prod_j \boldsymbol{\alpha_{jA}}(max) - \boldsymbol{\alpha_{jA}}(min)}$$

- estimating posteriors: $P(q|D) = \frac{\binom{n}{k} q^k (1-q)^{n-k}}{P(D)} P(q) \sim q^k (1-q)^{n-k} P(q)$

$$q_i(Z_i|D) = \frac{1}{Z} \exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right) \right\rangle_{\{j\neq i\}} \right)$$

applications: - numerical evaluation of complicated integrals

$$\rho = \frac{P(\mathrm{M_A}|D,I)}{P(\mathrm{M_B}|D,I)} = \frac{P(M_A)}{P(M_B)} \cdot \frac{\int P(D|\{\alpha\}_A, M_A, I)\, d\alpha_{Aj}}{\int P(D|\{\alpha\}_B, M_B, I)\, d\alpha_{Bj}} \cdot \frac{\prod_j \boldsymbol{\alpha_{jB}}(max) - \boldsymbol{\alpha_{jB}}(min)}{\prod_j \boldsymbol{\alpha_{jA}}(max) - \boldsymbol{\alpha_{jA}}(min)}$$

- estimating posteriors: $P(q|D) = \frac{\binom{n}{k} q^k (1-q)^{n-k}}{P(D)} P(q) \sim q^k (1-q)^{n-k} P(q)$

$$q_i(Z_i|D) = \frac{1}{\mathcal{Z}} \exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right)\right\rangle_{\{j\neq i\}}\right)$$

- modelling stochastic processes: Gillespie, Metropolis

- optimization: simulated annealing

- bootstrapping: estimating confidence intervals, random forest

let's start simple: $A_{circ} = \pi r^2 = \pi$ for $r = 1$

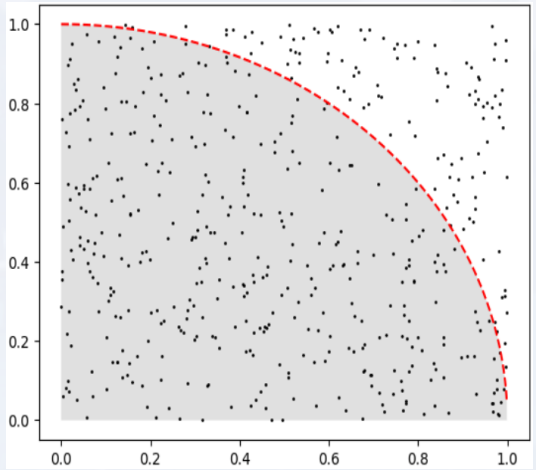
example: finding π via Monte Carlo

$$A_{square} = 1$$

$$A_{section} = \frac{\pi}{4}$$

$$\pi = 4\frac{A_{section}}{A_{square}}$$

picking N_{tot} random values [0,1] x [0,1]

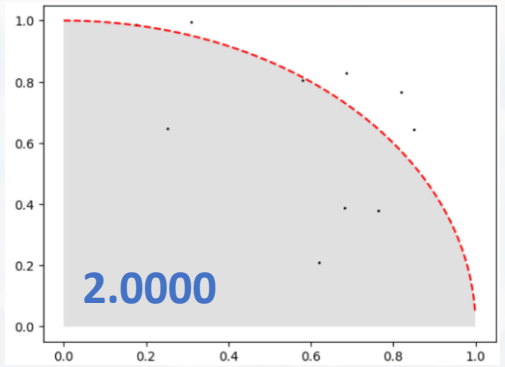


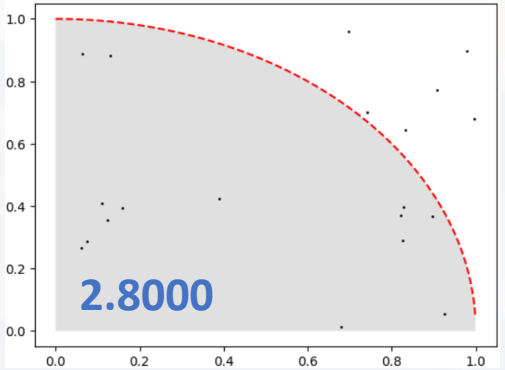
$$\pi \approx 4\frac{N_{section}}{N_{tot}}$$

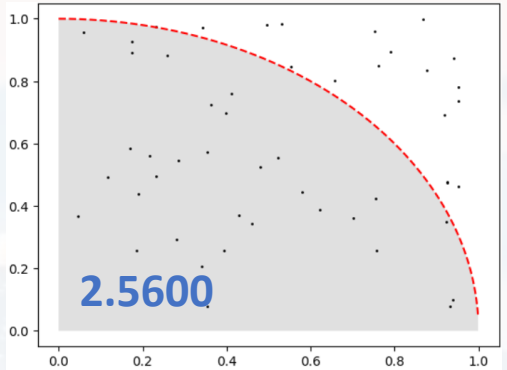
Monte Carlo Methods

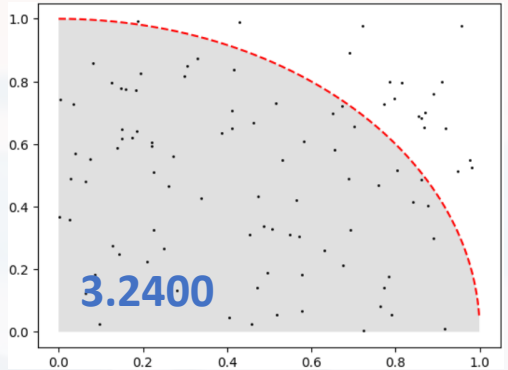
idea: generating a set of values randomly
i.e. repeated random sampling

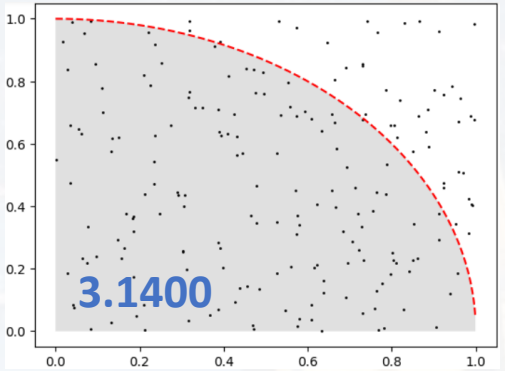
→ **M**onte **C**arlo method

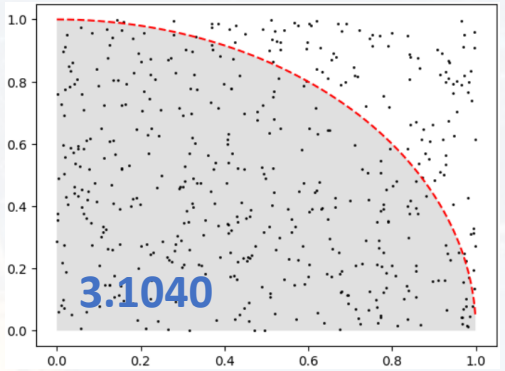


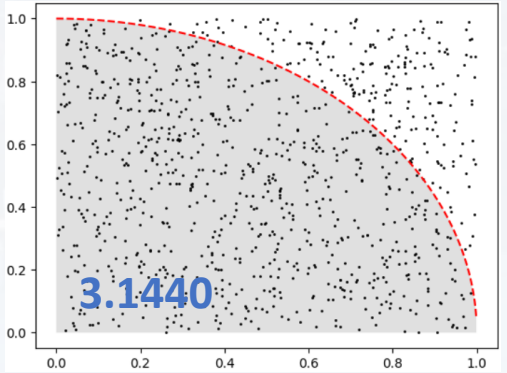


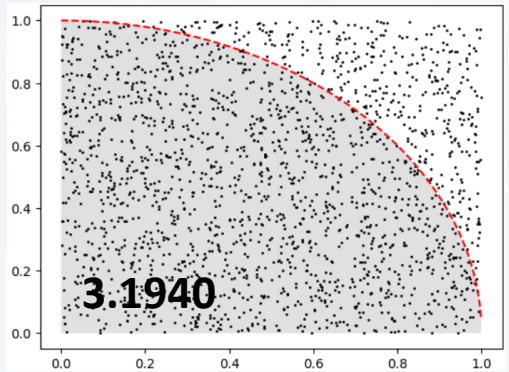


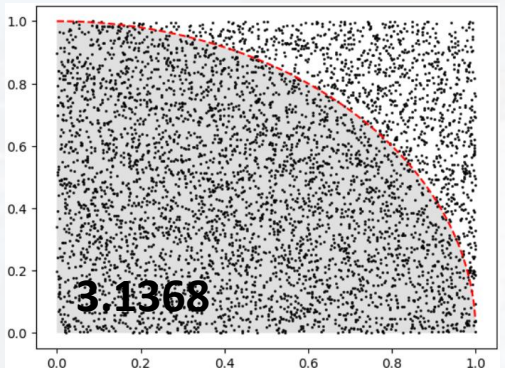


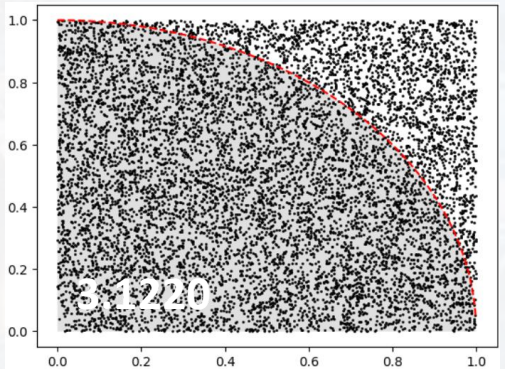


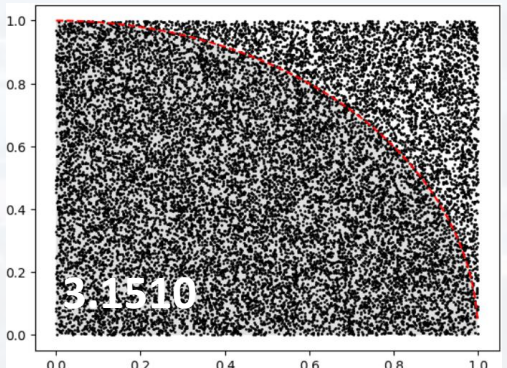








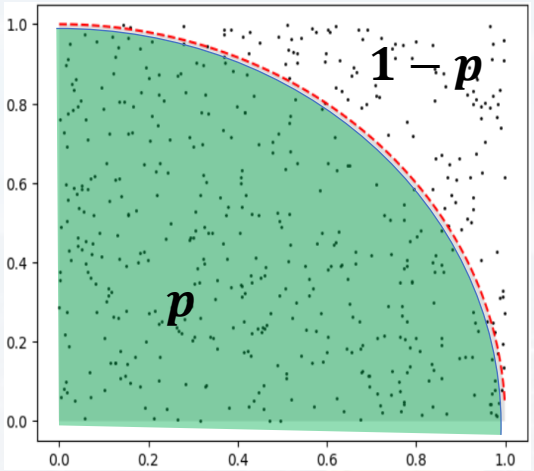




$\pi = 3.14159 \ldots$

example: finding π via Monte Carlo

How does the accuracy of π depend on N_{tot}?



$$\pi \approx 4 \frac{N_{section}}{N_{tot}}$$

We know that a point within the section is drawn with the probability p

That is a **binomial problem**!
In practice, k points fall into the section with probability p and $N_{tot} - k$ don't, with a probability of $1 - p$

Thus, we can tell the mean and the variance of **one simulation for one specific N_{tot}**

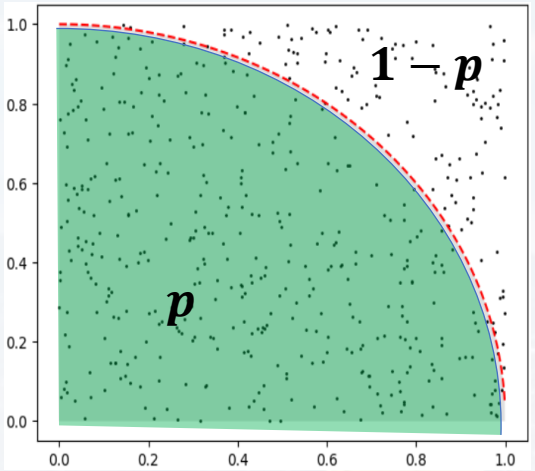
$$\sigma(k)^2 = N_{tot}\, p(1-p)$$

$$\mu(k) = pN_{tot}$$

(see Physics 77/88)

example: finding π via Monte Carlo

How does the accuracy of π depend on N_{tot}?

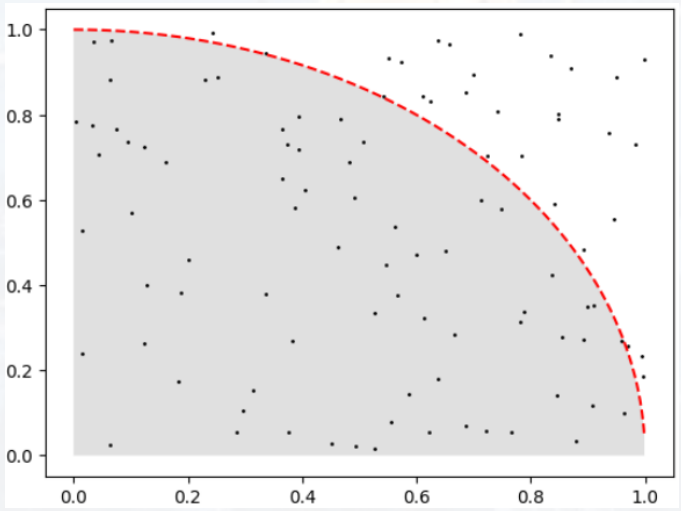


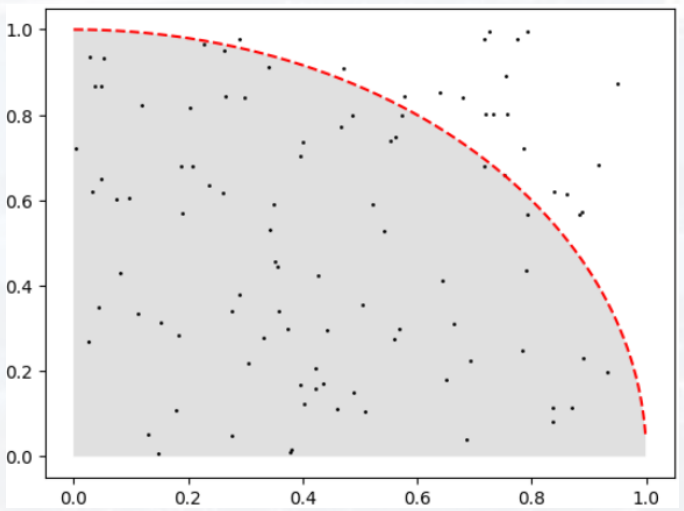
$$\pi \approx 4\frac{N_{section}}{N_{tot}} \qquad \sigma(k)^2 = N_{tot}\, p(1-p) \qquad \mu(k) = pN_{tot}$$

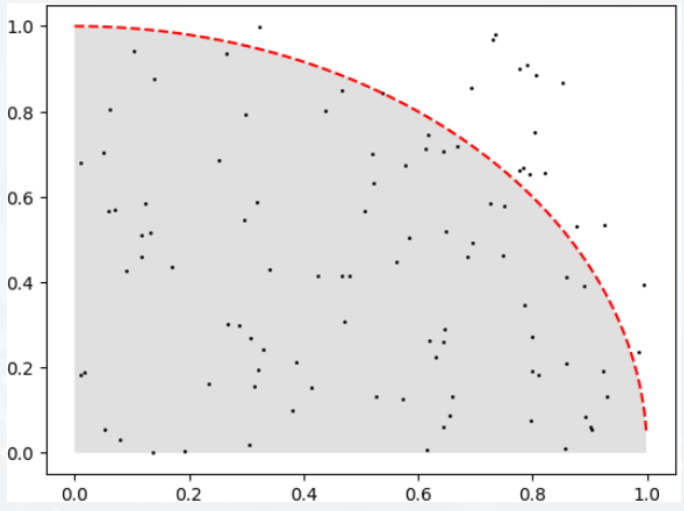
error of π: $4\sigma\left(\frac{N_{section}}{N_{tot}}\right) = 4\sigma\left(\frac{k}{N_{tot}}\right)$ **standard deviation σ of the ratio** $\frac{N_{section}}{N_{tot}}$

Say we run the simulation for a **specific N_{tot}** many times → $Var(k)$

$N_{tot} = 100$

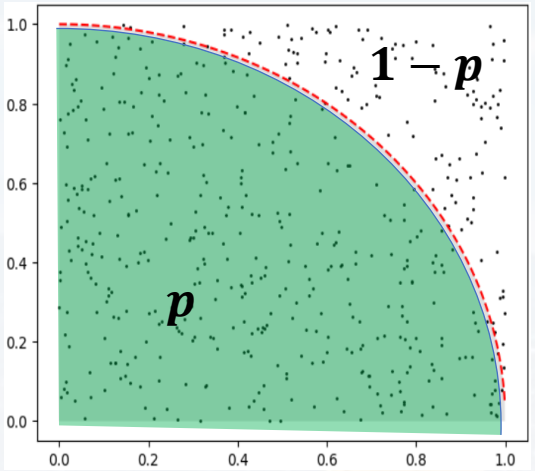






example: finding π via Monte Carlo

How does the accuracy of π depend on N_{tot}?



$$\pi \approx 4\frac{N_{section}}{N_{tot}}$$

$$\boxed{\sigma(k)^2 = N_{tot}\, p(1-p)} \qquad \mu(k) = pN_{tot}$$

error of π: $4\sigma\left(\frac{N_{section}}{N_{tot}}\right) = 4\sigma\left(\frac{k}{N_{tot}}\right)$

Say we run the simulation for a **specific N_{tot}** many times → $Var(k)$

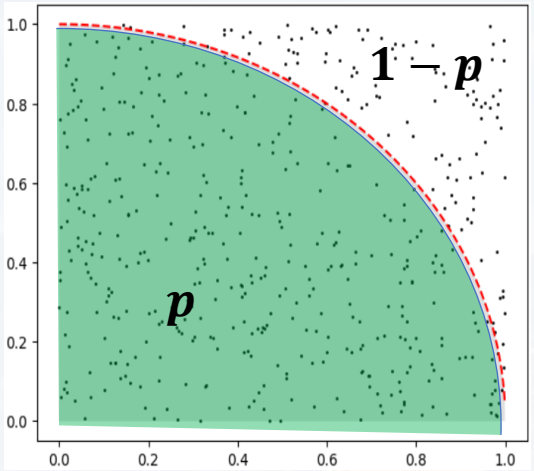
error of π: $4\sigma\left(\frac{N_{section}}{N_{tot}}\right) = 4\sigma\left(\frac{k}{N_{tot}}\right) = 4\sqrt{Var\left(\frac{k}{N_{tot}}\right)} = 4\sqrt{\frac{1}{{N_{tot}}^2}Var(k)}$ $\boxed{= 4\sqrt{\frac{1}{{N_{tot}}^2}N_{tot}\, p(1-p)}}$

We know: $Var(x) = \langle x^2\rangle - \langle x\rangle^2$

a = const $x \to ax$ $\quad Var(ax) = \langle (ax)^2\rangle - \langle ax\rangle^2 = a^2(\langle x^2\rangle - \langle x\rangle^2) = a^2 Var(x)$

example: finding π via Monte Carlo

How does the accuracy of π depend on N_{tot}?



$$\pi \approx 4\frac{N_{section}}{N_{tot}}$$

error of π: $4\sigma\left(\frac{N_{section}}{N_{tot}}\right) = 4\sqrt{\frac{1}{{N_{tot}}^2} N_{tot}\, p(1-p)} = 4\sqrt{\frac{1}{N_{tot}}\, p(1-p)}$

of course we **don't know** $p = \frac{\pi}{4} = \frac{A_{section}}{A_{square}}$

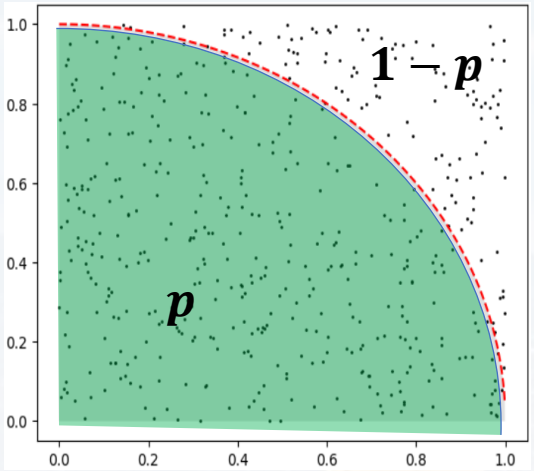
because we wanted find π in the first place

but we can ***estimate*** it during the simulation

$$4\sqrt{\frac{1}{N_{tot}}\, p(1-p)} \approx 4 \cdot 0.4\sqrt{\frac{1}{N_{tot}}}$$

example: finding π via Monte Carlo

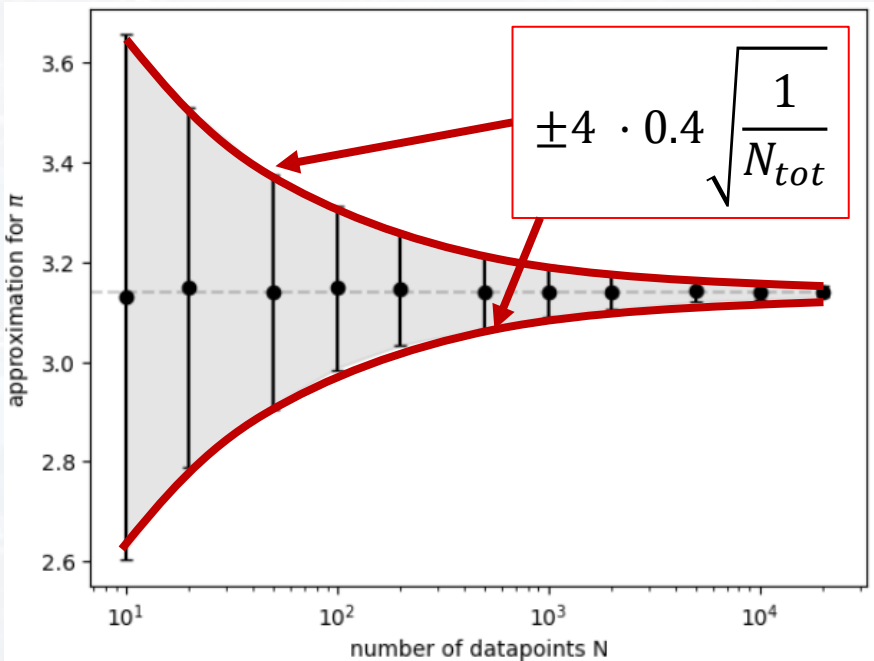
How does the accuracy of π depend on N_{tot}?



$$\pi \approx 4\frac{N_{section}}{N_{tot}}$$

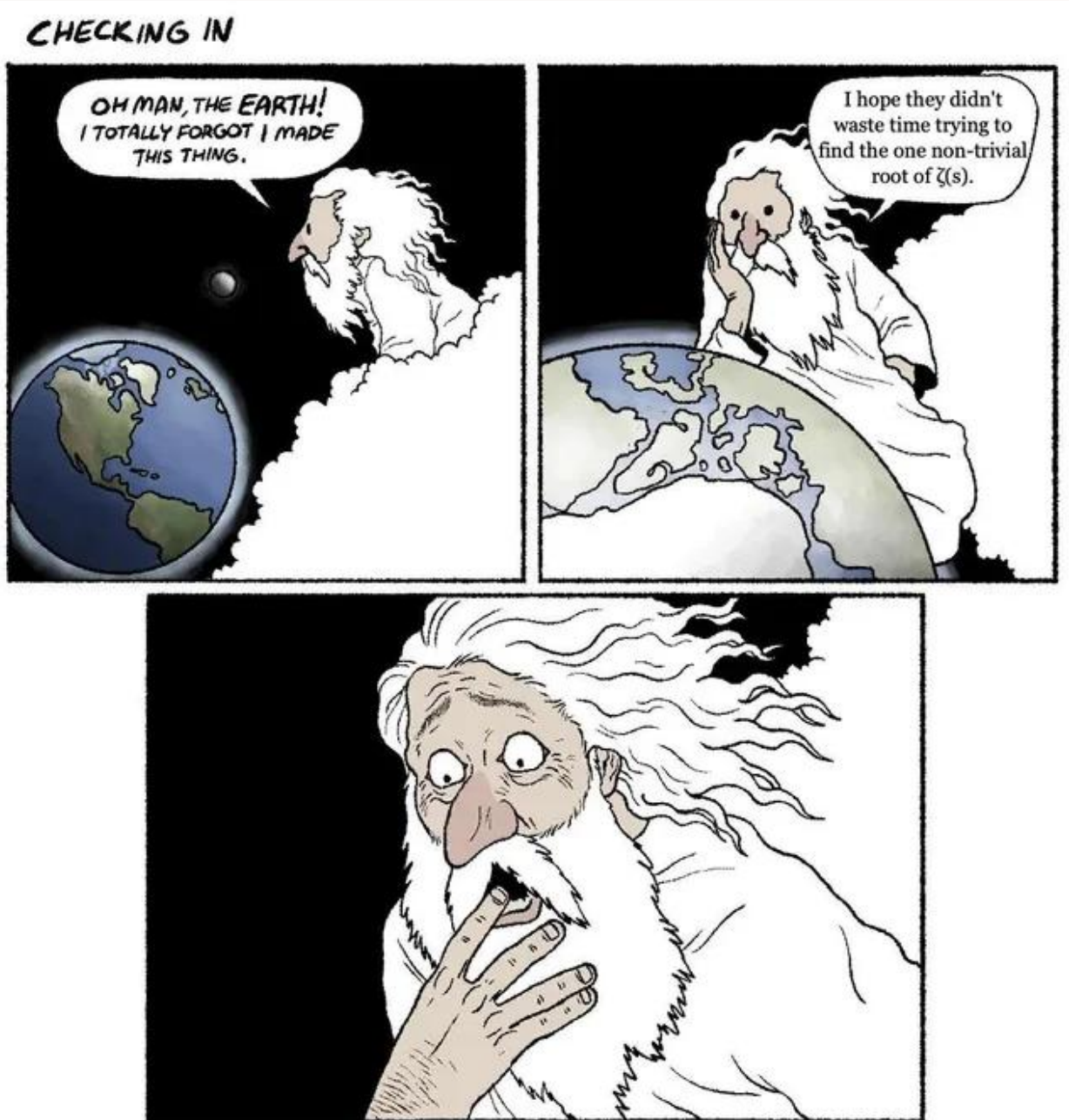
error of π: $4\sigma\left(\frac{N_{section}}{N_{tot}}\right) = 4\sqrt{\frac{1}{N_{tot}}\,p(1-p)} \approx 4 \cdot 0.4\sqrt{\frac{1}{N_{tot}}}$

running 100 simulations **for each** N_{tot}
→ calculating standard deviation of π
→ comparing to $4 \cdot 0.4\sqrt{\frac{1}{N_{tot}}}$



see **`Monte_Carlo_Simulation_PI.ipynb`**

Berkeley UNIVERSITY OF CALIFORNIA Monte Carlo Methods



Outline

Basic Idea & Finding Pi

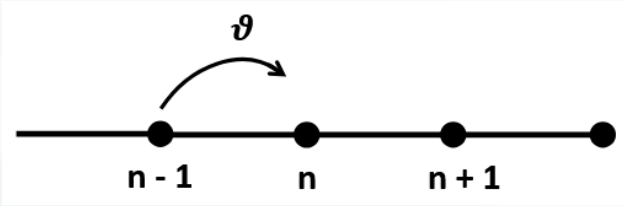
Mapping Distributions & Gibbs Sampling

Gillespie Algorithm

Metropolis (- Hastings) Algorithm

Bootstrapping

last time:

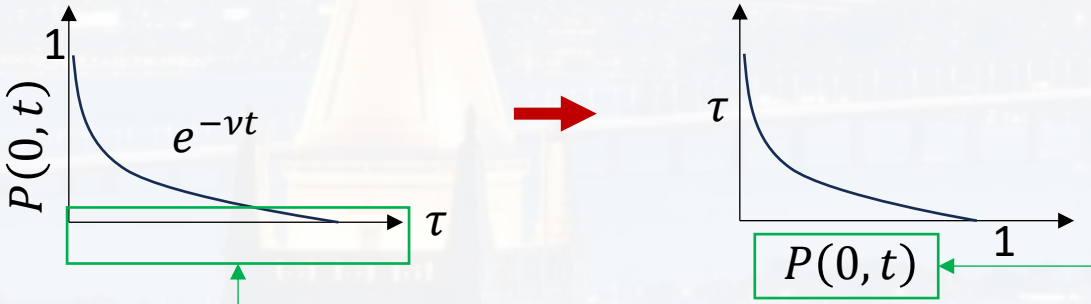


n:	**different states**
ϑ:	**hopping rate ($probability/time$)**

calculating the **waiting time** (time τ *between* two events)

$$P(0,t) = \frac{(\upsilon t)^0}{0!} e^{-\nu t} \qquad \tau = -\frac{1}{\nu} ln[P(0,t)]$$

We want to understand the process a bit better (i. e. find the **waiting time distribution** $w(\tau)$)



random number

$$\rho \in [a,b]$$
$$P(\rho) = const = 1/(b-a)$$

waiting time τ:

$$\tau \in [0,\infty)$$
$$w(\tau) \neq const$$

$$\int_a^b P(\rho)\, d\rho = 1 = \int_0^\infty w(\tau)\, d\tau$$

$$\tau = -\frac{1}{\nu} ln[P(0,t)] = -\frac{1}{\nu} ln[\rho]$$

n:	**different states**
ϑ:	**hopping rate ($probability/time$)**
ρ:	**uniformly dist random number**

$$\int_a^b f(x)\,dx = \int_{y(a)}^{y(b)} f[x(y)]\,\frac{dx}{dy}\,dy \qquad \textbf{(substitution)}$$

$$= \int_{y(a)}^{y(b)} g(y)\,dy \qquad g(y) := f[x(y)]\,\frac{dx}{dy}$$

note: we actually need $\left|\frac{dx}{dy}\right|$

inverse of y: $x(y) = y^{-1}$

our situation:

$$\int_{\rho=0}^{\rho=1} P(\rho)\,d\rho = \int_{\tau(0)}^{\tau(1)} P[\rho(\tau)]\left|\frac{d\rho}{d\tau}\right| d\tau = \int_{\tau(0)}^{\tau(1)} w(\tau)\,d\tau$$

$$w(\tau) = P[\rho(\tau)]\left|\frac{d\rho}{d\tau}\right| \qquad x(y) = \rho(\tau) = e^{-\vartheta\tau}$$

$$w(\tau) = 1 \cdot \vartheta e^{-\vartheta\tau}$$

mean waiting time: $t^* = \int_{\tau=0}^{\tau=\infty} \tau\, w(\tau)\,d\tau = \frac{1}{\vartheta}$

actual Gibbs Sampling:

D	: data set of size K
Z	: set of n (latent) parameter
σ^2	: variance
μ	: mean
$\frac{1}{\sigma^2} = \lambda$	: precision

we know from module 6: $q_i(Z_i|D) = \frac{1}{Z}\exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right)\right\rangle_{\{j\neq i\}}\right)$

which led to circular dependencies

$$q_\mu(\mu|D) \sim \mathcal{N}(\mu|\mu_K, \lambda_K^{-1})$$

where $\mu_K = \frac{\tau_0\,\mu_0 + K\,\bar{x}}{\tau_0 + K}$

$\lambda_K = (\tau_0 + K)\,\langle\tau\rangle_\tau$

$\bar{x} = \frac{1}{K}\sum_{k=1}^{K} x_k$

$$q_\tau(\tau|D) \sim \Gamma(\tau|a_K, b_K)$$

where $a_K = a + \frac{K+1}{2}$

$b_K = b + \frac{1}{2}\langle\sum_k (x_k - \mu)^2 + \tau_0(\mu - \mu_0)^2\rangle_\mu$

$$\langle\tau\rangle_\tau = \int \tau\, q_\tau(\tau|D)\, d\tau = \frac{a_K}{b_K}$$

set τ_0, μ_0, a and b to **small positive values** (largest ignorance)

actual Gibbs Sampling:

we know from module 6: $q_i(Z_i|D) = \frac{1}{\mathcal{Z}} \exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right)\right\rangle_{\{j\neq i\}}\right)$

D	: data set of size K
Z	: set of n (latent) parameter
σ^2	: variance
μ	: mean
$\frac{1}{\sigma^2} = \lambda$	: precision

$q_i\left(Z_i \middle| \{Z_{j\neq i}\}\right)$ sampling for all i in a particular order

say we have **three** parameters - randomly (or MLE guess from data) initialize Z_1, Z_2, Z_3

iteration t

draw Z_1 $q(Z_1(t+1)|Z_2(t), Z_3(t))$

draw Z_2 $q(Z_2(t+1)|Z_1(t+1), Z_3(t))$

draw Z_3 $q(Z_3(t+1)|Z_1(t+1), Z_2(t+1))$

iteration $t+2$

actual Gibbs Sampling:

we know from module 6: $q_i(Z_i|D) = \frac{1}{\mathcal{Z}} \exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right)\right\rangle_{\{j\neq i\}}\right)$

$q_i\left(Z_i \middle| \{Z_{j\neq i}\}\right)$ sampling for all i in a particular order

D	: data set of size K
Z	: set of n (latent) parameter
σ^2	: variance
μ	: mean
$\frac{1}{\sigma^2} = \lambda$	: precision

for the example from module 6 - now: randomly (or MLE guess from data) initialize μ, λ

$D|\mu, \lambda \sim \mathcal{N}(\mu, \lambda^{-1})$

$\mu \sim \mathcal{N}(\mu_0, \lambda_0^{-1})$

$\lambda \sim \Gamma(a_0, b_0)$

instead for calculating the means, we now sample:

$\mu|\lambda, D \sim \mathcal{N}(M_\lambda, L_\lambda^{-1})$ $\quad L_\lambda = (\lambda_0 + K)\lambda \quad M_\lambda = \frac{\mu_0\lambda_0 + \sum_{k=1}^{K} x_k}{\lambda_0 + K}$

$\lambda|\mu, D \sim \Gamma(a_K, b_K)$ $\quad a_K = a_0 + \frac{K}{2} \quad b_K = b_0 + \frac{1}{2}\sum_{k=1}^{K}(x_k - \mu)^2$

iteration $t \rightarrow t + 1$

actual Gibbs Sampling:

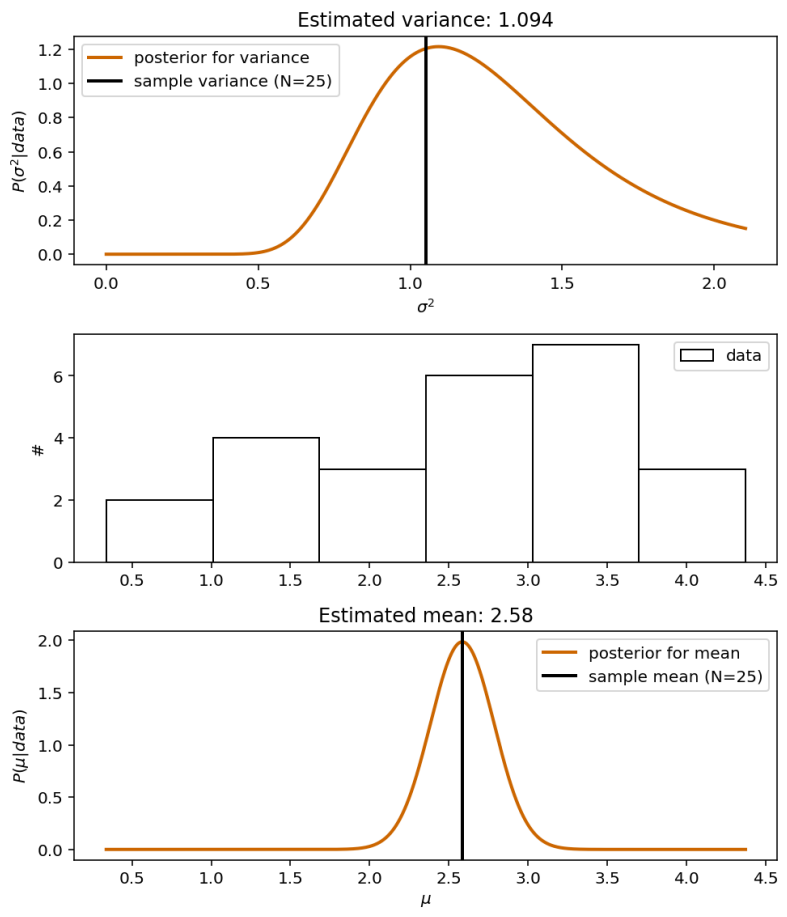
we know from module 6: $q_i(Z_i|D) = \frac{1}{Z}\exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right)\right\rangle_{\{j\neq i\}}\right)$

$q_i\left(Z_i\middle|\{Z_{j\neq i}\}\right)$ sampling for all i in a particular order

D	: data set of size K
Z	: set of n (latent) parameter
σ^2	: variance
μ	: mean
$\frac{1}{\sigma^2} = \lambda$	: precision

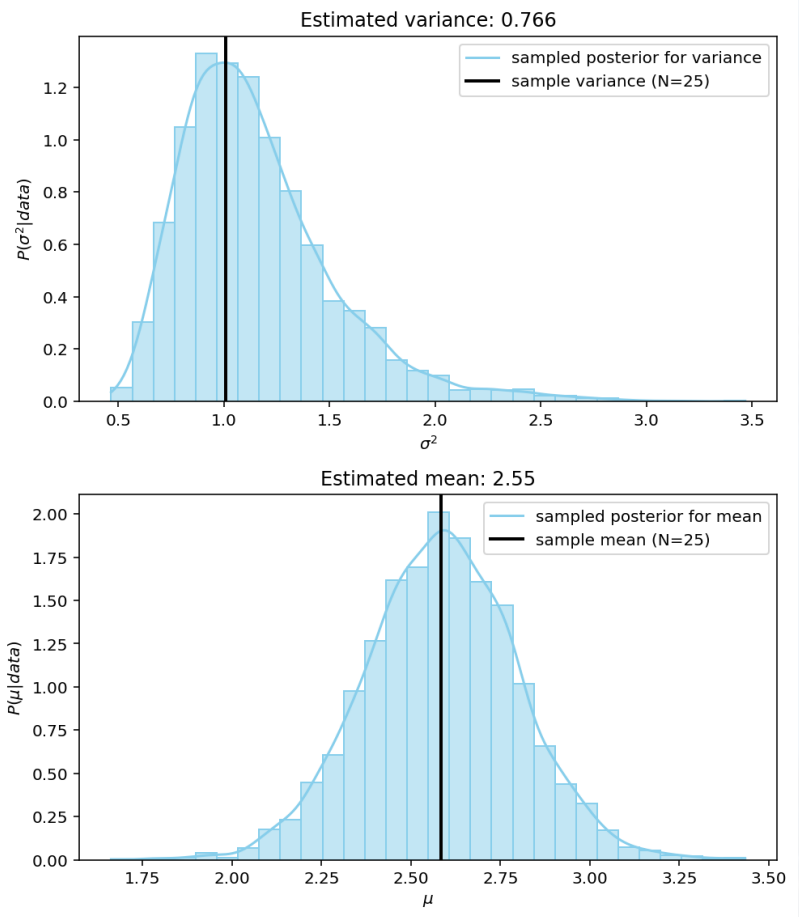
Variational Bayes

`Var_Bayes_Example.py`



Gibbs Sampling

`Gibbs_NormGamma_Example.py`



actual Gibbs Sampling:

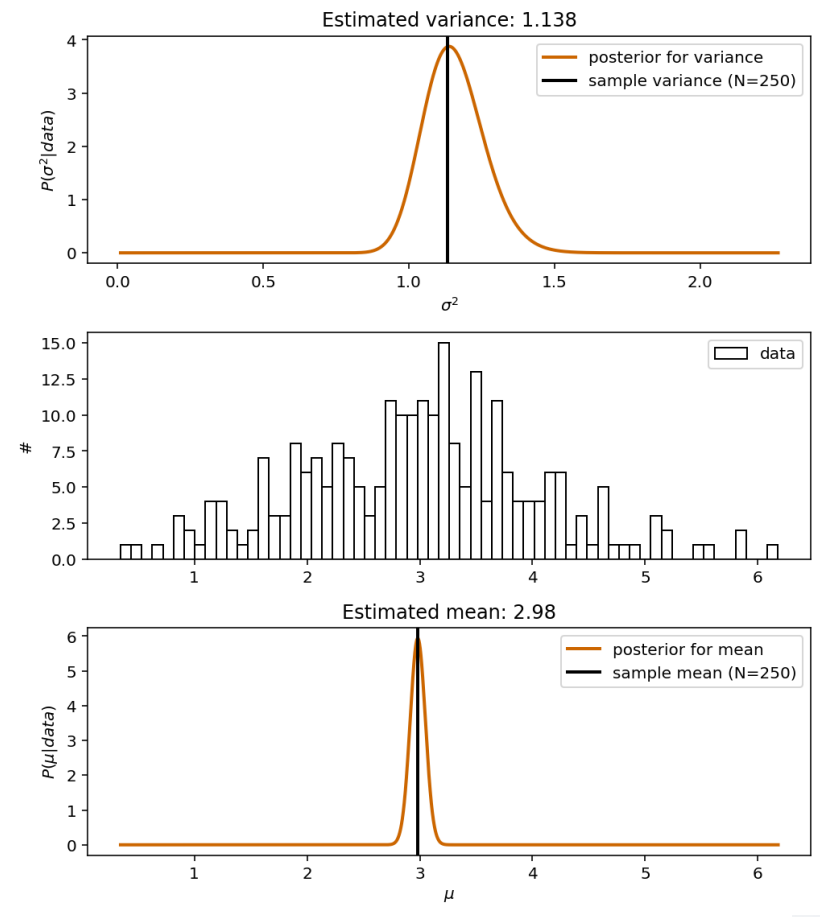
we know from module 6: $q_i(Z_i|D) = \frac{1}{Z}\exp\left(\left\langle E\left(Z_i, \{Z_{j\neq i}\}, D\right)\right\rangle_{\{j\neq i\}}\right)$

$q_i\left(Z_i\middle|\{Z_{j\neq i}\}\right)$ sampling for all i in a particular order

D	: data set of size K
Z	: set of n (latent) parameter
σ^2	: variance
μ	: mean
$\frac{1}{\sigma^2} = \lambda$	: precision

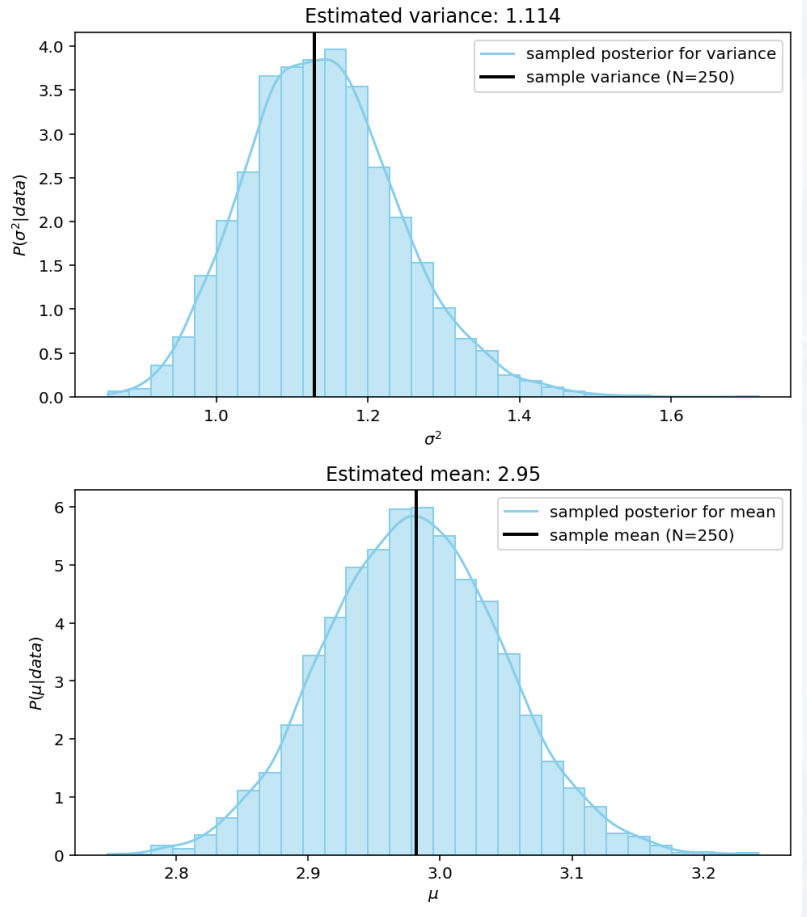
Variational Bayes

`Var_Bayes_Example.py`



Gibbs Sampling

`Gibbs_NormGamma_Example.py`

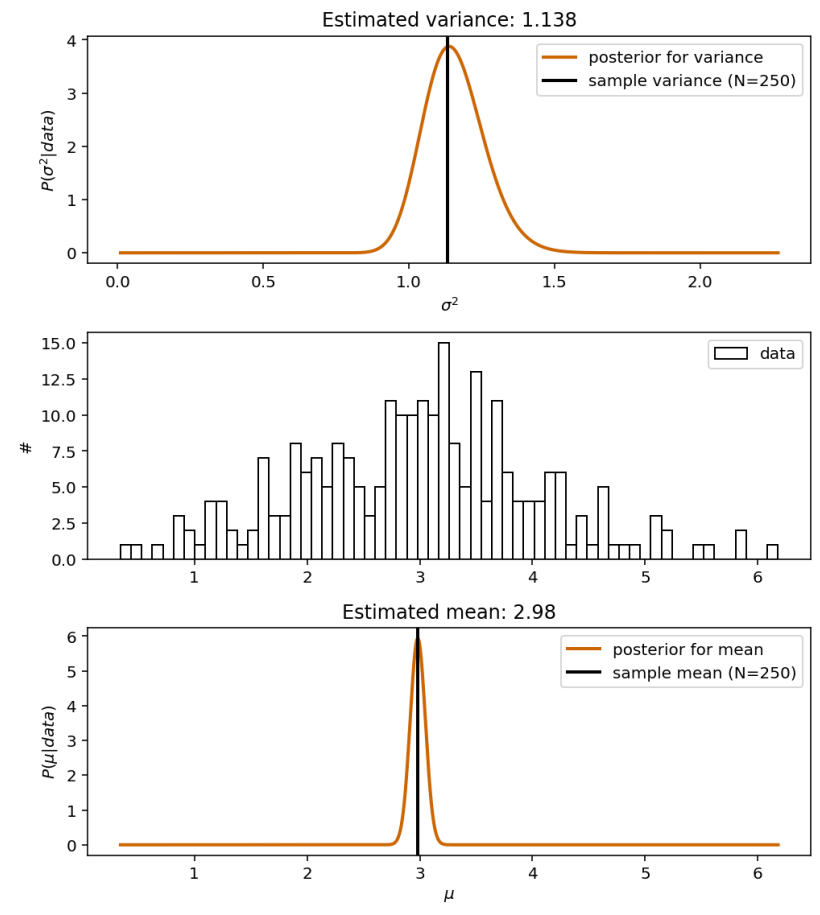


note: - Gibbs sampling does **not guarantee** to find a **global solution**!

- circular dependencies: **has to be ergodic**, otherwise interruption if $q_i\left(Z_i \middle| \{Z_{j\neq i}\}\right) = 0$ for any i

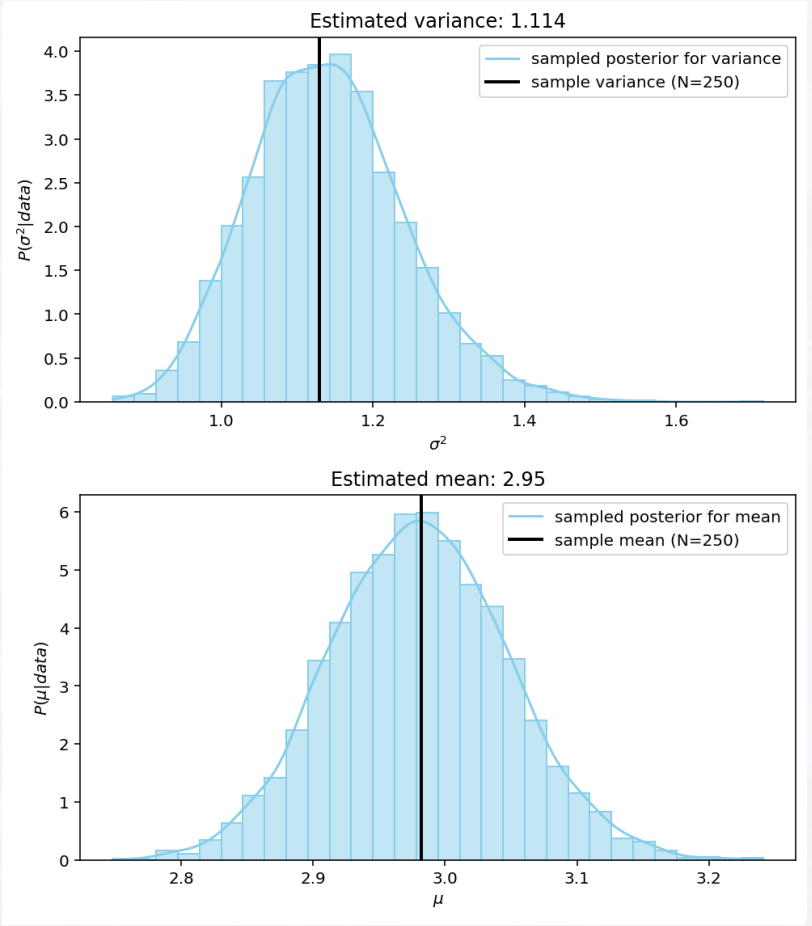
Variational Bayes

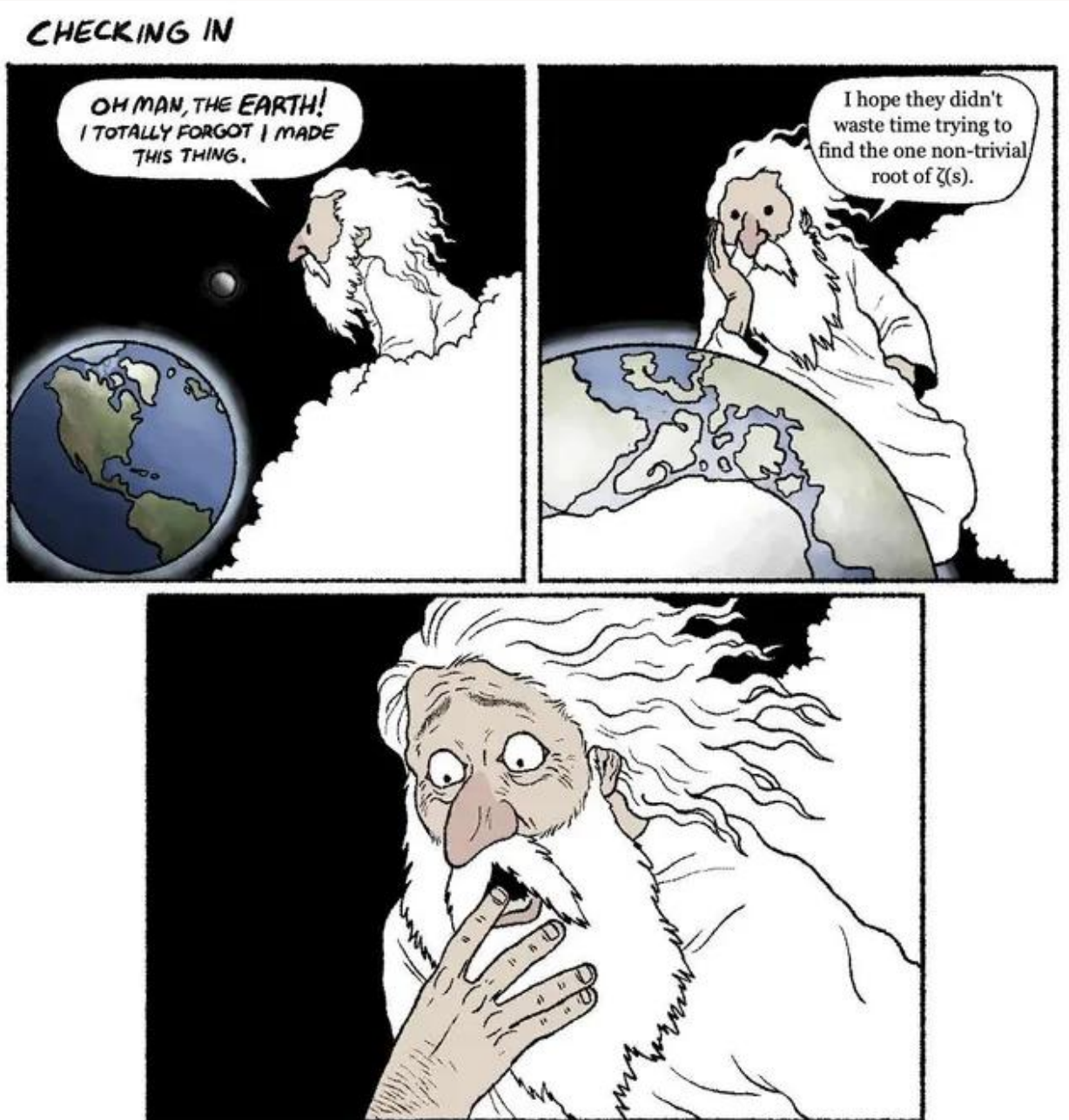
`Var_Bayes_Example.py`



Gibbs Sampling

`Gibbs_NormGamma_Example.py`





Outline

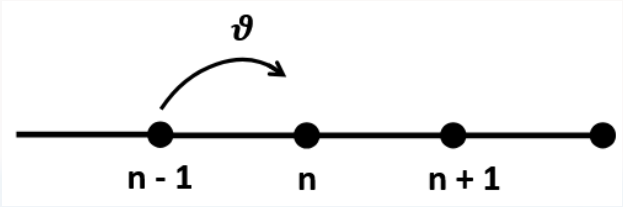
Basic Idea & Finding Pi

Mapping Distributions & Gibbs Sampling

Gillespie Algorithm

Metropolis (- Hastings) Algorithm

Bootstrapping



$$\tau = -\frac{1}{\nu} ln[\rho]$$

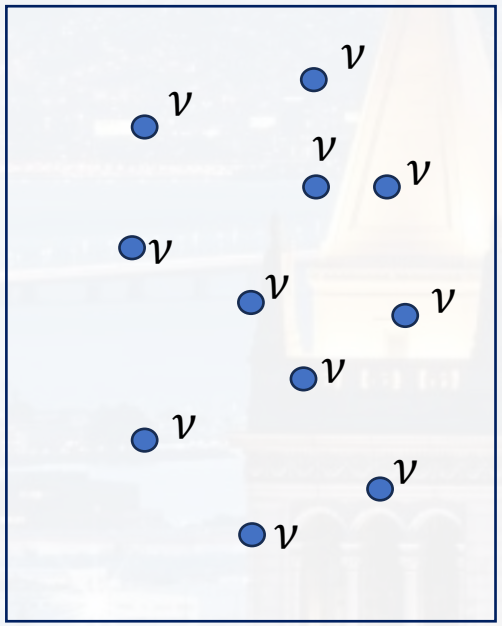
$$w(\tau) = \vartheta e^{-\vartheta\tau}$$

n:	**different states**
ϑ:	**hopping rate ($probability/time$)**
τ:	**waiting time**
$w(\tau)$:	**waiting time distribution**

mean waiting time: $t^* = \frac{1}{\vartheta}$

$A \xrightarrow{k} \emptyset$ stochastic scenario: number *n* of particles *A*

t



for $t = 0$ **many** atoms
→ **τ is small**

Δt

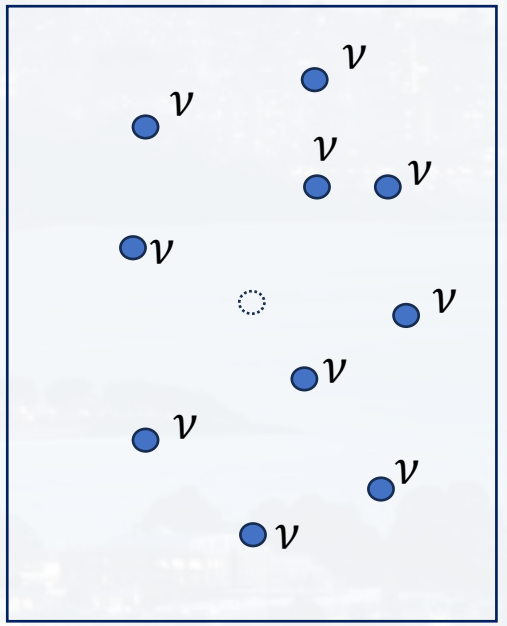
each atom has the probability ν to decay per time

logical ***or*** → **adding** the probabilities

$\nu \rightarrow \nu\, n(t)$

$$\Delta t = -\frac{1}{\nu\, n(t)} ln[P(0|t)]$$

t + Δt



Gillespie:

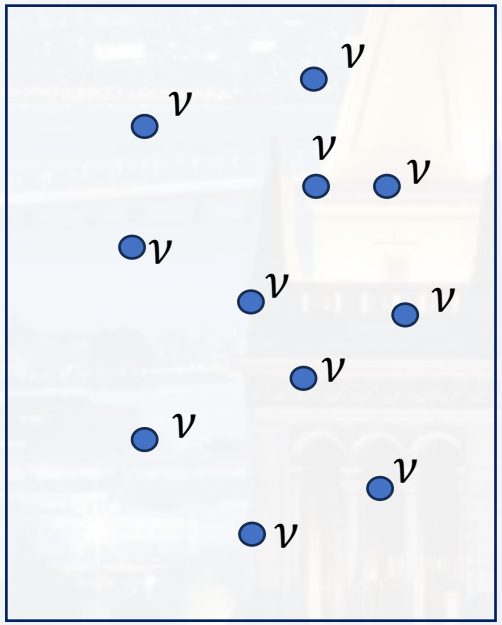
1) draw a **random number** ρ from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$

2) calculate the time Δt that elapses until the next decay

$$\Delta t = -\frac{1}{\nu\, n(t)} \ln \rho$$

3) set $t \rightarrow t + \Delta t$ and $n(t + \Delta t) = n(t) - 1$

4) repeat



$$\Delta t = -\frac{1}{\nu\, n(t)} ln[\rho]$$

Gillespie:

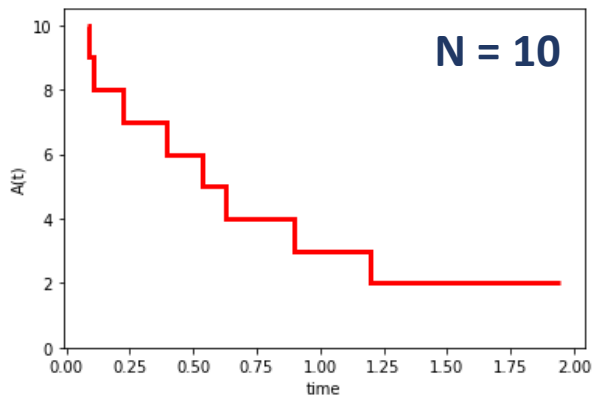
1) draw a **random number** ρ from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$

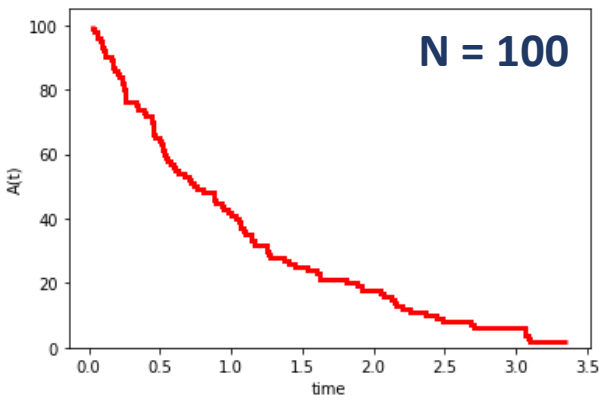
2) calculate the time Δt that elapses until the next decay

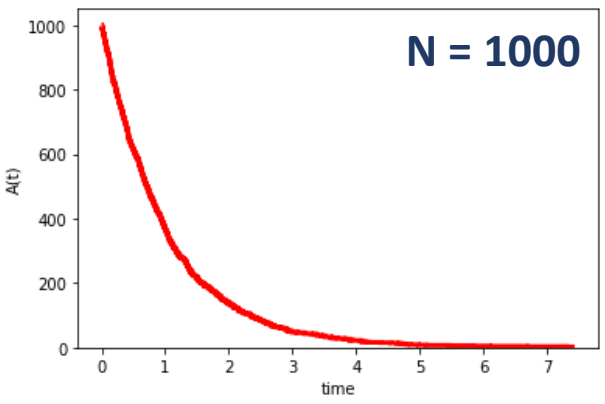
$$\Delta t = -\frac{1}{\nu\, n(t)} \ln \rho$$

3) set $t \rightarrow t + \Delta t$ and $n(t + \Delta t) = n(t) - 1$

4) repeat







see `Decay.py`

$A \underset{k_-}{\overset{k_+}{\rightleftarrows}} B$

n: number of particles of A
m: number of particles of B

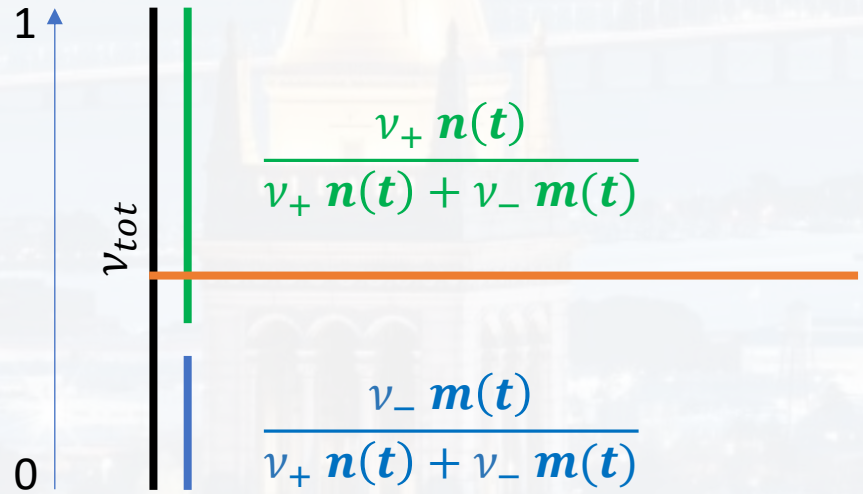
n:	**different states**
ϑ:	**hopping rate ($probability/time$)**
τ:	**waiting time**
$w(\tau)$:	**waiting time distribution**

$$\nu(A) \rightarrow \nu_+ \, n(t) \qquad \nu(B) \rightarrow \nu_- \, m(t) \qquad \nu_{tot} = \nu(A) + \nu(B) = \nu_+ \, n(t) + \nu_- \, m(t)$$

$$\Delta t = -\frac{1}{\nu_+ \, n(t) + \nu_- \, m(t)} ln[\rho]$$

time that elapses until **a** reaction to occurs

next: deciding ***which*** reaction should occur



generating a random number from a **uniform distribution** in the interval $(0, 1)$

depending into which fraction this random number falls
→ this reaction occurs

$A \underset{k_-}{\overset{k_+}{\rightleftarrows}} B$

n: number of particles of A
m: number of particles of B

n:	**different states**
ϑ:	**hopping rate ($probability/time$)**
τ:	**waiting time**
$w(\tau)$:	**waiting time distribution**

Gillespie:

1) draw a **random number** ρ_1 from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$

2) calculate the time Δt that elapses until the next reaction

$$\Delta t = -\frac{1}{\nu_+ n(t) + \nu_- m(t)} \ln \rho_1$$

3) draw a **second** **random number** ρ_2 from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$

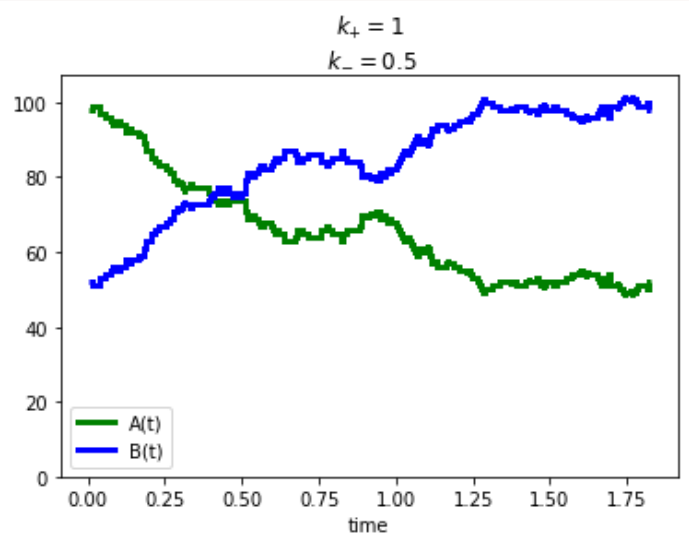
4) decide which reaction occurs:

if $\rho_2 < \frac{\nu_+ n(t)}{\nu_+ n(t) + \nu_- m(t)}$: reaction A → B is more likely

$$n(t + \Delta t) = n(t) - 1$$
$$m(t + \Delta t) = m(t) + 1$$

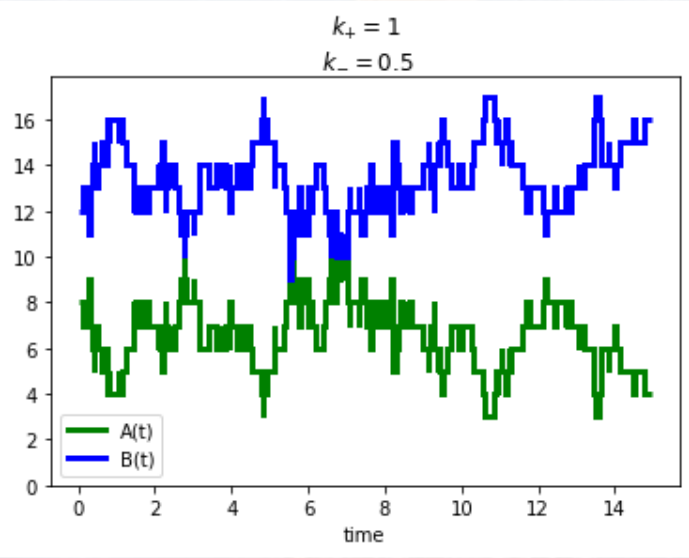
else: reaction B → A is more likely

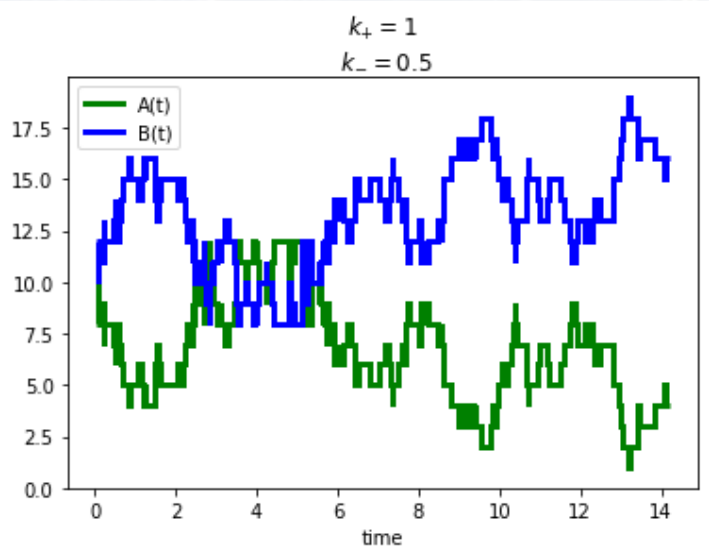
$$n(t + \Delta t) = n(t) + 1$$
$$m(t + \Delta t) = m(t) - 1$$

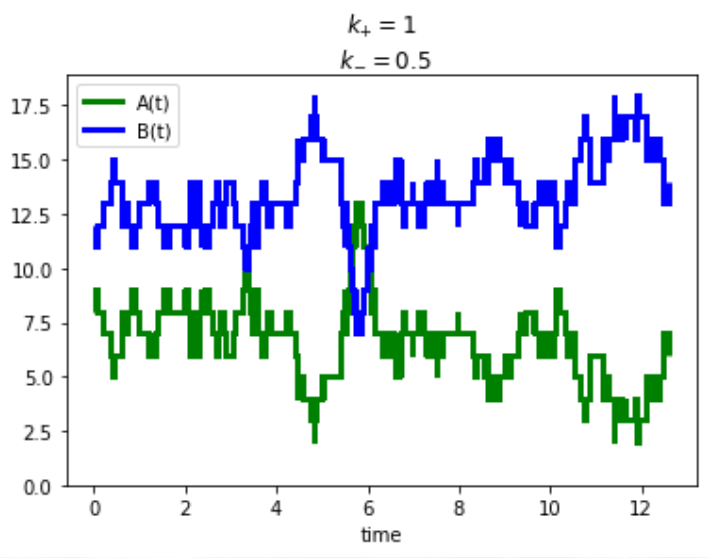


equilibrium at $\frac{A}{B} = \frac{k_-}{k+}$

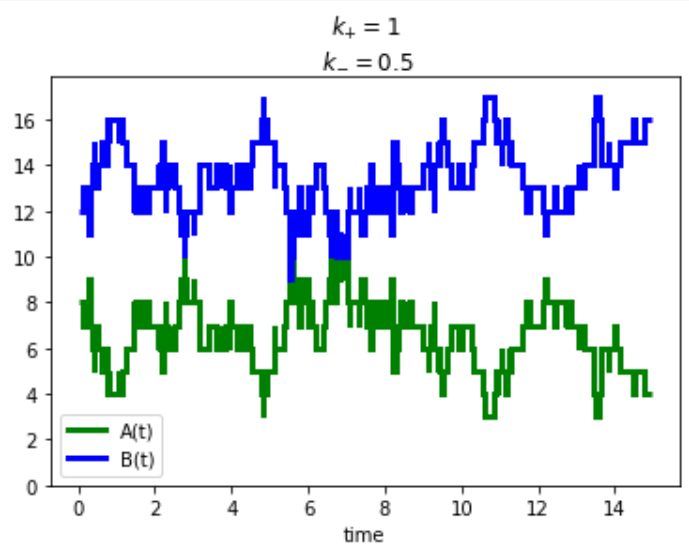
N = 10

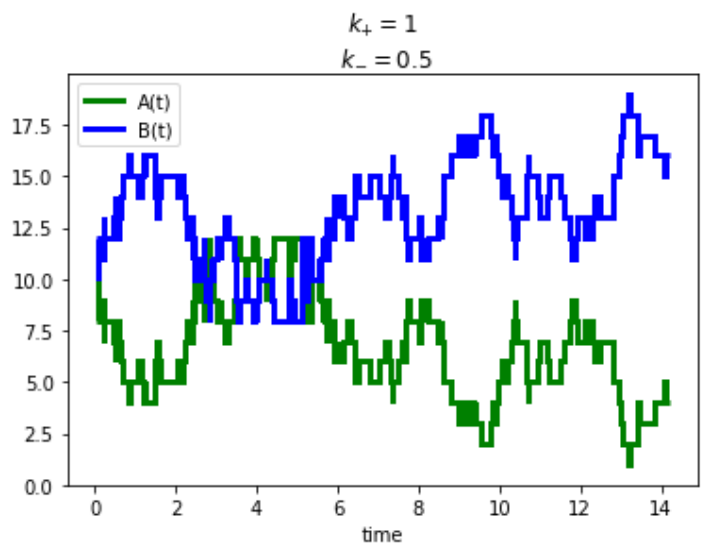


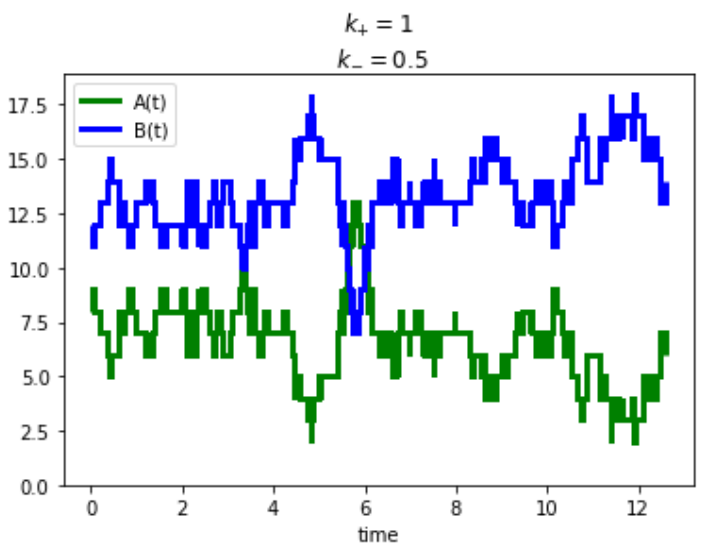




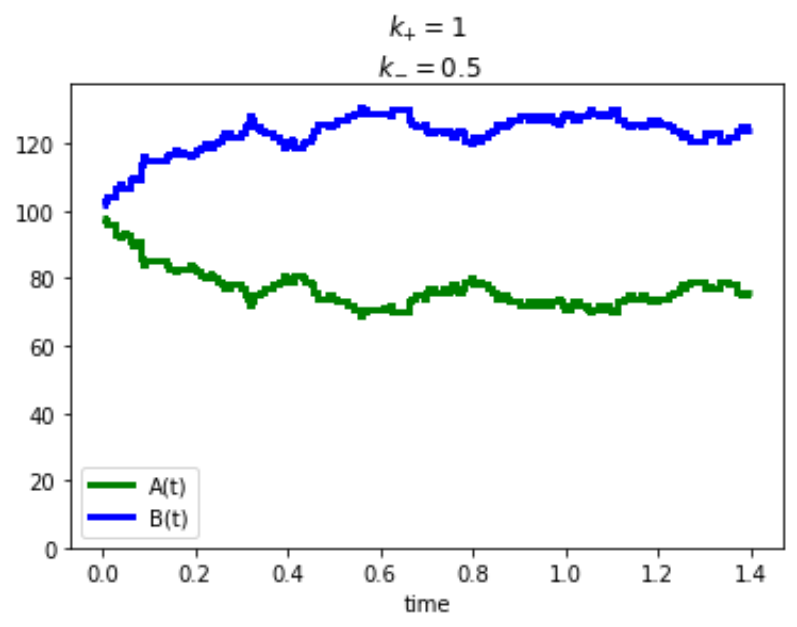
N = 10



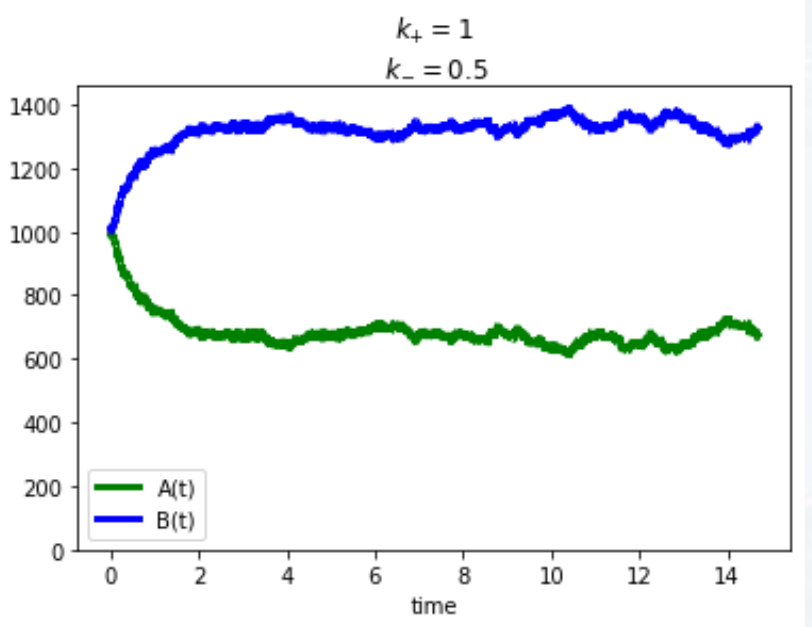


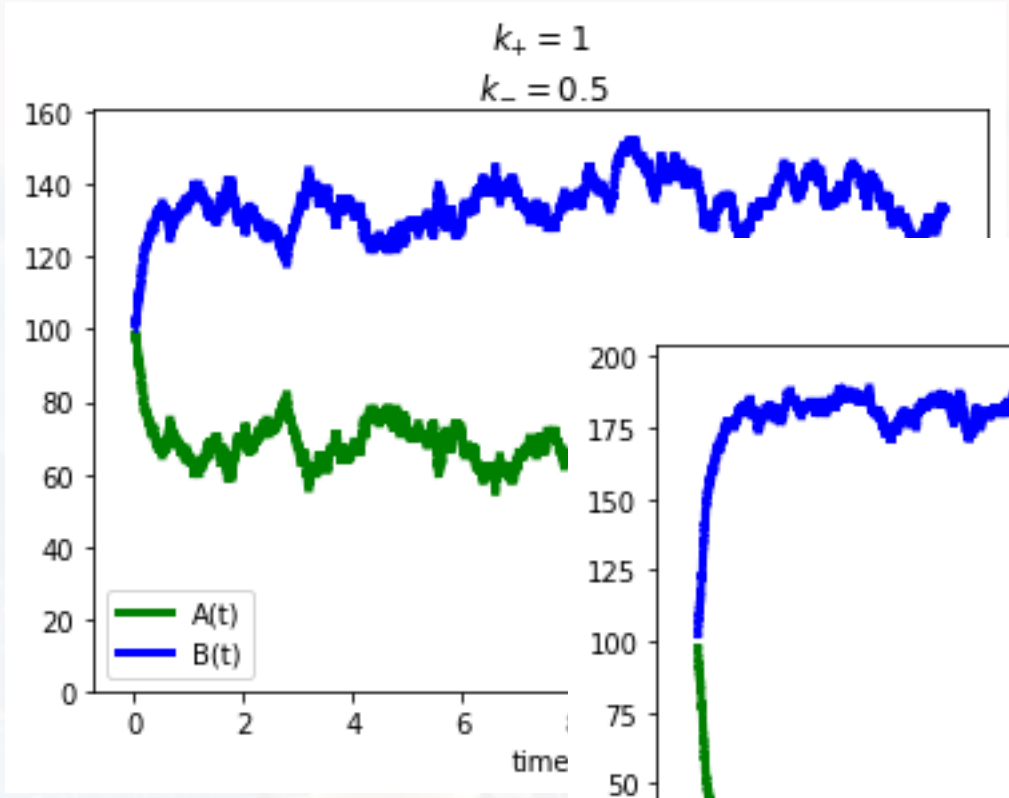


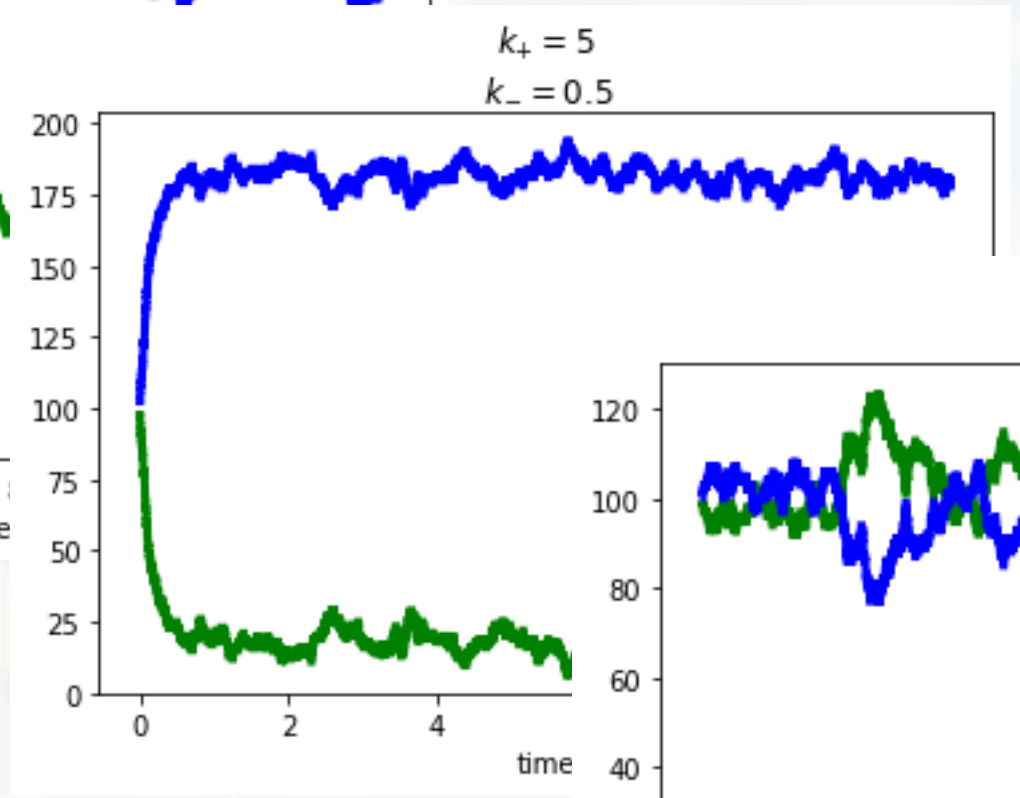
N = 100

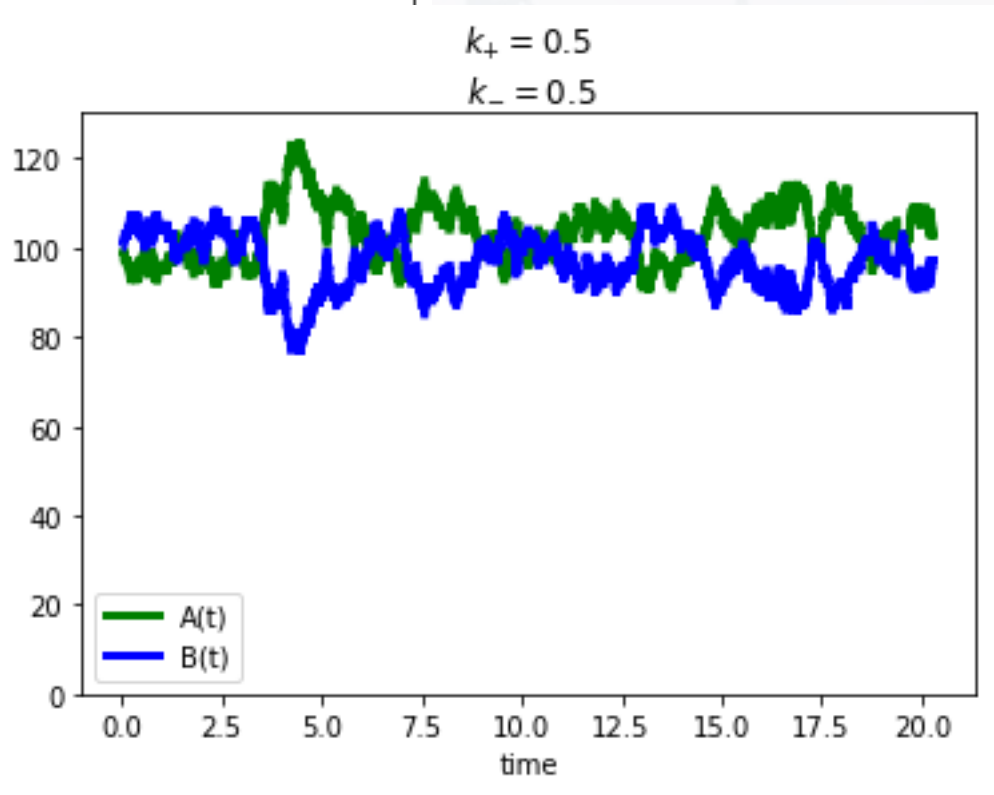


N = 1000




$k_+ = 1$
$k_- = 0.5$
160
140
120
100
80
60
40
20
0
0
2
4
6
time
A(t)
B(t)


$k_+ = 5$
$k_- = 0.5$
200
175
150
125
100
75
50
25
0
0
2
4
time


$k_+ = 0.5$
$k_- = 0.5$
120
100
80
60
40
20
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
time
A(t)
B(t)

We also know how to solve the Predator-Prey model now!

L: sheep (lambs)
W: wolfs
E: "empty"

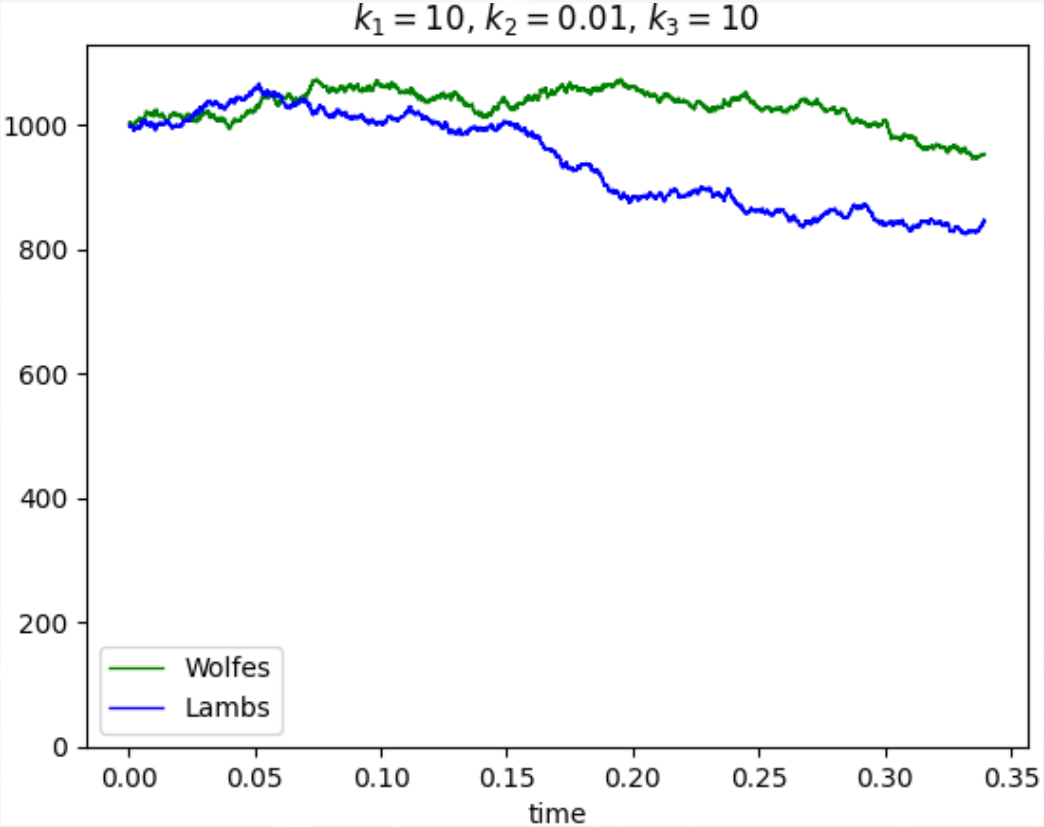
$$L \xrightarrow{k_1} 2\,L$$

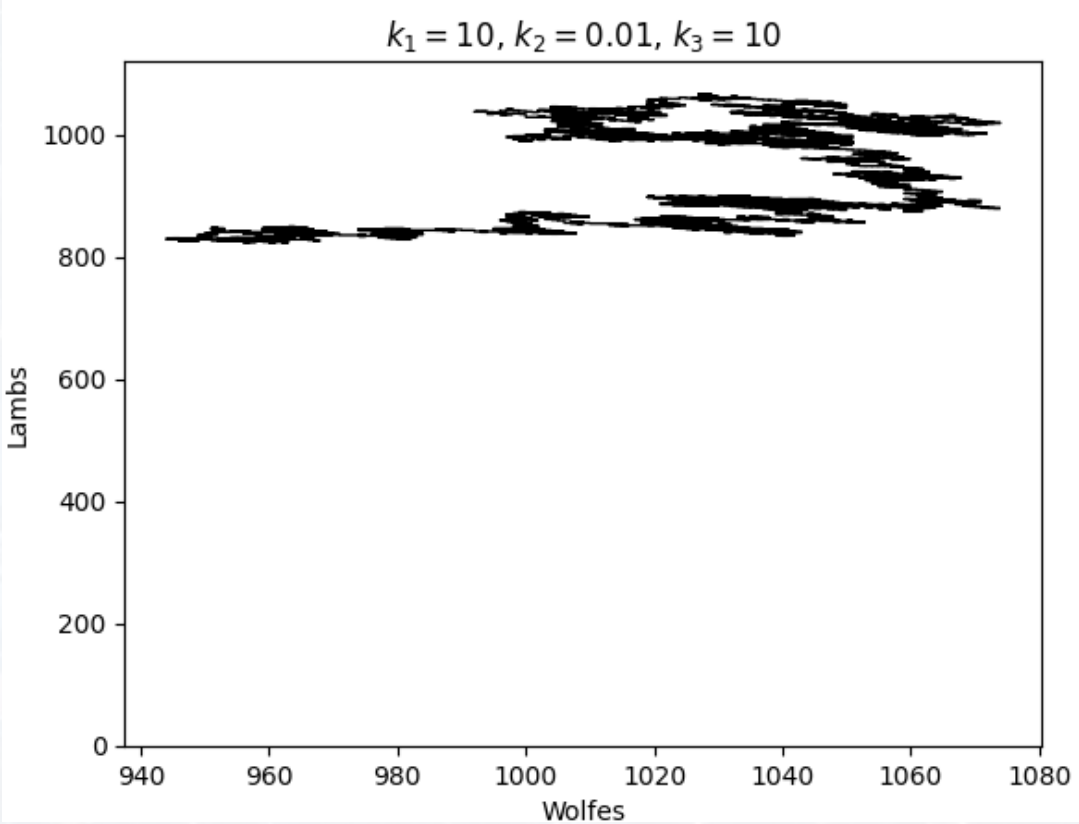
$$L + W \xrightarrow{k_2} 2\,W$$

$$W \xrightarrow{k_3} \Phi$$

$$A \underset{k_-}{\overset{k_+}{\rightleftarrows}} B$$

n:	different states
ϑ:	hopping rate
dt:	time increment
$\tau = -ln[P(0,t)]\,/\nu$:	waiting time





We also know how to solve the Predator-Prey model now!

L: sheep (lambs)
W: wolfs
E: “empty”

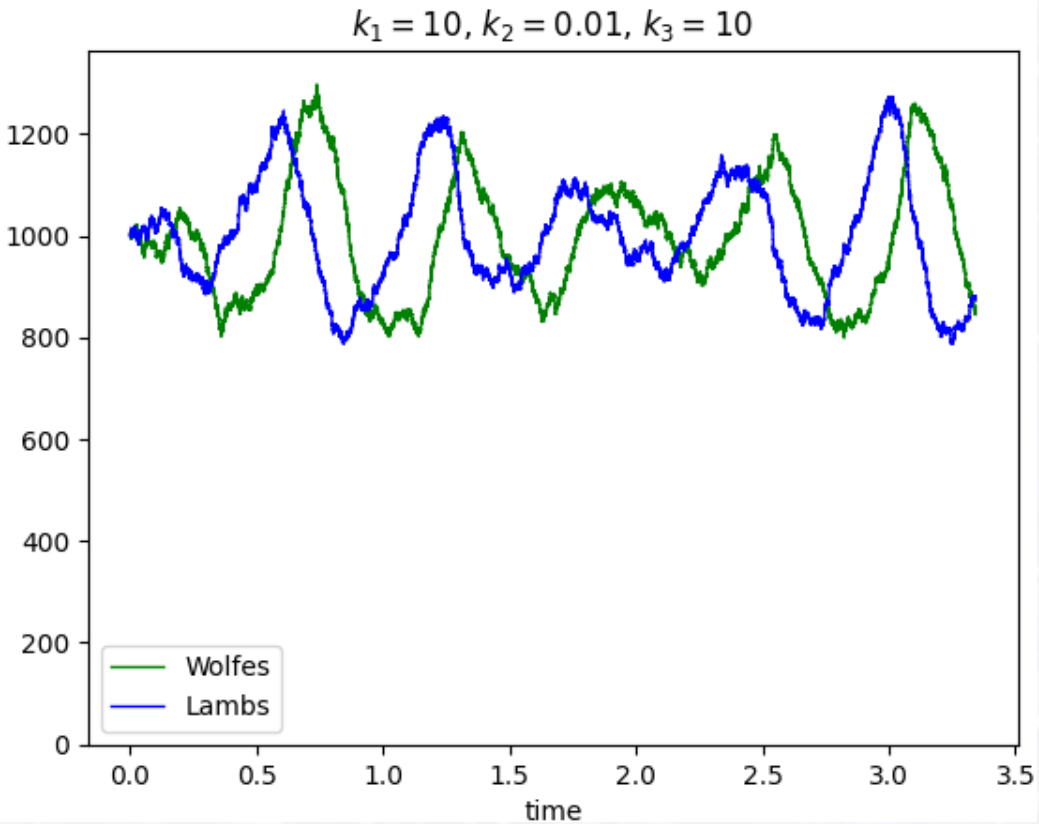
$$L \xrightarrow{k_1} 2\,L$$

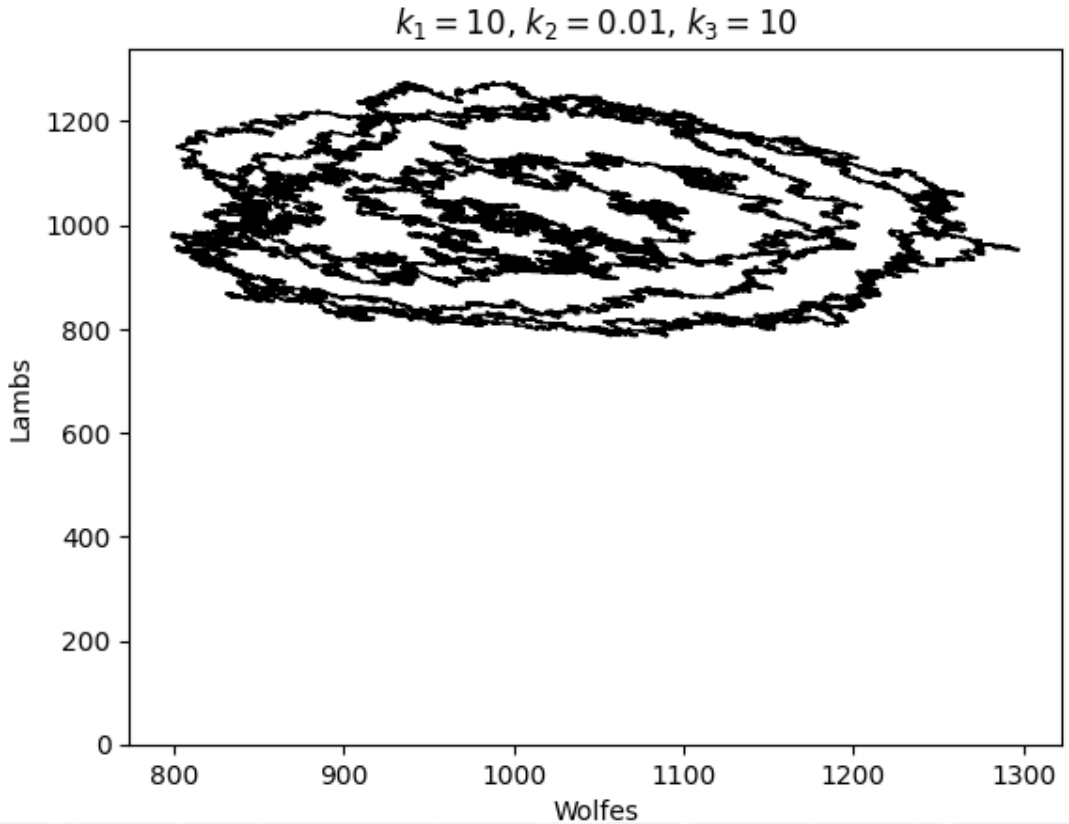
$$L + W \xrightarrow{k_2} 2\,W$$

$$W \xrightarrow{k_3} \Phi$$

$$A \underset{k_-}{\overset{k_+}{\rightleftarrows}} B$$

n:	different states
ϑ:	hopping rate
dt:	time increment
$\tau = -ln[P(0,t)]\,/\nu$:	waiting time





We also know how to solve the Predator-Prey model now!

L: sheep (lambs)
W: wolfs
E: “empty”

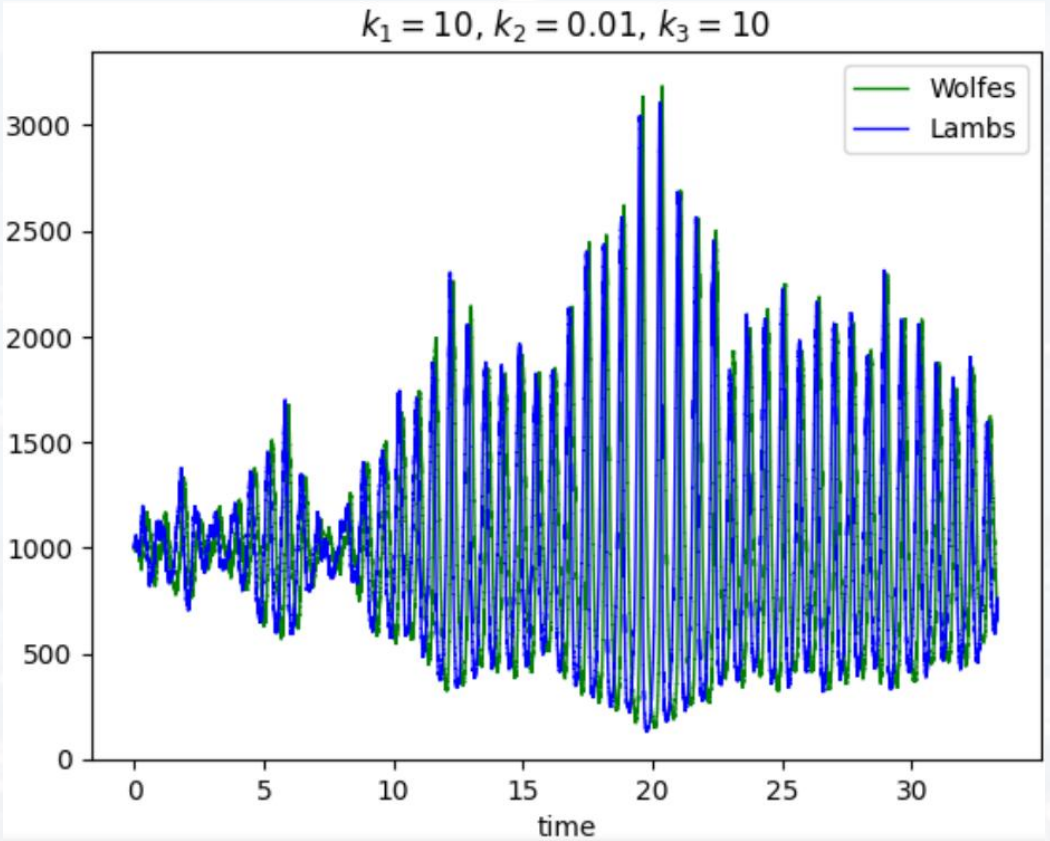
$$L \xrightarrow{k_1} 2L$$

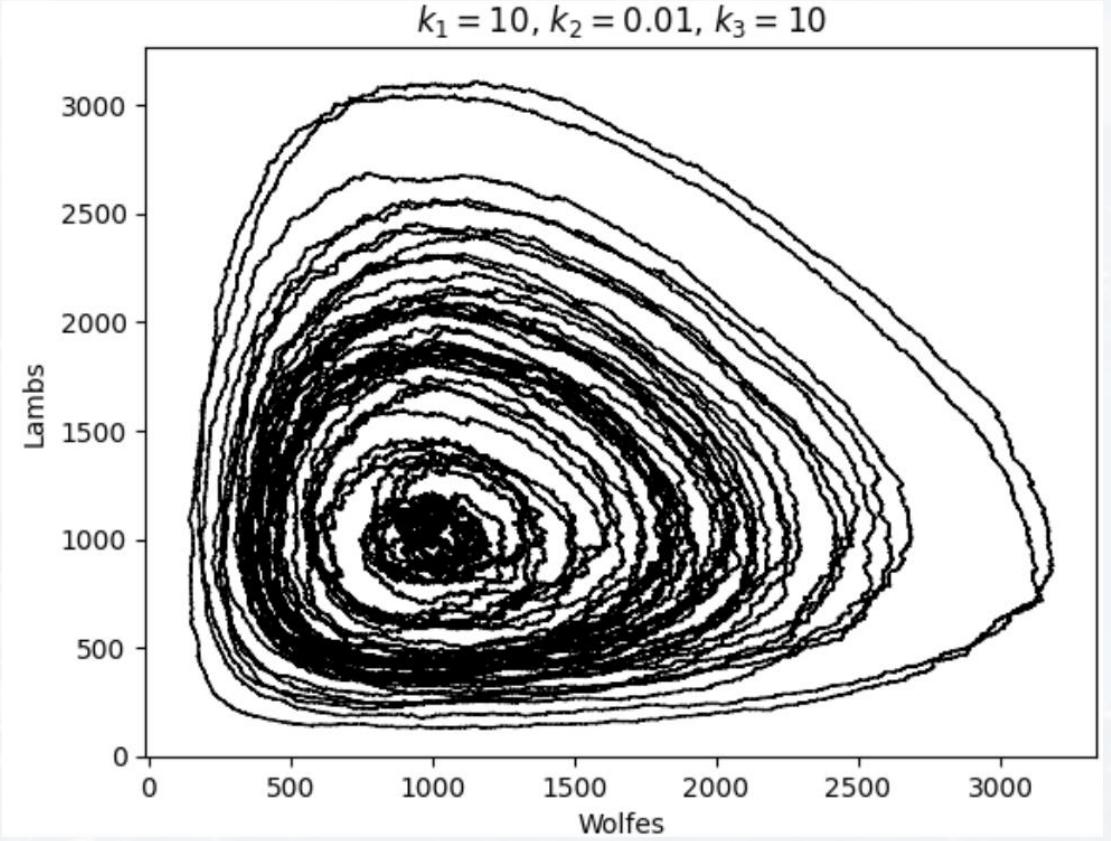
$$L + W \xrightarrow{k_2} 2W$$

$$W \xrightarrow{k_3} \Phi$$

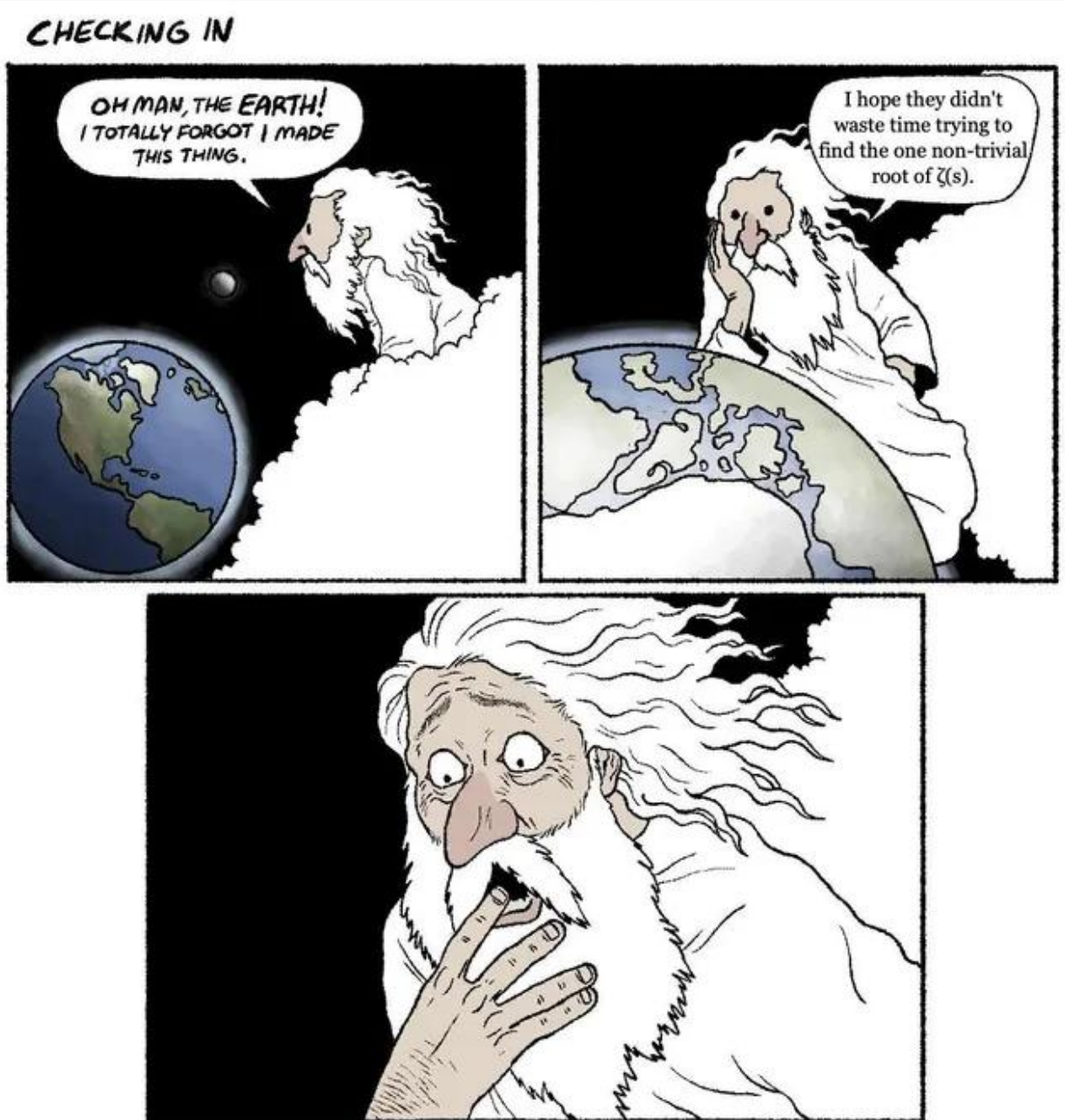
$$A \underset{k_-}{\overset{k_+}{\rightleftarrows}} B$$

n:	different states
ϑ:	hopping rate
dt:	time increment
$\tau = -ln[P(0,t)]\,/\nu$:	waiting time





Berkeley UNIVERSITY OF CALIFORNIA Monte Carlo Methods



Outline

Basic Idea & Finding Pi

Mapping Distributions & Gibbs Sampling

Gillespie Algorithm

Metropolis (- Hastings) Algorithm

Bootstrapping

problem: sometimes we need to draw from a probability distribution (= **target**) that is **difficult to sample** from directly

idea: drawing from **a proposal distribution**
accept/reject the proposal based on an acceptance probability

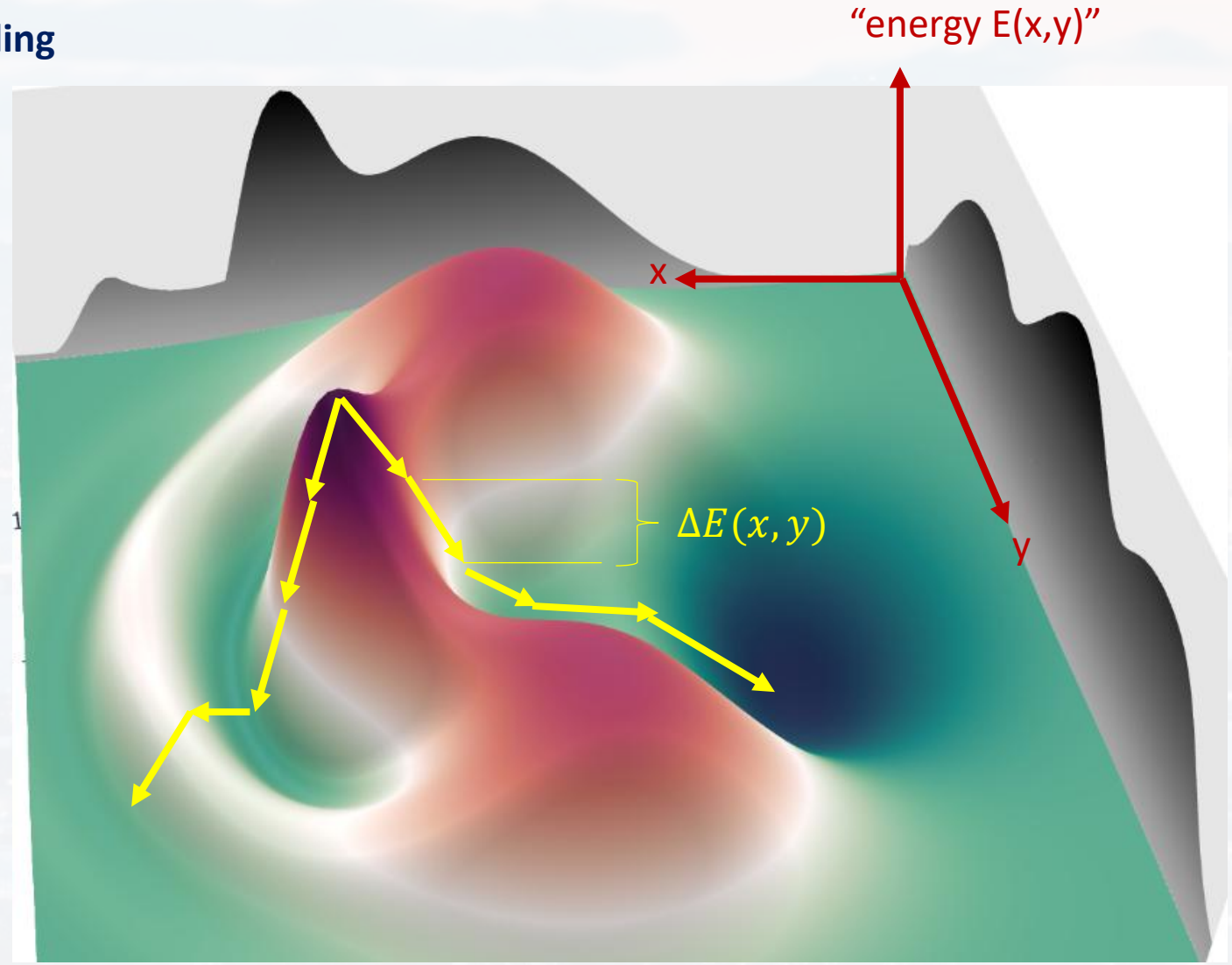
application: in Physics: often **energy-based models**

sampling from $p_i(x|\vartheta) \sim exp\{-E(x,\vartheta)\}$

essentially samples the **partition function** $\mathcal{Z}$

Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

example:
simulated annealing



If $\Delta E(x,y)$ is **negative**:
→ **always move**
(a ball always rolls down the hill)

If $\Delta E(x,y)$ is **positive**:
→ calculate the **probability to move**
→ leaves some chance to escape local minimum

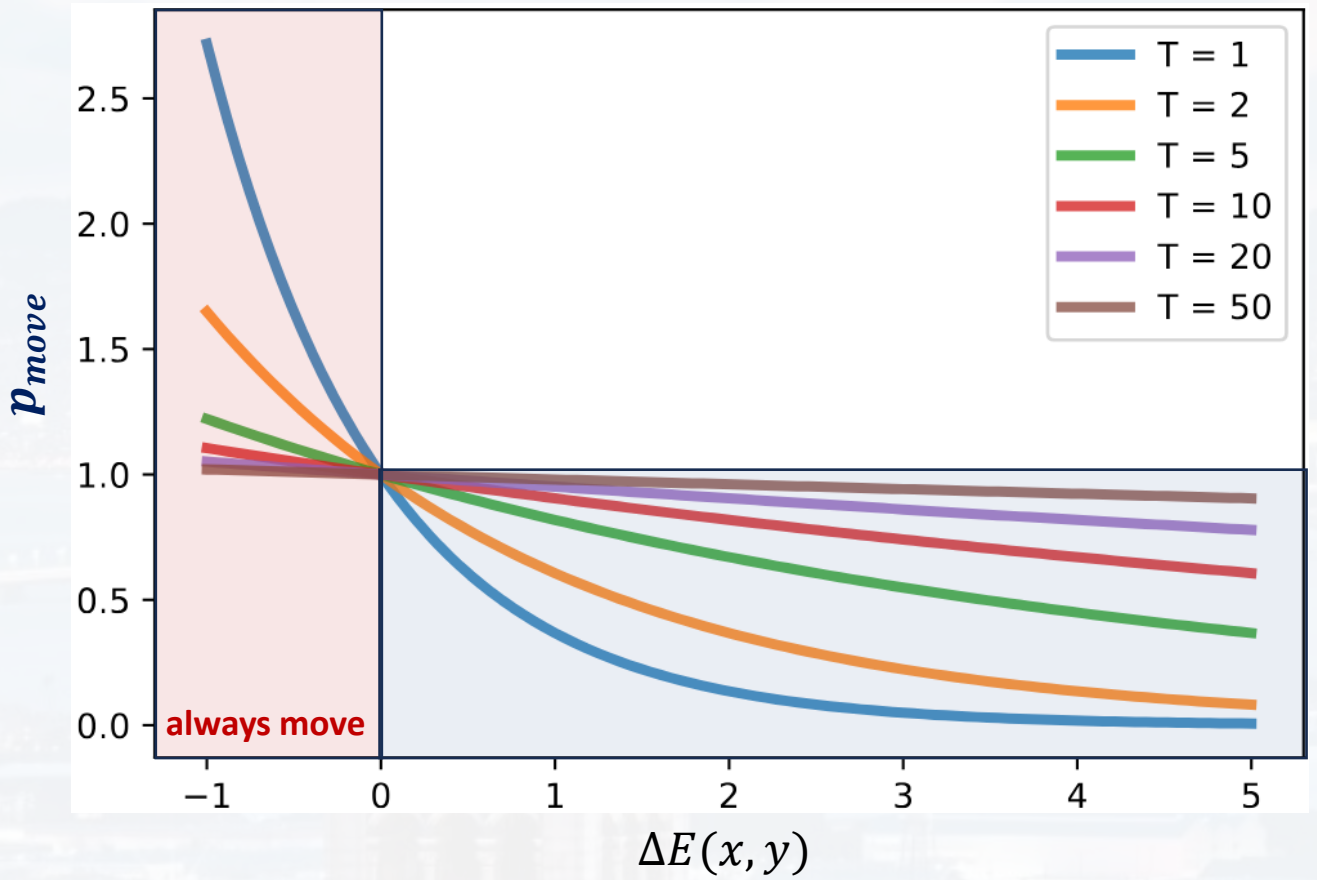
T: temperature

Boltzmann factor

$$p_{move} \sim \exp\left[-\frac{\Delta E(x,y)}{T}\right]$$

Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

example:
simulated annealing



slowly reducing T → making larger jumps ($\Delta E(x, y)$) less likely over time

If $\Delta E(x, y)$ is **negative**:
→ **always move**
(a ball always rolls down the hill)

If $\Delta E(x, y)$ is **positive**:
→ calculate the **probability to move**
→ leaves some chance to escape local minimum

T: temperature

Boltzmann factor

$$p_{move} \sim \exp\left[-\frac{\Delta E(x, y)}{T}\right]$$

Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

Metropolis:

1) suggest a random move $\Delta\vec{r}$

2) calculate $\Delta E = E(\vec{r}) - E(\vec{r} + \Delta\vec{r})$

3) move or not:

a) move if $\Delta E < 0$

b) if $\Delta E > 0$

- draw a **random number ρ** from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$
- move if $\rho < \exp\left[-\frac{\Delta E}{T}\right]$

4) reduce *T* and repeat

Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

Metropolis: 1) suggest a random move $\Delta\vec{r}$

2) calculate $\Delta E = E(\vec{r}) - E(\vec{r} + \Delta\vec{r})$

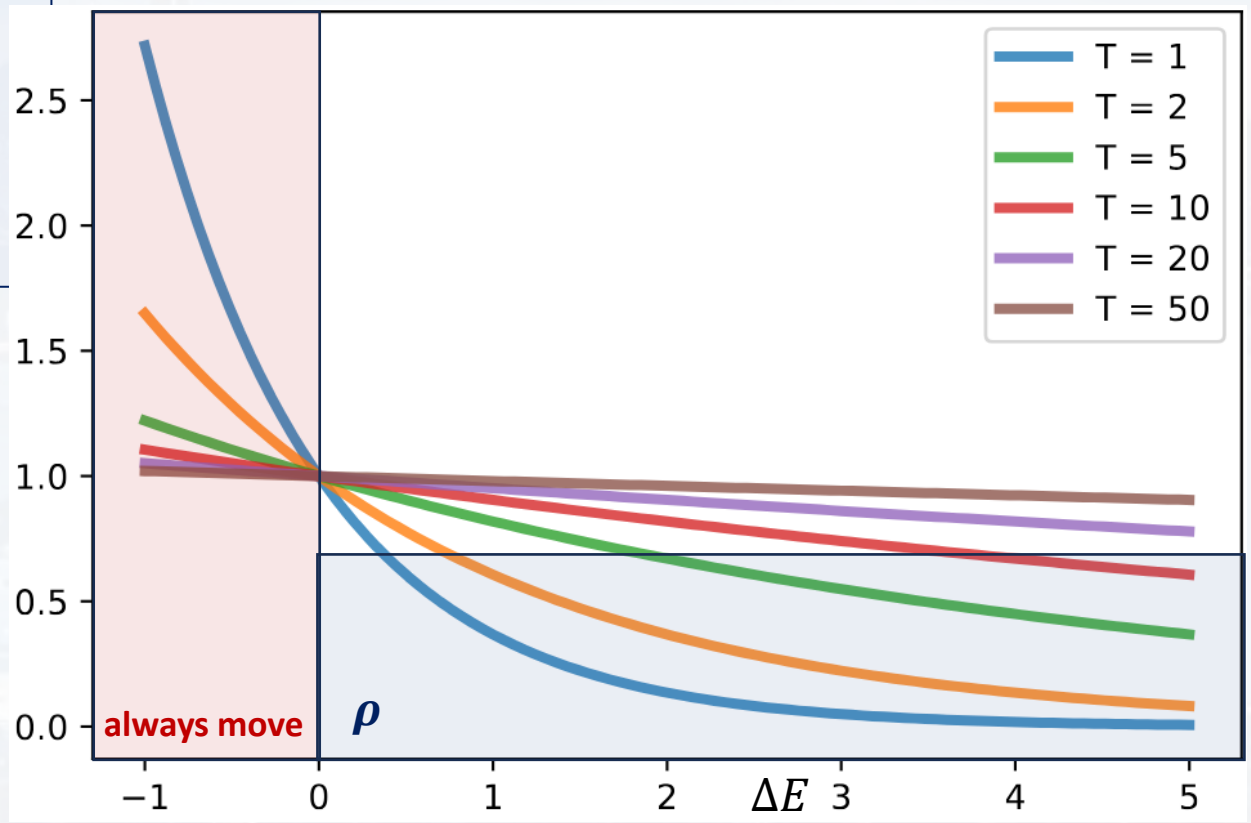
3) move or not:

a) move if $\Delta E < 0$

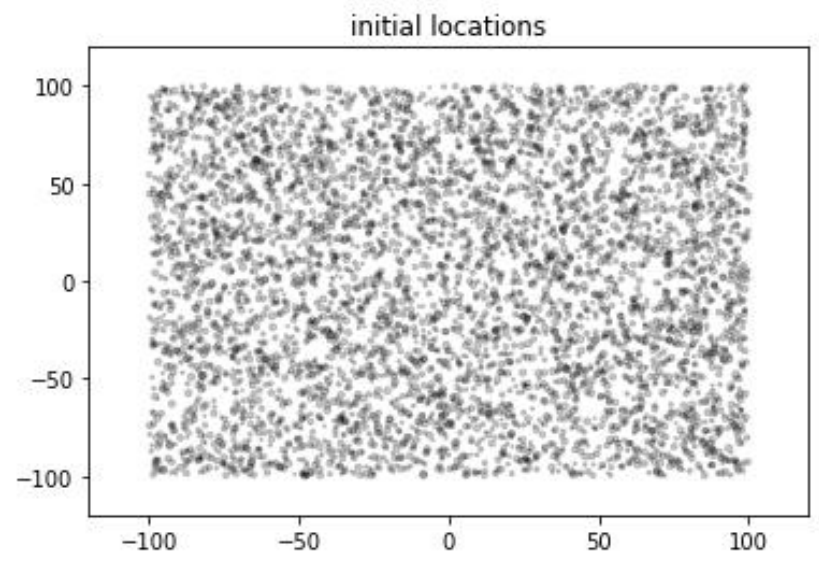
b) if $\Delta E > 0$

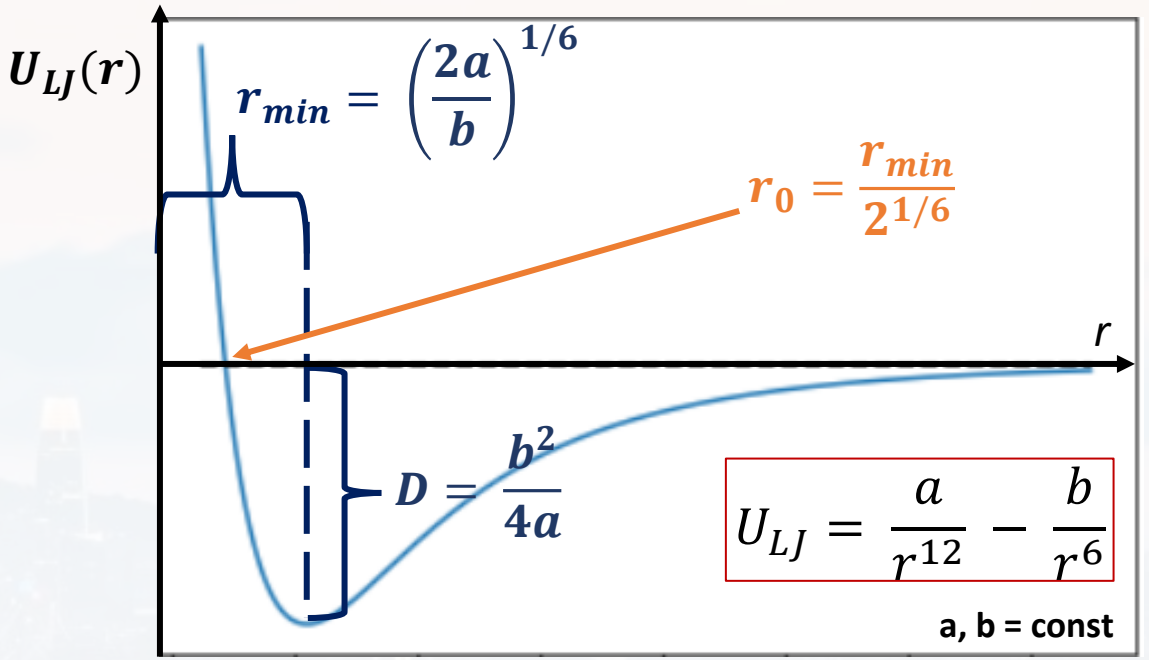
- draw a **random number** $\boldsymbol{\rho}$ from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$
- move if $\rho < \exp\left[-\frac{\Delta E}{T}\right]$

4) reduce *T* and repeat



example: alternative to finite differences





simulating many particles

Naïve solution: solving **Newton's equation of motion**

total force/potential that acts on the particle

$$x_{t+\Delta t} = x_t + v(x)_t \cdot \Delta t + \frac{1}{2} a(x)_t \Delta t^2 \qquad a(x) = \frac{F(x)_{tot}}{m} = \frac{1}{m}\frac{\partial U_{tot}(x,y)_{LJ}}{\partial x}$$

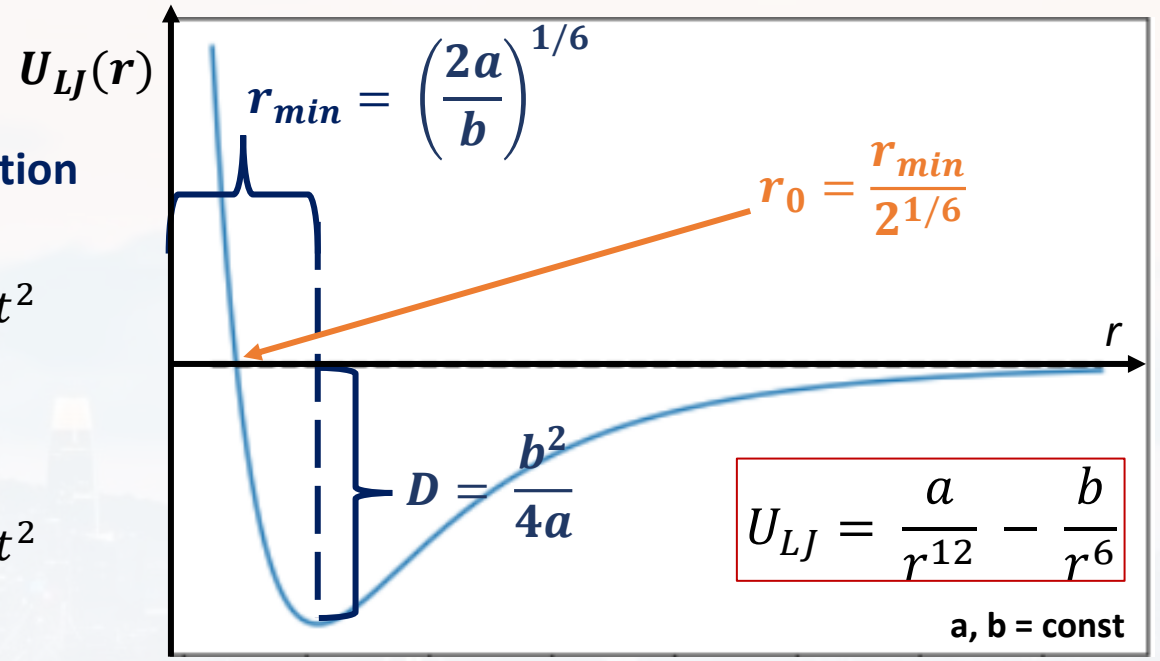
$$y_{t+\Delta t} = y_t + v(y)_t \cdot \Delta t + \frac{1}{2} a(y)_t \Delta t^2 \qquad a(y) = \frac{F(y)_{tot}}{m} = \frac{1}{m}\frac{\partial U_{tot}(x,y)_{LJ}}{\partial y}$$

example: alternative to finite differences

Naïve solution: solving **Newton's equation of motion**

$$x_{t+\Delta t} = x_t + v(x)_t \cdot \Delta t + \frac{1}{2m}\frac{\partial U_{tot}(x,y)_{LJ}}{\partial x}\bigg|_t \Delta t^2$$

$$y_{t+\Delta t} = y_t + v(y)_t \cdot \Delta t + \frac{1}{2m}\frac{\partial U_{tot}(x,y)_{LJ}}{\partial y}\bigg|_t \Delta t^2$$



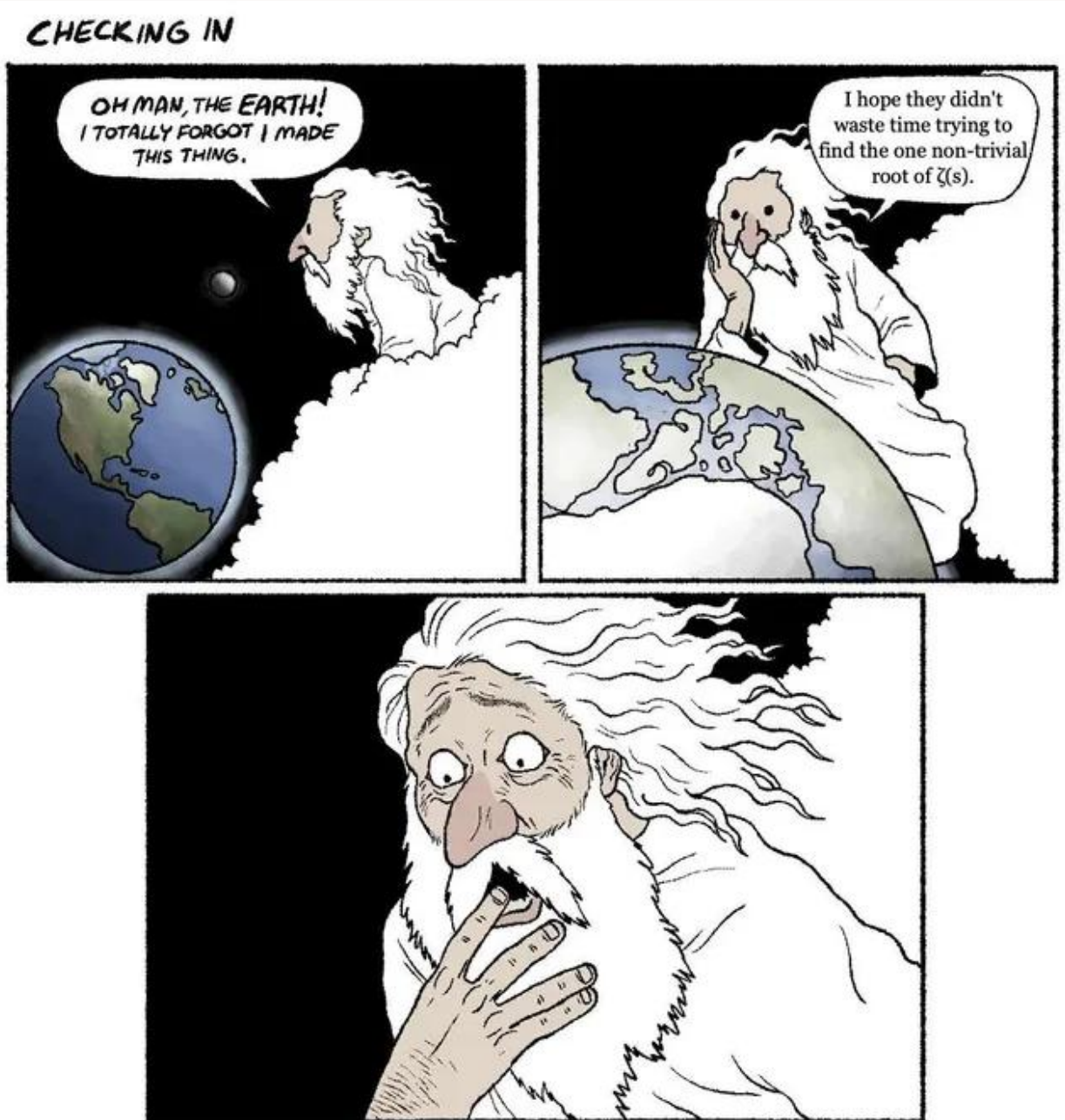
We pick a specific value for Δt and update locations, velocities and acceleration

for particles with $r \approx r_0$:

- $\frac{\partial U(x,y)_{LJ}}{\partial y}$ or $\frac{\partial U(x,y)_{LJ}}{\partial x}$ explode
- particles get kicked out
- wouldn't have gotten so close in the first place → Δt **too large**

for particles with $r \gg r_0$:

- $\frac{\partial U(x,y)_{LJ}}{\partial y}$ or $\frac{\partial U(x,y)_{LJ}}{\partial x} \approx 0$
- nothing happens, very inefficient → Δt **too small**



Outline

Basic Idea & Finding Pi

Mapping Distributions & Gibbs Sampling

Gillespie Algorithm

Metropolis (- Hastings) Algorithm

Bootstrapping

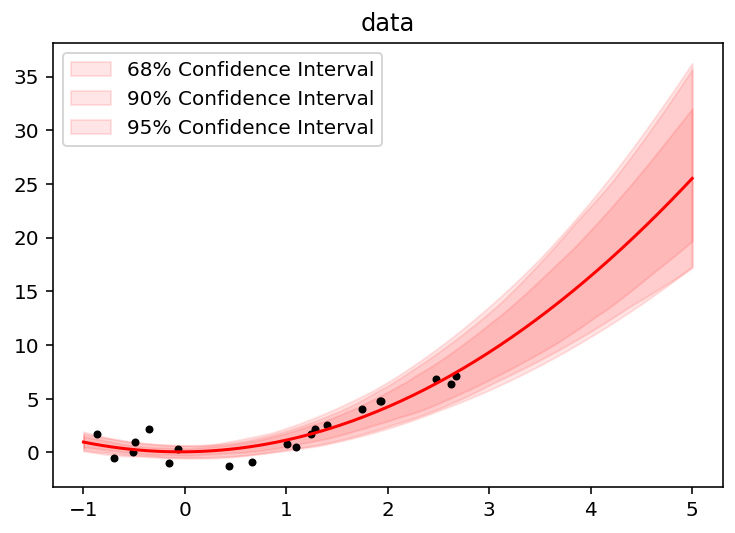
problem: sometimes datapoints x_i don't have error bars

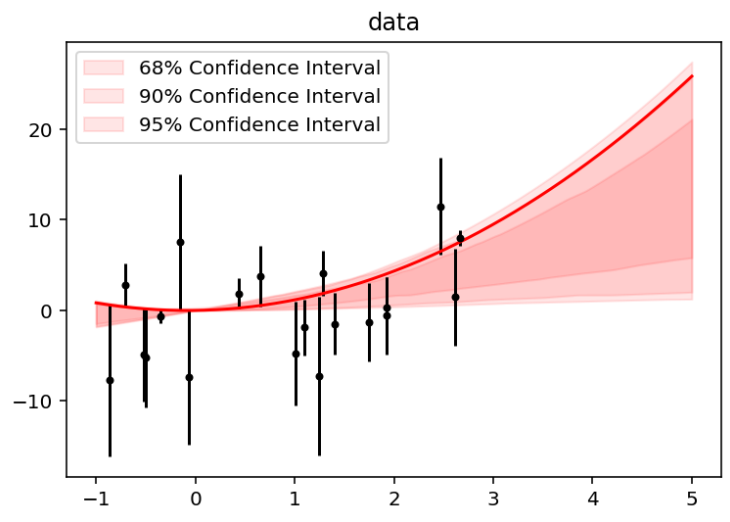
idea: → sampling from x_i in order to estimate **confidence intervals**

$x_i|model \sim ?$

if x_i have error bars σ_i, usually sampling from normal distribution

$x_i|\sigma \sim \mathcal{N}(\bar{x}_i, \sigma_i^2)$





Physics 77/88

```
from scipy.optimize import curve_fit

def fun_to_fit(x, a, b, c):
        return a*x**2 + b*x + c

ValsBest, Cov = curve_fit(fun_to_fit, x, y)
```

Physics 77/88

```
from scipy.optimize import curve_fit

def fun_to_fit(x, a, b, c):
        return a*x**2 + b*x + c

ValsBest, Cov = curve_fit(fun_to_fit, x, y)
```

bootstrapping:

if **no errors of y_i known**

→ assuming that fitted parameters follow a **normal distribution**, i.e. $a = 1.37 \pm 0.39$ where $\mu_a = 1.37$ and $\sigma_a = 0.39$ and so on…

→ **varying** the parameters **within their errors** using `np.random.normal(`μ_a`,` σ_a`, N)` **N** times

→ for each **N, generating a curve fit**

→ from set of N curve fits → **calculating percentiles** for confidence band/ interval

Physics 77/88

```
from scipy.optimize import curve_fit

def fun_to_fit(x, a, b, c):
        return a*x**2 + b*x + c

ValsBest, Cov = curve_fit(fun_to_fit, x, y)
```

bootstrapping:

if **errors of y_i known**

→ assuming that errors of y_i follow a **normal distribution**, i.e. $y_i(boot) = y_i \pm \sigma_i$

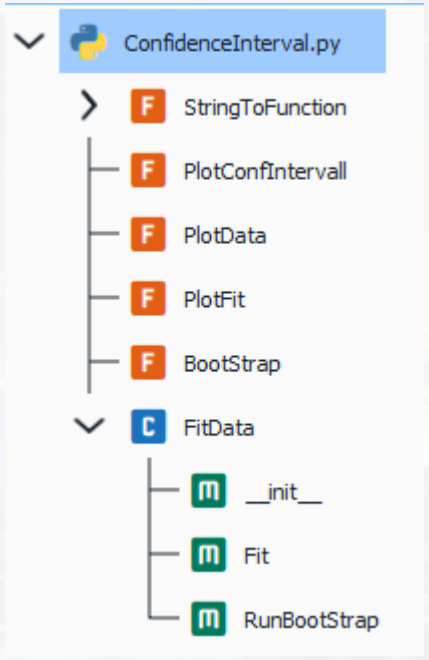
→ **varying** all y_i **within their errors** using `np.random.normal(`y_i`,` σ_i`, N)` **N** times

→ for each **N, generating a curve fit**

→ from set of N curve fits → **calculating percentiles** for confidence band/ interval

bootstrapping

`ConfidenceInterval.py`




```
USAGE:
····generating·a·test·sample:
····
····x·········=·np.linspace(-1,3,20)
····err·······=·np.random.normal(0,·1,·(len(x),))#1sigma·errorbars
····y·········=·x**2·+·err
····errorbars·=·abs(err)
····
····
····1)·plotting·data
········
····F1·=·FitData(x,·y)
····F2·=·FitData(x,·y,·errorbars)
····F3·=·FitData(x,·y,·errorbars,·time·=·'[s]',·pressure·=·'[MPa]')
····
····
····2)·fitting·data·(returns·best·values·of·fitted·params,·1sigma·confidence·and
····················reduced·chi2·if·errorbars·given,·MSE·else)
····················
····res1··=·F1.Fit()
····res2··=·F2.Fit()
····res2··=·F3.Fit()
····
····res12·=·F1.Fit("a*x**2",·[1],·(-0.5,·10))
····
····
····3)·Bootstrapping·(either·varying·within·errorbars·or·within·conf·of·fitted
····················params)
····················
····F1.RunBootStrap()
····F2.RunBootStrap()
····F3.RunBootStrap()
····
····F1.RunBootStrap(100,·[90,·95],·np.linspace(-1,5,200))
····F3.RunBootStrap(100,·[90,·95],·np.linspace(-1,5,200))
"""
```

Thank you very much for your attention!