

Monte Carlo Methods

Random Numbers

- How random is random?
 - From Pang: we want
 - Long period before sequence repeats
 - Little correlation between numbers (plot r_{i+1} vs r_i —should fill the plane)
 - Fast
- Typical random number generators return a number in $[0,1)$
 - Should uniformly fill that space
 - Seeds can be used to allow for reproducibility (from one run to the next)



Random Numbers

(Garcia)

- Simple generator: **linear congruential method**

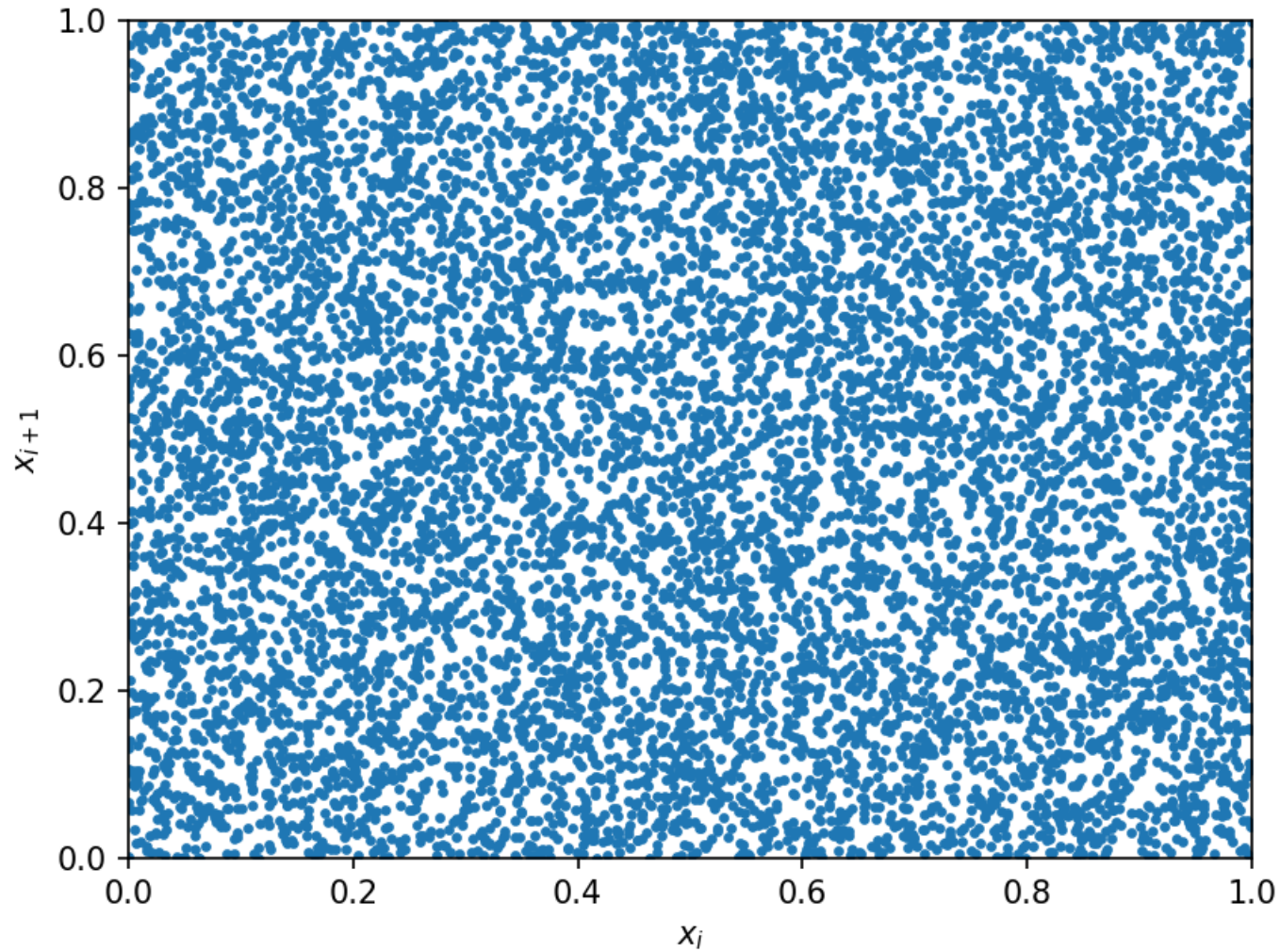
$$I_{\text{new}} = (aI_{\text{old}} + c) \bmod M$$

$$r = I_{\text{new}}/M$$

- Here we pass in a **seed** and 3 parameters
- Typical choices: $a = 7^5$, $c = 0$, $M = 2^{31} - 1$ (Park & Miller)
 - a and M both even would generate only even or odd numbers
- Periodicity of M
 - Note that $2^{31} - 1$ is the largest signed integer for 32-bit integers
 - It is also a Mersenne prime
- Seed allows for reproducibility (don't pick $I_{\text{old}} = 0$!)
- Parallel algorithms require a parallel random number generator
- Physical random number generators exist (e.g. SGI lava lamp)
- **You probably should not write your own...**

Random Numbers

(García)

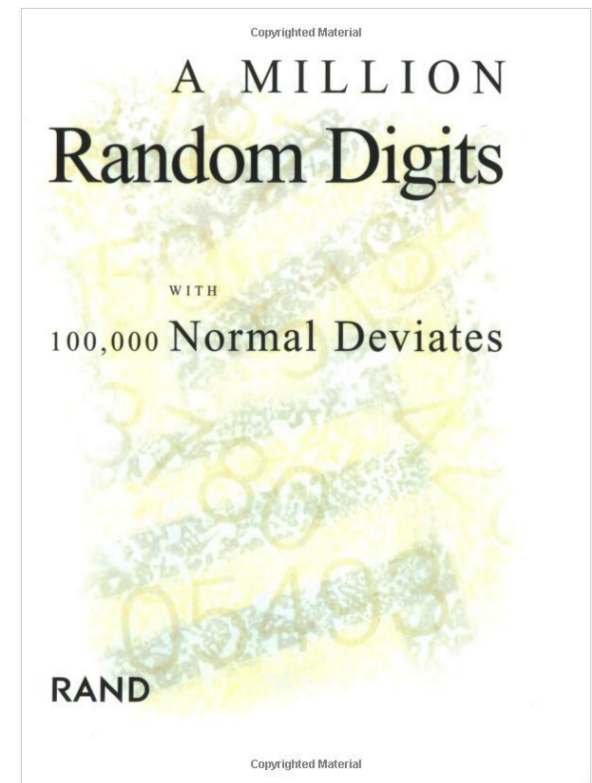


Notice that successive random numbers fill the plane

code: `random_test.py`

Pseudo-Random

- Technically, the linear congruent method generates *pseudo-random* numbers
 - This is probably okay for your application
 - Some places will use hardware (CCD noise, e.g.) or observation to generate true random numbers
 - See: <https://www.random.org/>
 - You can even buy a book on amazon of random numbers:
 - Great comments too: *“Whatever generator they used was not fully tested. The bulk of each page seems random enough. However at the lower left and lower right of alternate pages, the number is found to increment directly.”*
- Cryptographic applications may have more stringent requirements than MC methods



Seeds

- The choice of a seed specifies all of the numbers that follow
- This is useful to debugging—you can ensure that everytime you run your random simulation that you get the same random numbers
- Often you want to “randomize” the seed (e.g., set it to current system time) to get a different outcome each time your program runs
 - Most random number functions do this for you

```
random.seed(a=None, version=2)
```

Initialize the random number generator.

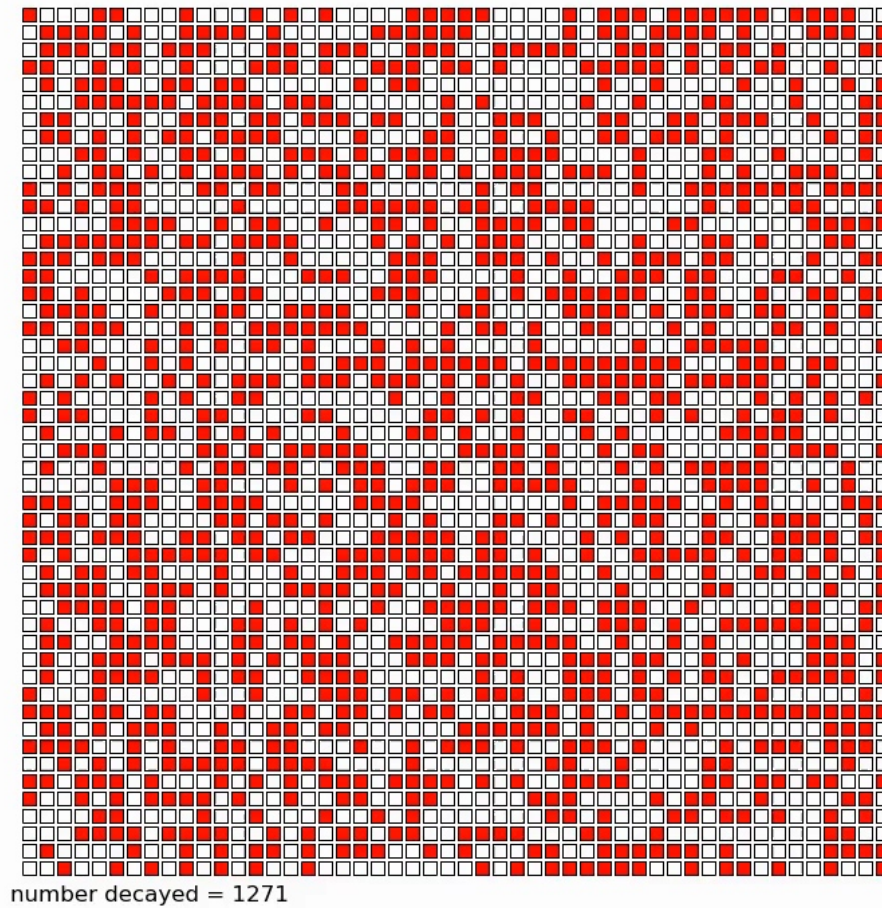
If `a` is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the `os.urandom()` function for details on availability).

If `a` is an `int`, it is used directly.

True Random Number Generators

- Useful in cryptography
- Quantum sources (from Wikipedia):
 - shot noise: QM noise in electric circuit, e.g., light shining on a photodiode
 - nuclear decay is random
 - photons through semi-transparent mirror (reflection and transmission form bits)
- Non-quantum
 - thermal noise from a resistor
 - atmospheric noise detected by radio receiver

Radioactive Decay



Radioactive Decay

(following Newman)

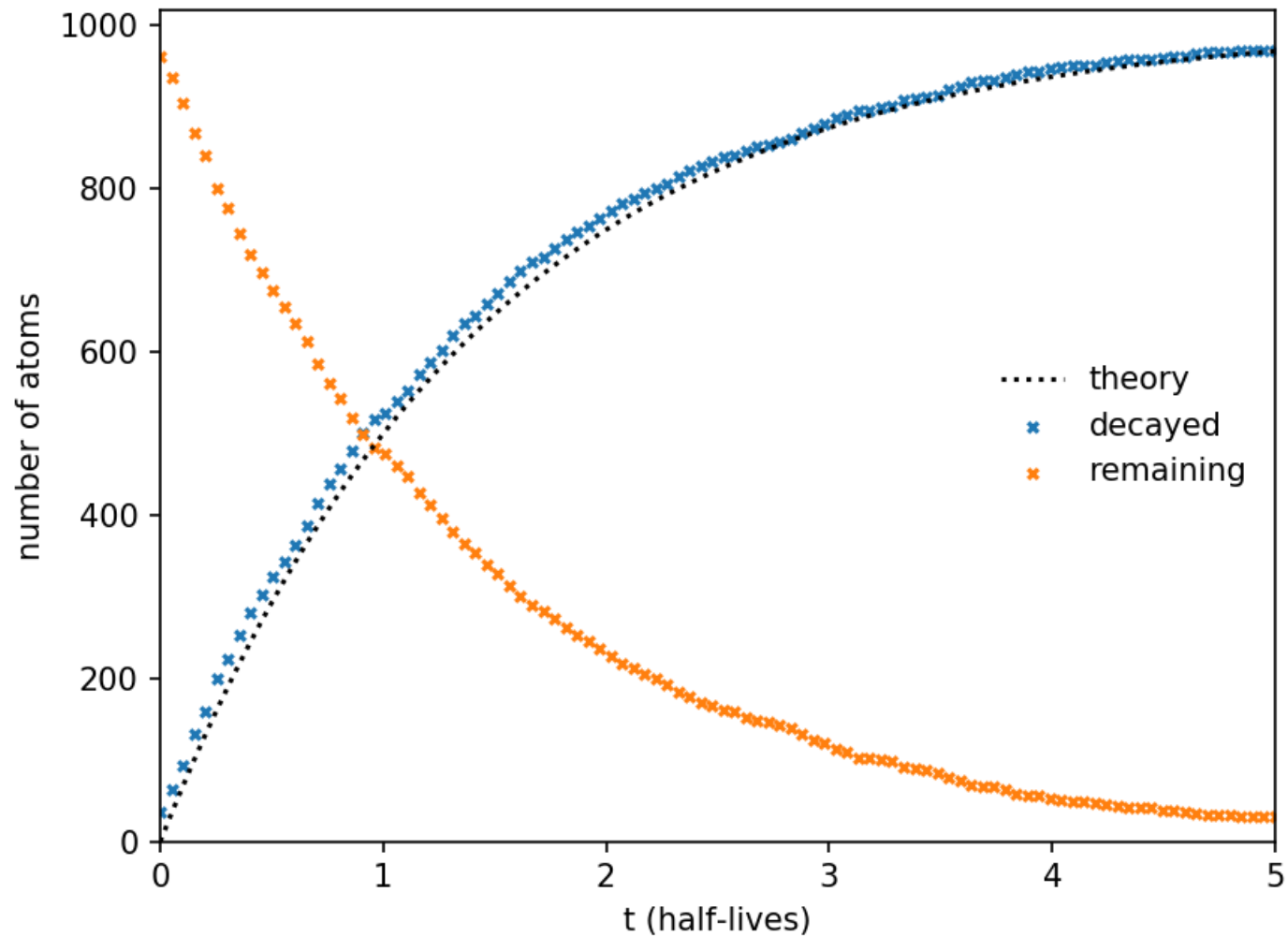
- Consider radioactive decay
 - Parent atoms decay into daughter atoms with some characteristic half-life, $\tau_{1/2}$
 - Number of parent atoms after time t :
$$N(t) = N_0 2^{-t/\tau_{1/2}}$$
 - Probability that a single particular atom has decayed in a time interval of length t :
$$p(t) = 1 - 2^{-t/\tau_{1/2}}$$
- For a small collection of atoms, there will be noise in the decay curve, owing to the randomness of the process

Radioactive Decay

(following Newman)

- Simple decay model:
 - Create a bunch of atoms
 - Divide time up into small intervals, $dt < \tau_{1/2}$
 - Probability that an parent atom that has made it this far will decay in our interval is:
$$p(dt) = 1 - 2^{-dt/\tau_{1/2}}$$
 - For each atom, use a random number generator that yields r in $[0,1)$, and say that we have decayed if $r < p$
 - This works because both p and r are uniform over the same distribution

Radioactive Decay



code: simple_continuous.py

Other Distributions

(following Newman)

- The distribution of decay probability is the following:
 - Probability an atom makes it to time t without decaying is

$$p = 2^{-t/\tau_{1/2}}$$

- Probability that that atom decays in the interval dt following t is:

$$p = 1 - 2^{-dt/\tau_{1/2}} = 1 - e^{-dt \ln 2 / \tau_{1/2}} \approx \frac{\ln 2}{\tau_{1/2}} dt$$

- Total probability distribution $p(t) dt$ for decay between t and $t+dt$ is:

$$p(t)dt = 2^{-t/\tau_{1/2}} \frac{\ln 2}{\tau_{1/2}} dt$$

- First term is probability that we make an atom makes it to time t
 - More efficient to use this distribution to calculate decay of our atoms
 - For 1000 atoms, simply sample this distribution 1000 times

Other Distributions

(following Newman)

- Basic idea:
 - We have a random number generator that gives us numbers with a distribution $q(z) dz$
 - e.g. $q(z)$ is the probability of getting a number between z and $z+dz$
 - For uniform generator, $q(z) = 1$
 - Imagine a function $x(z)$ that maps from our distribution to another
 - Total probability is 1, so we require:

$$p(x)dx = q(z)dz$$

- We want to find the $x(z)$ that gives us the desired $p(x)dx$. Integrate:

$$\int_{-\infty}^{x(z)} p(x')dx' = \int_0^z dz' = z$$

Note that the starting point of the integration will be the first value for which the distribution is non-zero

Other Distributions

(following Newman)

- For radioactive decay, we need:

- On $[0, \infty)$

$$p(x) = \mu e^{-\mu x}, \quad \mu = \frac{\ln 2}{\tau_{1/2}}$$

- Our transformation function is then:

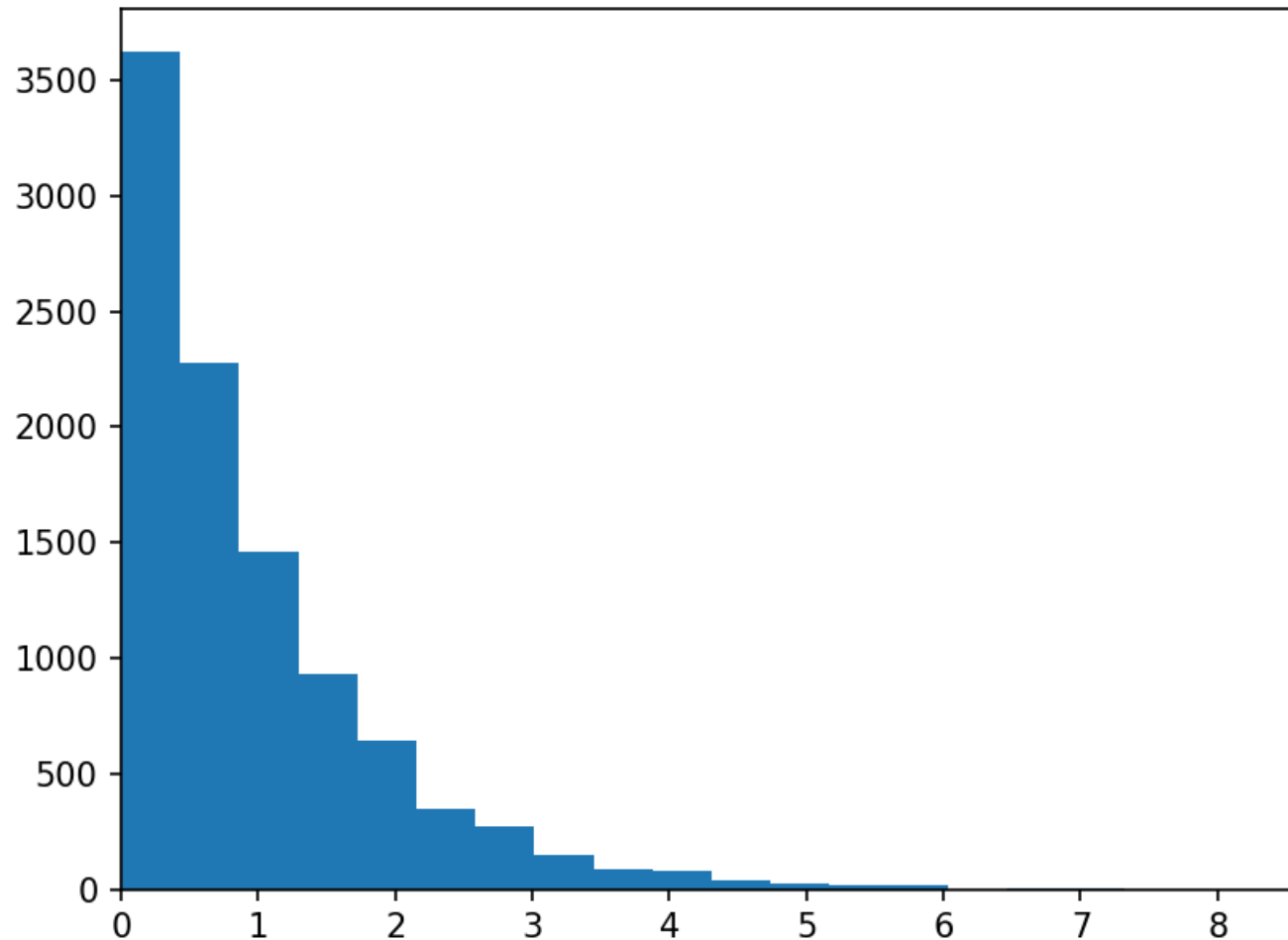
$$\mu \int_0^{x(z)} e^{-\mu x'} dx' = 1 - e^{-\mu x} = z$$

$$x(z) = -\frac{1}{\mu} \ln(1 - z)$$

- This requires only a single random number for each atom
 - Let's look at the code (`transform_random.py`)

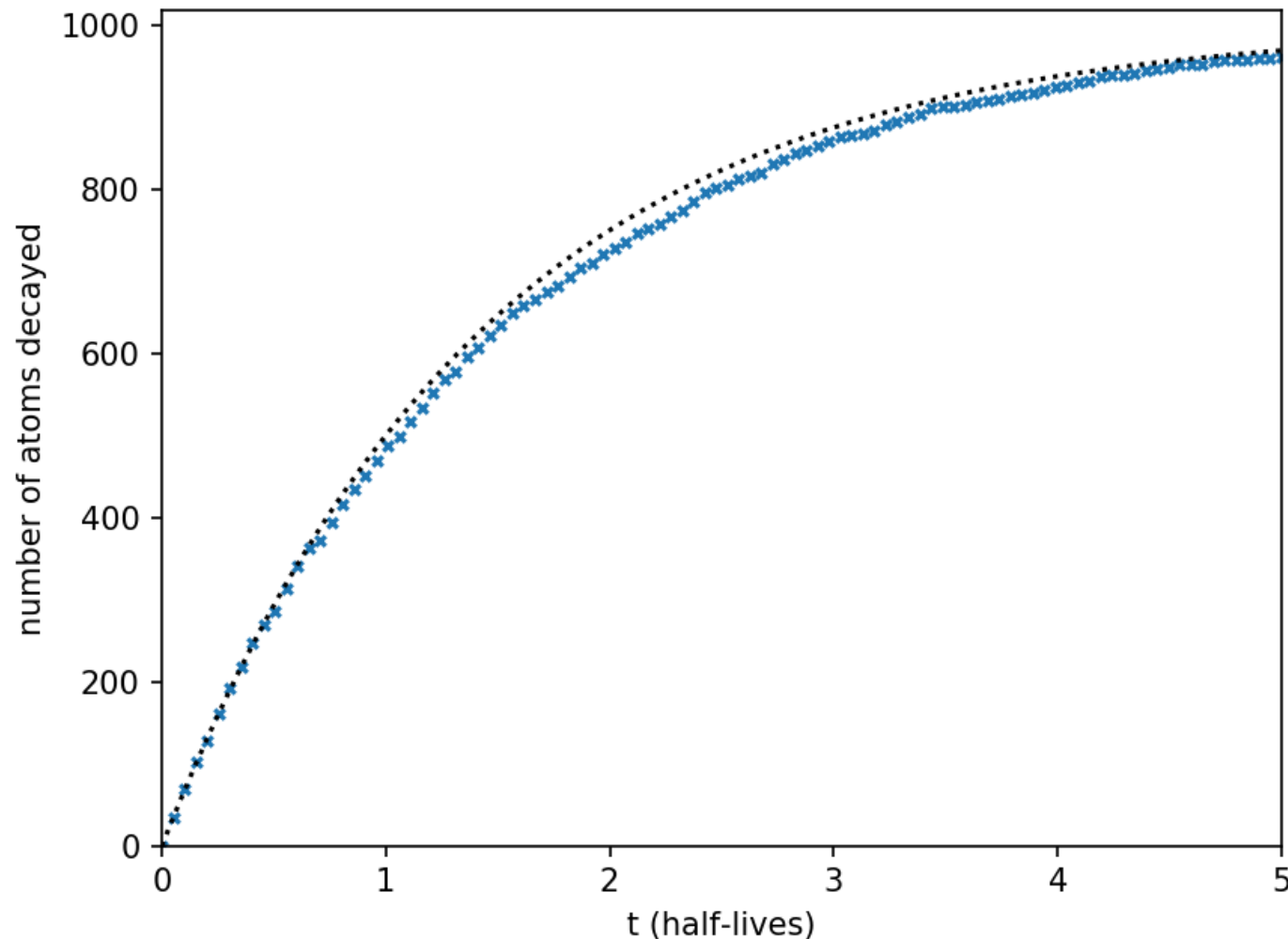
Other Distributions

- Here's the random numbers from our new exponential distribution (for $\mu = 1$)



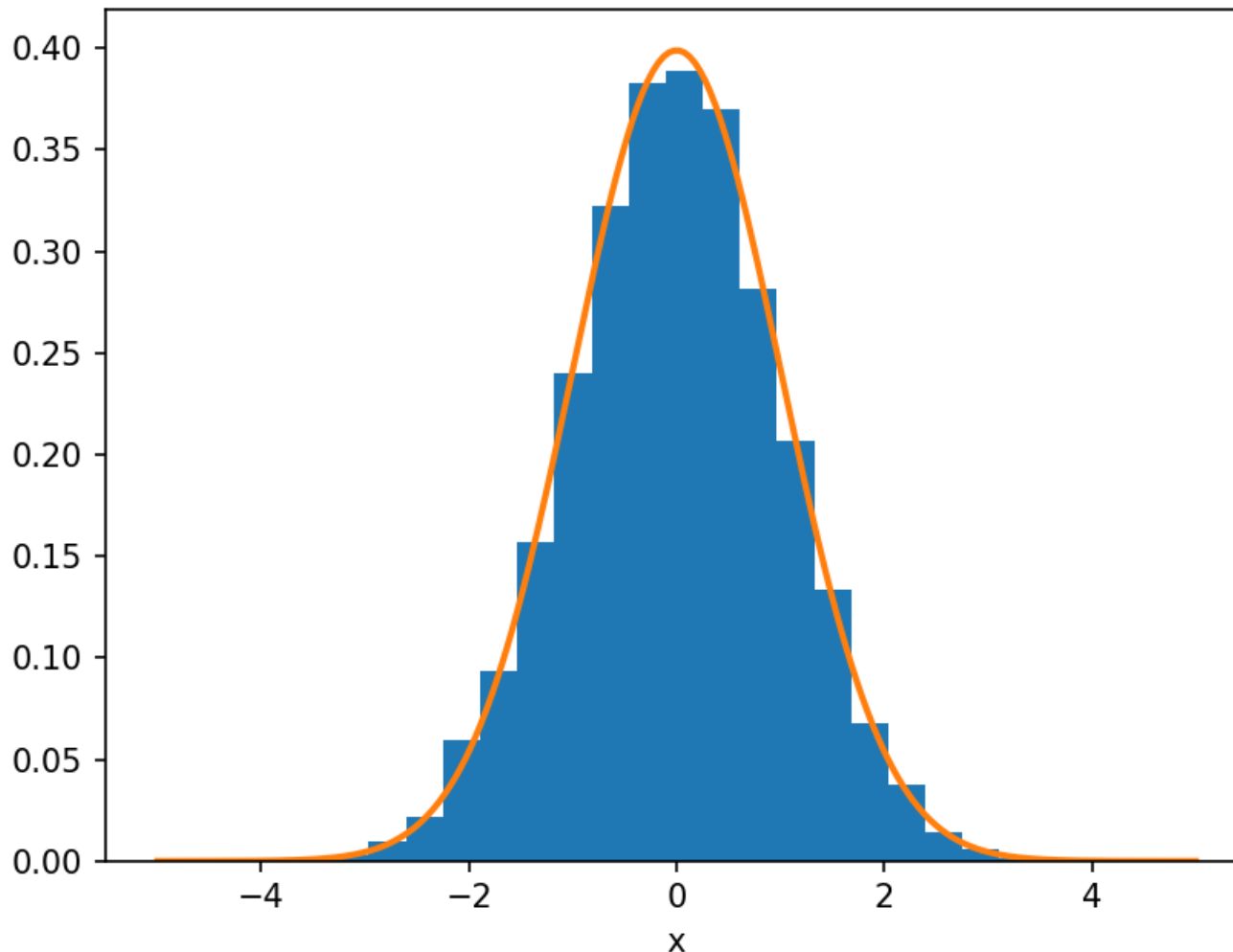
Other Distributions

- And here's the number of atoms decayed vs. time using this new distribution



Other Distributions

- Many other distributions are possible. We already saw the Gaussian normal distribution when we did fitting



Note: we cannot derive this using the simple transformation seen here. Instead, there is a technique called the Box-Muller transformation that is commonly used.

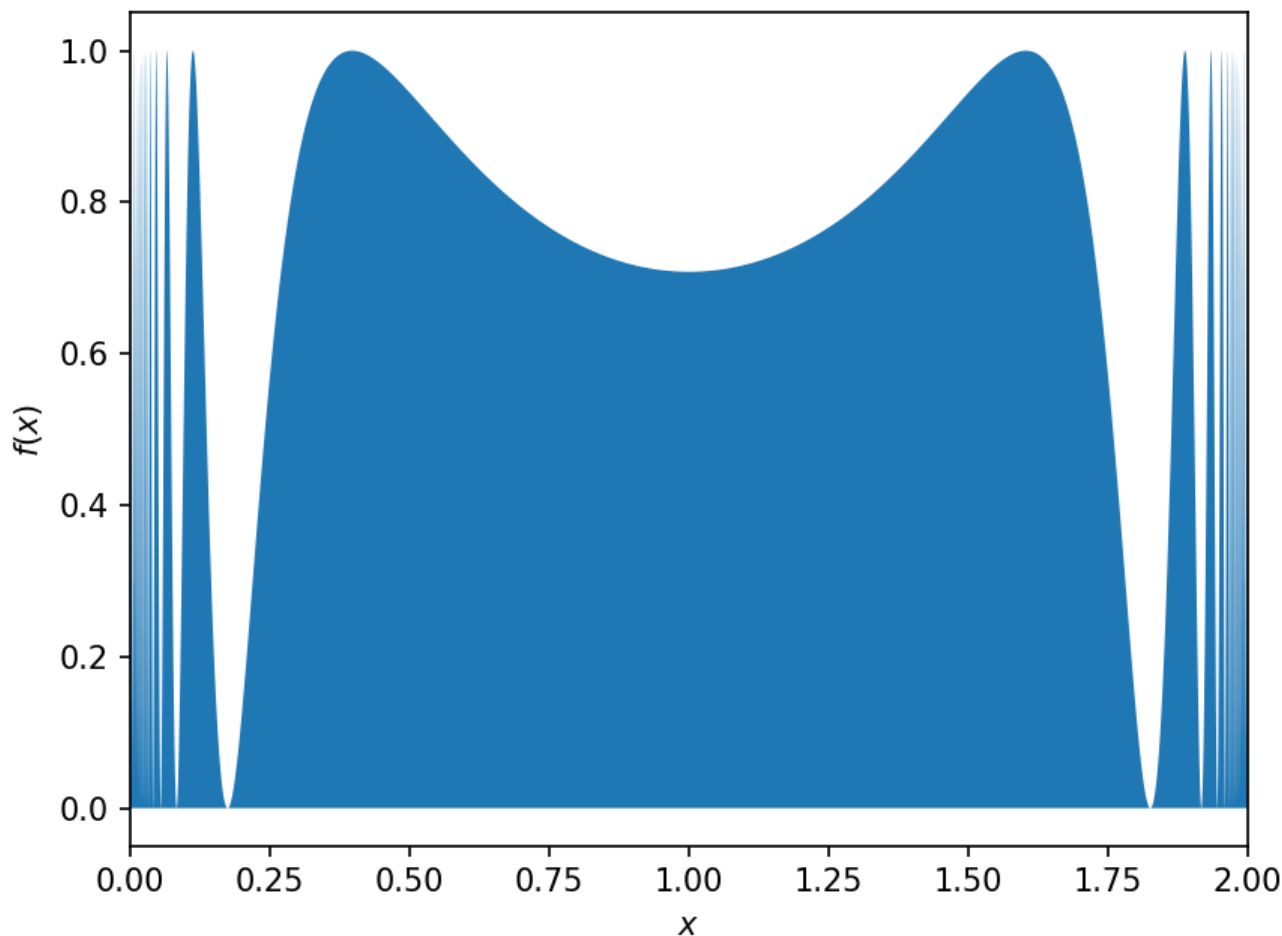
Monte Carlo Integration

(following Newman)

- Consider an integrand with a lot of structure, e.g.,

$$I = \int_0^2 \sin^2 \left(\frac{1}{x(2-x)} \right) dx$$

The integral is just the area under the curve—we see that it is bounded by the area of our plot rectangle, 2



Monte Carlo Integration

- Trying Gaussian quadrature (through `scipy.integrate.quadrature`):
 - `/usr/lib64/python2.7/sitepackages/scipy/integrate/quadrature.py:183: AccuracyWarning: maxiter (50) exceeded. Latest difference = 5.665988e-02 AccuracyWarning)`
 - It gives $I = 1.43798143456$ with an estimate of the error as 0.0566598755277
 - We're used to Gaussian quadrature giving really nice accuracy, but this doesn't do that well
- Not that much better with ODEPACK (gives 1.45168775052 ; estimates the error as 0.0027)
- For a complex integrand like this, Monte Carlo methods can do well

Monte Carlo Integration

(following Newman)

- Basic idea of Monte Carlo integration

- Consider:

$$I = \int_a^b f(x) dx$$

- We need to know the bounds of $f(x)$ in $[a,b]$. Let's take them to be $[0, F]$
 - $A = (b - a) F$ is the area of the smallest rectangle that contains the function we are integrating
 - This is a crude estimate of the integral, I
 - Randomly pick pairs of points, (x_i, y_i) , that fall in our bounding rectangle
 - Define q as the fraction of points that fall below the curve, e.g., $y_i < f(x_i)$
 - Our integral is then simply:

$$q = \frac{I}{A}$$

Monte Carlo Integration

- Note that this converges very slowly
 - Error drops as \sqrt{N}
- Ex: for our test integral, we find

N	I
1000	1.428
10000	1.4474
100000	1.45584
1000000	1.453152

- Recall Trapezoid was $O(N^{-2})$ and Simpson's was $O(N^{-4})$

code: MC_integral.py

Mean Value Integration

- There is a slightly better method that is easier to generalize
- Consider the average value of a function:

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) dx = \frac{I}{b-a}$$

- we can compute the average of f by random sampling and then get the value of the integral from this:

$$I = (b-a)\langle f \rangle \approx \frac{b-a}{N} \sum_{i=1}^N f(x_i)$$

- here the x_i are points randomly sampled in $[a, b]$

Mean Value Integration

- Results of the same integral with mean-value integration:

N	I
1000	1.4215540209
10000	1.44738373721
100000	1.45043659232
1000000	1.45149071316

- note: you'll get different results when you run this each time—this gives a guide of your error

code: `mean_value_integral.py`

Multidimensional Integrals

(following Newman)

- Consider a hypersphere with radius 1 defined on $[-1,1]^d$, where d is the dimensionality
 - We could do this using the methods we learned earlier, like trapezoid rule
 - This would require us discretizing on a d -dimensional grid—a lot of points and a lot of function evaluations
 - For $d \geq 4$, Monte Carlo integration can be a win
- Extension of mean-value integration:

$$I \sim \frac{V}{N} \sum_{i=1}^N f(\mathbf{r}_i)$$

- Here, V is the volume we are sampling over and \mathbf{r}_i is a randomly sampled point in V

Multidimensional Integrals

(following Newman)

- The analytic volume for a hypersphere is:

$$V_d(R) = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2} + 1)} R^d$$

- Our MC results:

10-d hypersphere volume:

N: 10, V = 0.0

N: 100, V = 0.0

N: 1000, V = 3.072

N: 10000, V = 2.1504

N: 100000, V = 2.47808

N: 1000000, V = 2.505728

analytic value = 2.55016403988

- Since this uses random sampling, you can get a feel for the error by running it multiple times with the same N

Importance Sampling

(following Newman)

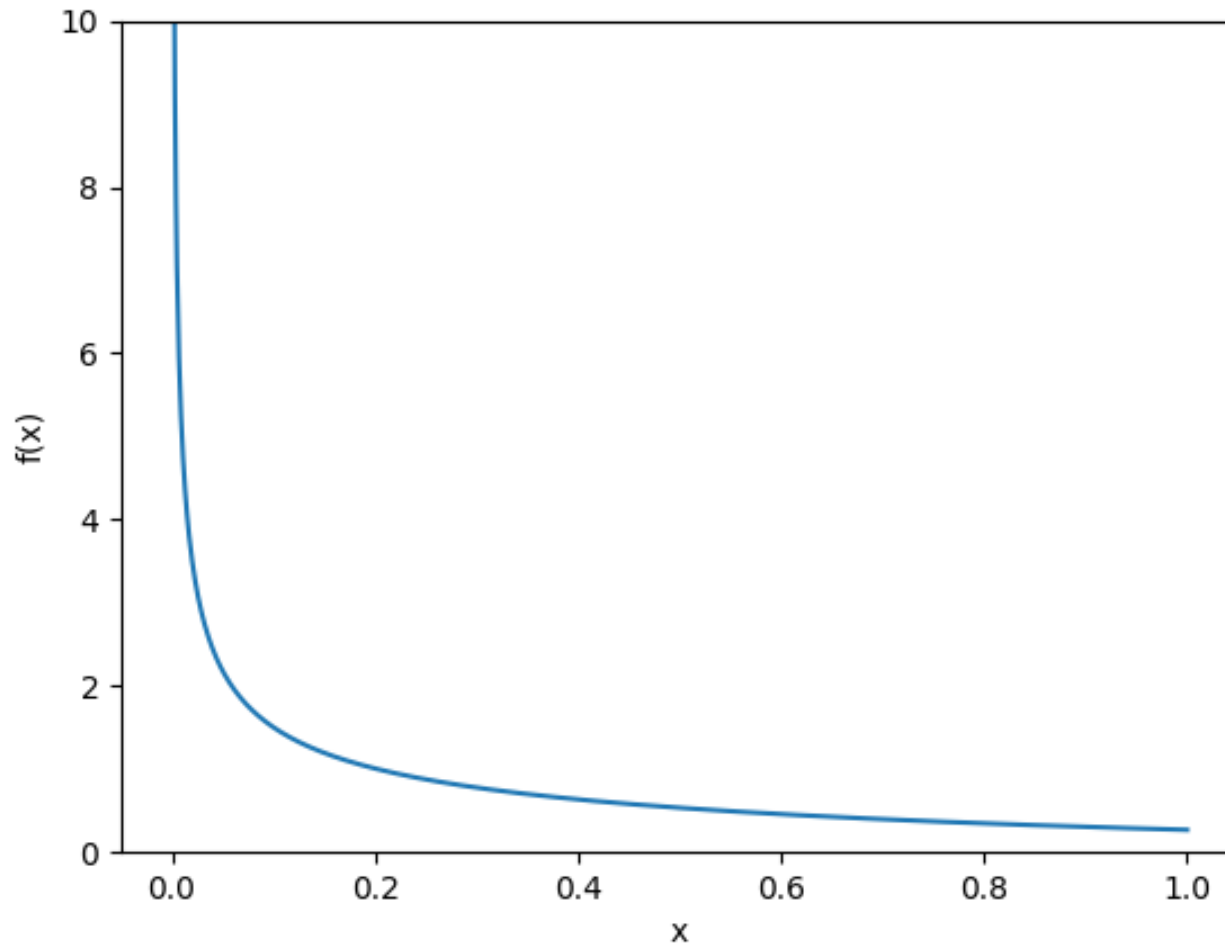
- For functions with complex structures, we might want to do better than picking random points uniformly
 - A constant function needs only a single point to get an exact result
 - A general function doesn't need many samples in its smooth, slowly varying parts
- Consider:

$$I = \int_0^1 \frac{x^{-1/2}}{e^x + 1} dx$$

- Integrand blows up near the origin, but integral is finite
- Just directly sampling this will give unreliable results (very large differences from one realization to next)

Importance Sampling

(following Newman)



$$I = \int_0^1 \frac{x^{-1/2}}{e^x + 1} dx$$

Importance Sampling

(following Newman)

- Results of sampling from a uniform distribution

N: 100000, I = 0.832311684886

N: 100000, I = 0.851119851297

N: 100000, I = 0.865786350986

N: 100000, I = 0.847456828921

N: 100000, I = 0.831354748327

- We see that the error is in the 2nd decimal place using 100000 samples

Importance Sampling

(following Newman)

- Importance sampling helps to remove pathologies
- Consider the weighted average:

$$\langle g \rangle_w \equiv \frac{\int_a^b w(x)g(x)dx}{\int_a^b w(x)dx}$$

– Here $w(x)$ is the weight function

- Factor $w(x)$ out of our function:

$$\left\langle \frac{f(x)}{w(x)} \right\rangle_w = \frac{\int_a^b w(x)f(x)/w(x)dx}{\int_a^b w(x)dx} = \frac{\int_a^b f(x)dx}{\int_a^b w(x)dx} = \frac{I}{\int_a^b w(x)dx}$$

$$\therefore I = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \int_a^b w(x)dx$$

Importance Sampling

(following Newman)

- Now we choose the weighted average to correspond to a probability distribution:

$$p(x) = \frac{w(x)}{\int_a^b w(x) dx}$$

- This has the property that: $\int_a^b p(x) dx = 1$
- Probability of getting a point in $[x, x+dx]$ is $p(x) dx$
 - Taking N samples from $p(x)$, then number of samples in $[x, x+dx]$ is $N p(x) dx$
 - This means that if we draw an x_i from $p(x)$ distribution, then we can express:

$$\sum_{i=1}^N g(x_i) \approx \int_a^b N p(x) g(x) dx$$

- Important to remember that x_i are sampled from $p(x)$

Importance Sampling

(following Newman)

- We can write our weighted average as:

$$\langle g \rangle_w = \frac{\int_a^b w(x)g(x)dx}{\int_a^b w(x)dx} = \int_a^b \left[\frac{w(x)}{\int_a^b w(x)dx} \right] g(x)dx$$

$$= \int_a^b p(x)g(x)dx \approx \frac{1}{N} \sum_{i=1}^N g(x_i)$$

- Where x_i are sampled from $p(x)$

Importance Sampling

(following Newman)

- Our integral was:

$$I = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \int_a^b w(x) dx$$

- Using this new probability distribution, our integral is:

$$I \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{w(x_i)} \int_a^b w(x) dx$$

Keep in mind,
we need to
sample the x_i in
our distribution,
 $p(x)$

- We choose $w(x)$ to remove pathologies in our integrand

- Note: $w(x) = 1$ gives us mean-value integration
- For

$$I = \int_0^1 \frac{x^{-1/2}}{e^x + 1} dx$$

we pick $w(x) = x^{-1/2}$

- We need to then generate random numbers obeying our distribution function

Importance Sampling

(following Newman)

- For our integral, we want $w(x) = x^{-1/2}$
- Our PDF is:

$$p(x) = \frac{w(x)}{\int_0^1 w(x) dx} = \frac{x^{-1/2}}{2x^{1/2}|_0^1} = \frac{1}{2\sqrt{x}}$$

- Use the transformation method to get random numbers obeying this distribution

$$p(x)dx = q(z)dz = dz$$

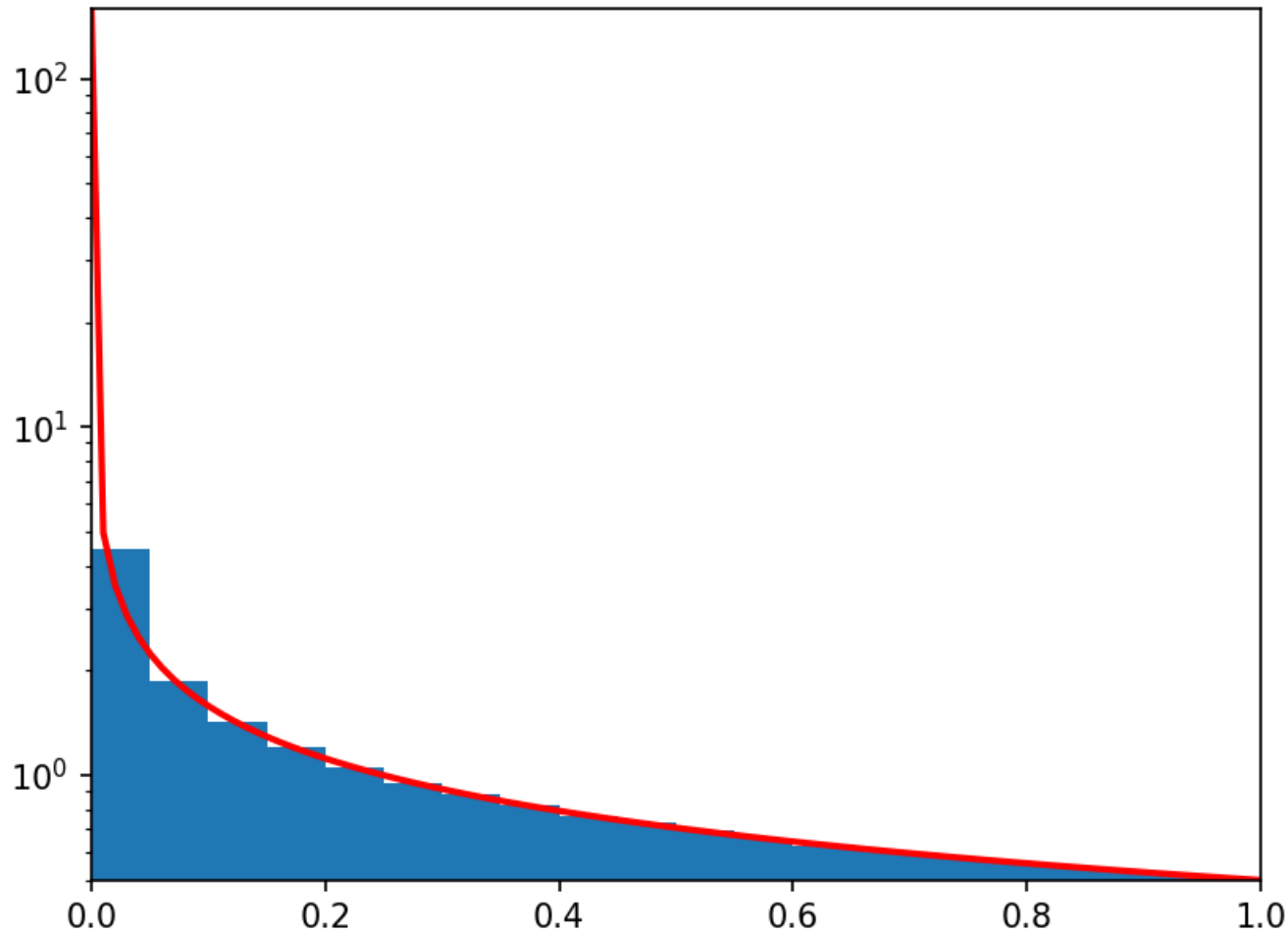
- $q(z)$ is the uniform distribution $[0,1)$ we know

$$\int_0^{x(z)} p(x') dx' = \int_0^z dz' = z$$

$$\frac{1}{2} \int_0^{x(z)} (x')^{-1/2} = z \rightarrow x = z^2$$

Importance Sampling

(following Newman)



Sampling

$$p(x) = \frac{1}{2\sqrt{x}}$$

Importance Sampling

(following Newman)

- Sampling $p(x)$, we have

$$I \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{w(x_i)} \cdot \int_a^b w(x) dx = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{x_i^{-1/2}} \cdot 2$$

- Results:

N: 100000, I = 0.839581975542

N: 100000, I = 0.838399197498

N: 100000, I = 0.839107731663

N: 100000, I = 0.83821922173

N: 100000, I = 0.838637047804

- Now the variation is in the 3rd decimal place for the same number of samples

Monte Carlo Simulation: Poker

- Random numbers can also be used to create realizations of a system to study statistical behavior / probabilities
- Consider the odds of getting a particular hand in straight poker
 - Dealt 5 cards
 - Ranking of hands:
 - One pair (two of the same rank)
 - Two pair (two pairs)
 - Three of a kind (three of the same rank)
 - Straight (5 cards in a sequence of rank, regardless of suit)
 - Flush (5 cards of the same suit)
 - Full house (three of a kind + one pair)
 - Four of a kind (four cards of the same rank)
 - Straight flush (a straight that is also a flush)

Monte Carlo Simulation: Poker

- Monte Carlo calculation of the odds:
 - Do N times:
 - Create a deck
 - Shuffle
 - Draw 5 cards
 - Check if you have any hands







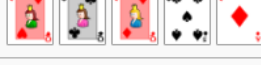



Monte Carlo Simulation: Poker

Number of hands: 100000

Straight Flush: (2) 2e-05
 Four of a kind: (23) 0.00023
 Full House: (149) 0.00149
 Flush: (220) 0.0022
 Straight: (342) 0.00342
 Three of a kind: (2048) 0.02048
 Two pair: (4769) 0.04769
 One pair: (42037) 0.42037

Number of hands: 1000000

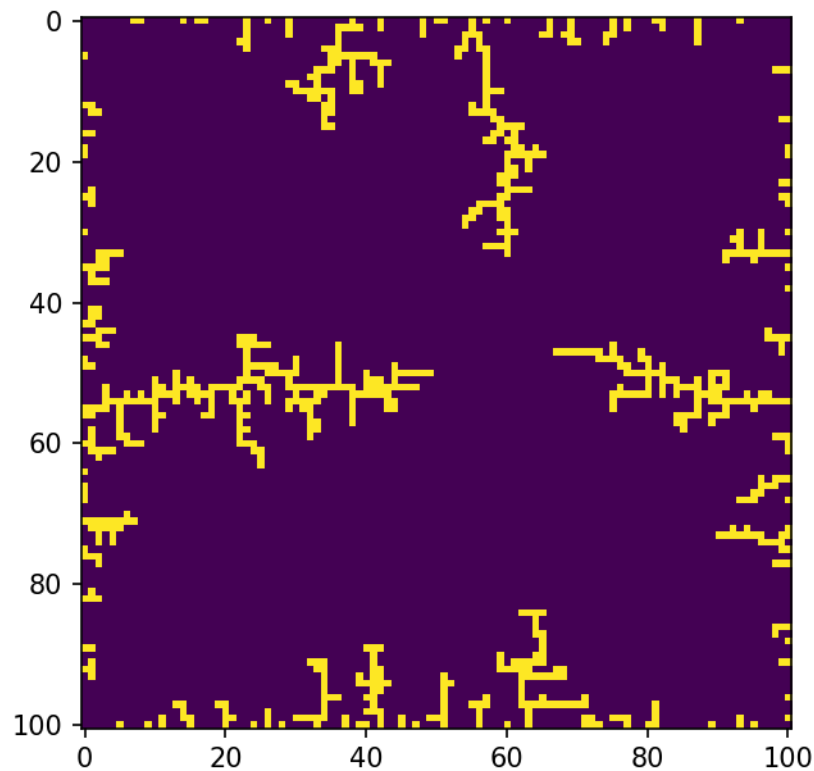
Straight Flush: (17) 1.7e-05
 Four of a kind: (225) 0.000225
 Full House: (1478) 0.001478
 Flush: (1972) 0.001972
 Straight: (3563) 0.003563
 Three of a kind: (21327) 0.021327
 Two pair: (47761) 0.047761
 One pair: (422944) 0.422944

Hand	Distinct Hands	Frequency	Probability	Cumulative probability	Odds	Mathematical expression of absolute frequency
Royal flush 	1	4	0.000154%	0.000154%	649,739 : 1	$\binom{4}{1}$
Straight flush (excluding royal flush) 	9	36	0.00139%	0.0014%	72,192 : 1	$\binom{10}{1}\binom{4}{1} - \binom{4}{1}$
Four of a kind 	156	624	0.0240%	0.0256%	4,164 : 1	$\binom{13}{1}\binom{12}{1}\binom{4}{1}$
Full house 	156	3,744	0.1441%	0.17%	693 : 1	$\binom{13}{1}\binom{4}{3}\binom{12}{1}\binom{4}{2}$
Straight (excluding royal flush and straight flush) 	1,277	5,108	0.1965%	0.367%	508 : 1	$\binom{13}{5}\binom{4}{1} - \binom{10}{1}\binom{4}{1}$
Flush (excluding royal flush and straight flush) 	10	10,200	0.3925%	0.76%	254 : 1	$\binom{10}{1}\binom{4}{1}^5 - \binom{10}{1}\binom{4}{1}$
Three of a kind 	858	54,912	2.1128%	2.87%	46.3 : 1	$\binom{13}{1}\binom{4}{3}\binom{12}{2}\binom{4}{1}^2$
Two pair 	858	123,552	4.7539%	7.62%	20.0 : 1	$\binom{13}{2}\binom{4}{2}^2\binom{11}{1}\binom{4}{1}$
One pair 	2,860	1,098,240	42.2569%	49.9%	1.37 : 1	$\binom{13}{1}\binom{4}{2}\binom{12}{3}\binom{4}{1}^3$
No pair / High card 	1,277	1,302,540	50.1177%	100%	0.995 : 1	$\left[\binom{13}{5} - 10\right] \left[\binom{4}{1}^5 - 4\right]$
Total	7,462	2,598,960	100%	---	0 : 1	$\binom{52}{5}$

code: poker.py

Monte Carlo Simulation: DLA

- Drop a particle onto a lattice. It diffuses via a random walk
- It “sticks” when it hits a wall or another particle



Markov Chain Monte Carlo

(following Newman)

- We'll use statistical mechanics as our motivation
- Example: compute expectation values of a system in equilibrium with temperature T

- Probability of occupying state w/ energy E_i :

$$P(E_i) = \frac{e^{-\beta E_i}}{Z} \quad Z = \sum_i e^{-\beta E_i} \quad \beta = (k_B T)^{-1}$$

- The partition function, Z , is the normalization—this is summed over all the states in the system
- This sum can be really large (consider the number of molecules in the air in this room)
- Expectation value of quantity X with value X_i in state i is:

$$\langle X \rangle = \sum_i X_i P(E_i)$$

Markov Chain Monte Carlo

(following Newman)

- We will approximate this *sum* using the same sampling techniques we did with *integrals*
 - For integrals, we randomly picked points to evaluate the function at
 - For the sum, we will randomly pick terms in the sum

$$\langle X \rangle \approx \frac{\sum_{k=1}^N X_k P(E_k)}{\sum_{k=1}^N P(E_k)}$$

- Here, k is the index over our sample of N states
- The denominator appears because we are summing only a subset of states, so $\sum_k P(E_k) \neq 1$
- Note that $P(E_i \gg k T) \ll 1$ —a lot of our random samples will be terms that don't really contribute to the sum
 - We need importance sampling—focus on the most probable states

Markov Chain Monte Carlo

(following Newman)

- Method mirrors our approach with integration
 - Define weighted average:

$$\langle g \rangle_w = \frac{\sum_i w_i g_i}{\sum_i w_i}$$

- Note: here we are summing over all states
 - Pick the $g_i = X_i P(E_i)/w_i$

$$\left\langle \frac{X_i P(E_i)}{w_i} \right\rangle_w = \frac{\sum_i w_i X_i P(E_i)/w_i}{\sum_i w_i} = \frac{\sum_i X_i P(E_i)}{\sum_i w_i} = \frac{\langle X \rangle}{\sum_i w_i}$$

- Our expectation value is then

$$\langle X \rangle = \left\langle \frac{X_i P(E_i)}{w_i} \right\rangle_w \sum_i w_i$$

- So far, this involves *all* the states in each term

Markov Chain Monte Carlo

(following Newman)

- We will do the weighted average by taking N samples from a probability distribution function (as we did previously)

- Probability of being in state i

$$p_i = \frac{w_i}{\sum_j w_j}$$

- Our weighted average is:

$$\langle g \rangle_w \sim \frac{1}{N} \sum_{k=1}^N g_k$$

- Where here states k are sampled according to our distribution function

- Final result:

$$\langle X \rangle = \frac{1}{N} \sum_{k=1}^N \frac{X_k P(E_k)}{w_k} \sum_i w_i$$

- Note that the sums are over different sets of states here!

Markov Chain Monte Carlo

- Look at this result:

$$\langle X \rangle = \frac{1}{N} \sum_{k=1}^N \frac{X_k P(E_k)}{w_k} \sum_i w_i$$

- and compare with the result we had with importance sampling

$$I \approx \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{w(x_i)} \int_a^b w(x) dx$$

- the only real difference is that we now have a sum over w_i instead of an integral—this is because we were originally dealing with sums instead of integrals

Markov Chain Monte Carlo

(following Newman)

- We choose our weights to be the probabilities from the Boltzmann distribution:

$$w_i = P(E_i)$$

- The second sum is now trivial:

$$\sum_i w_i = 1$$

- Expectation value is:

$$\langle X \rangle = \frac{1}{N} \sum_{k=1}^N X_k$$

- Here, this sum is over states sampled in proportion to $P(E_i)$
- Remaining difficulty: we don't know how to sample with the probability distribution we want
 - In particular, we'd need to compute the partition function

Markov Chain Monte Carlo

- Markov chain:
 - Generate a sequence of states where the next state only depends on the current state
- Example (from Wikipedia)
 - Consider weather prediction

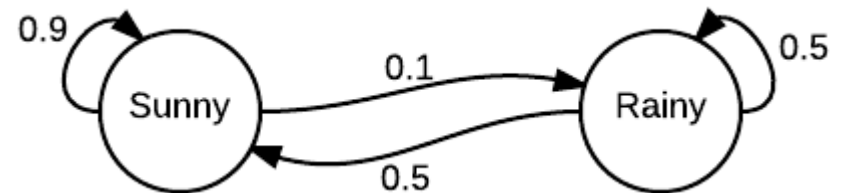
- Directed-graph shows the probability of a sunny or rainy day based on the current weather
- As a transition matrix:

$$\mathbf{T} = \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix}$$

- Start with a sunny day—what is tomorrow?

$$\mathbf{x}^{(0)} = (1 \ 0)$$

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} \mathbf{T} = (1 \ 0) \begin{pmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{pmatrix} = (0.9 \ 0.1)$$



Markov Chain Monte Carlo

- Weather for some arbitrary day:

- $\mathbf{x}^{(n)} = \mathbf{x}^{(n-1)}\mathbf{T}$

- $\mathbf{x}^{(n)} = \mathbf{x}^{(0)}\mathbf{T}^n$

Markov Chain Monte Carlo

- MCMC is a technique for integration using importance sampling where we don't draw random numbers directly from the probability distribution function for our model
- Instead we use a *Markov chain* to go from one state (random number) to the next

Markov Chain Monte Carlo

(following Newman)

- We'll use a Markov chain to walk from one state to the next
 - Start with state i
 - Generate next state by making a small change (e.g. changing the state of one molecule by one level)
- Eliminates the need to create a random number generator for the Boltzmann probability distribution
- Transition probabilities generate choice of new state:
 - T_{ij} : probability of transition from state i to j
 - Require:

$$\sum_j T_{ij} = 1$$

This just says that if we start on i we have a 100% chance of going to another state

$$\frac{T_{ij}}{T_{ji}} = \frac{P(E_j)}{P(E_i)} = \frac{e^{-\beta E_j} / Z}{e^{-\beta E_i} / Z} = e^{-\beta(E_j - E_i)}$$

This is called detailed balance in Stat Mech

Markov Chain Monte Carlo

(Landau & Paez)

- T_{ij}/T_{ji} is the relative probability, we can write this as:

$$\Delta P = e^{-\beta \Delta E}$$

- Metropolis algorithm:
 - If $\Delta E < 0$, then the change from $i \rightarrow j$ results in a lower energy state
 - $\Delta P > 1 \rightarrow$ always accept the change
 - If $\Delta E > 0$, then the change from $i \rightarrow j$ results in a higher energy state
 - Thermodynamics says that the probability of this change is $0 < \Delta P < 1$
 - Get a random number, r , in $[0, 1)$
 - Accept the change if $\Delta P > r \rightarrow$ this means that we accept those changes where the probability is closest to 1 more often than those closer to 0

Markov Chain Monte Carlo

(following Newman)

- Two important features of this Markov chain
 - The Boltzmann distribution is a fixed point of the Markov chain—if we are in a Boltzmann distribution then we stay on it
 - Markov chain will always converge to the Boltzmann distribution—although it might take a while
- Metropolis-Hastings algorithm
 - Create a *move set* that is the set of possible changes from the current state to the next
 - Pick a move randomly from the move set
 - Accept the move with a probability:

$$P_a = \begin{cases} 1 & E_j \leq E_i \\ e^{-\beta(E_j - E_i)} & E_j > E_i \end{cases}$$

- Do nothing if the move is rejected

Markov Chain Monte Carlo

(following Newman)

- This can be show to meet our requirements on the transition probability matrix
- Physically: we accept a move always if it decreases the energy, and with some finite probability if the energy increases
- Requirements:
 - In calculating expectation value, you need to count even those states when you don't move
 - Number of possible moves in move set going from $i \rightarrow j$ needs to be the same as number in $j \rightarrow i$
 - Every state must be accessible (move set is *ergodic*)

Example: Ideal Gas

(following Newman)

- Consider an idea gas of N atoms
- Quantum states of particle in a box:

$$E = \sum_i^N E(n_x^{(i)}, n_y^{(i)}, n_z^{(i)}) \quad E(n_x, n_y, n_z) = \frac{\pi^2 \hbar^2}{2mL^2} (n_x^2 + n_y^2 + n_z^2)$$

- What is the internal energy of the gas?
 - Move set: move a single atom by +/- 1 in one of its quantum numbers
 - Change in energy from move:

$$\Delta E = \frac{\pi^2 \hbar^2}{2mL^2} \{ [(n_x + 1)^2 + n_y^2 + n_z^2] - (n_x^2 + n_y^2 + n_z^2) \} = \frac{\pi^2 \hbar^2}{2mL^2} (2n_x + 1)$$

for +1 change,

$$\Delta E = \frac{\pi^2 \hbar^2}{2mL^2} \{ [(n_x - 1)^2 + n_y^2 + n_z^2] - (n_x^2 + n_y^2 + n_z^2) \} = \frac{\pi^2 \hbar^2}{2mL^2} (-2n_x + 1)$$

for -1 change

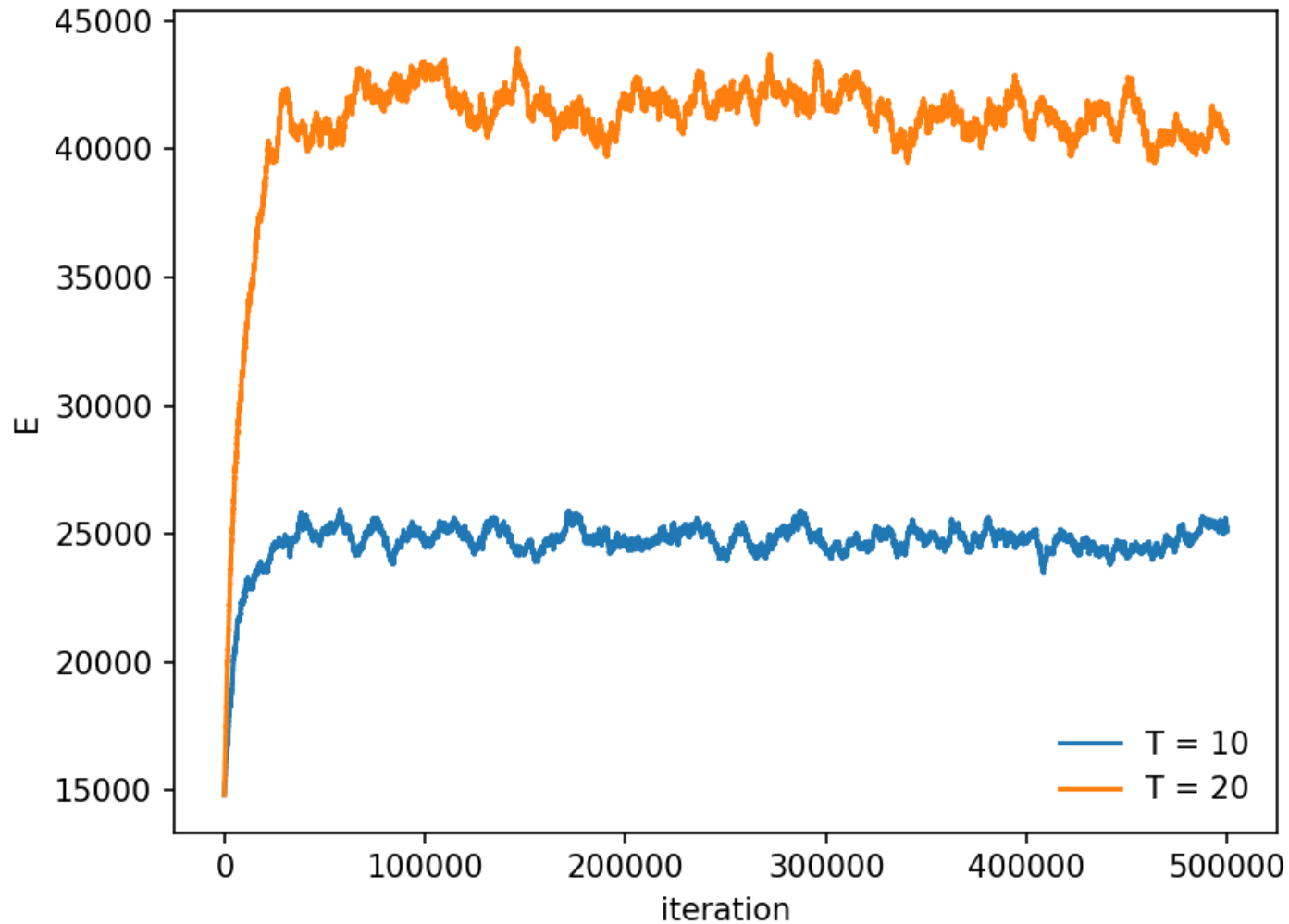
Example: Ideal Gas

(following Newman)

- Take $N = 1000$ particles, $k_B T = 10$, $m = \hbar = 1$
- Starting state doesn't matter—we'll put everything in the ground state
- If we are in $n = 1$ and decrease, then we reject the move

Example: Ideal Gas

(following Newman)



code: ideal_gas.py

Ising Model

- Ising model for ferromagnetism
 - Consider periodic lattice with atoms of +1 or -1 spin
 - Atoms interact with nearest neighbors
 - All aligned = strongly magnetic
 - Energy is minimized when spins are aligned
 - Above a critical temperature, magnetization goes away
- Energy of the system is:

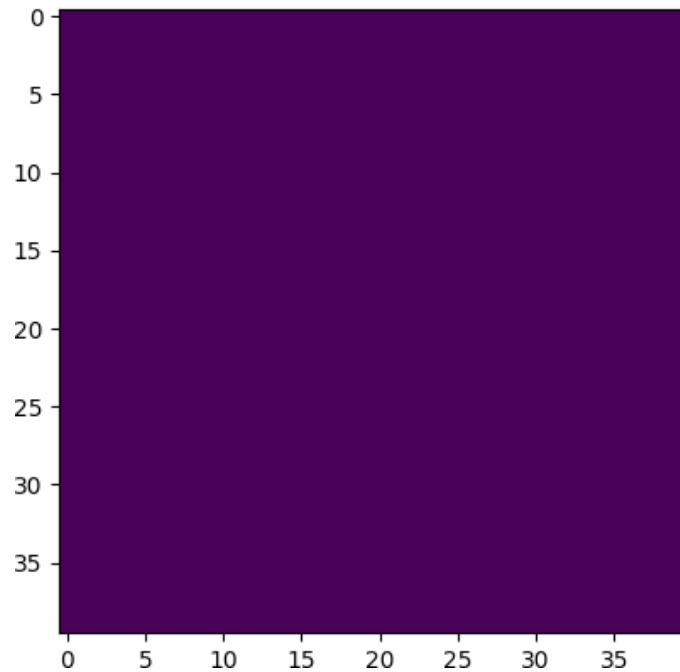
$$E = -J \sum_{\langle ij \rangle} s_i s_j$$

- here the sum is done over pairs adjacent on the lattice (4 neighbors)

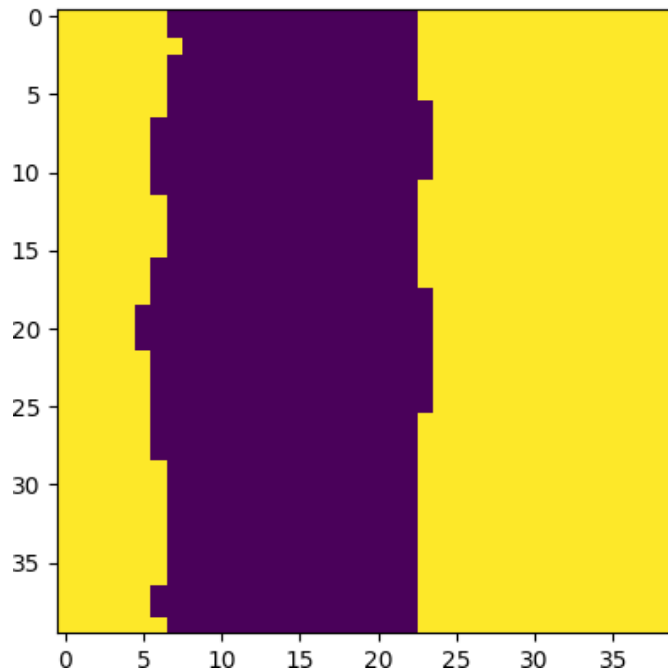
Ising Model

- As temperature is increased, magnetization goes away

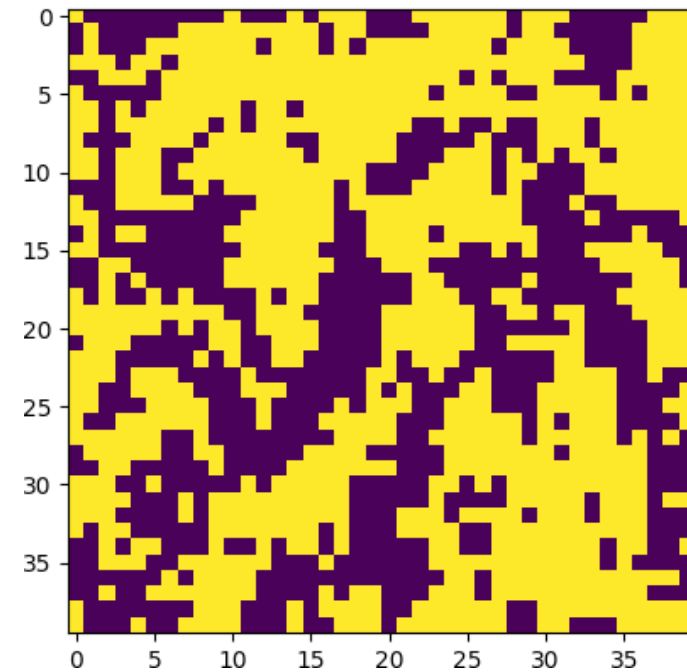
$T = 0.50$; $E = -3200.0$, $M = -1600.0$



$T = 0.90$; $E = -3012.0$, $M = 250.0$



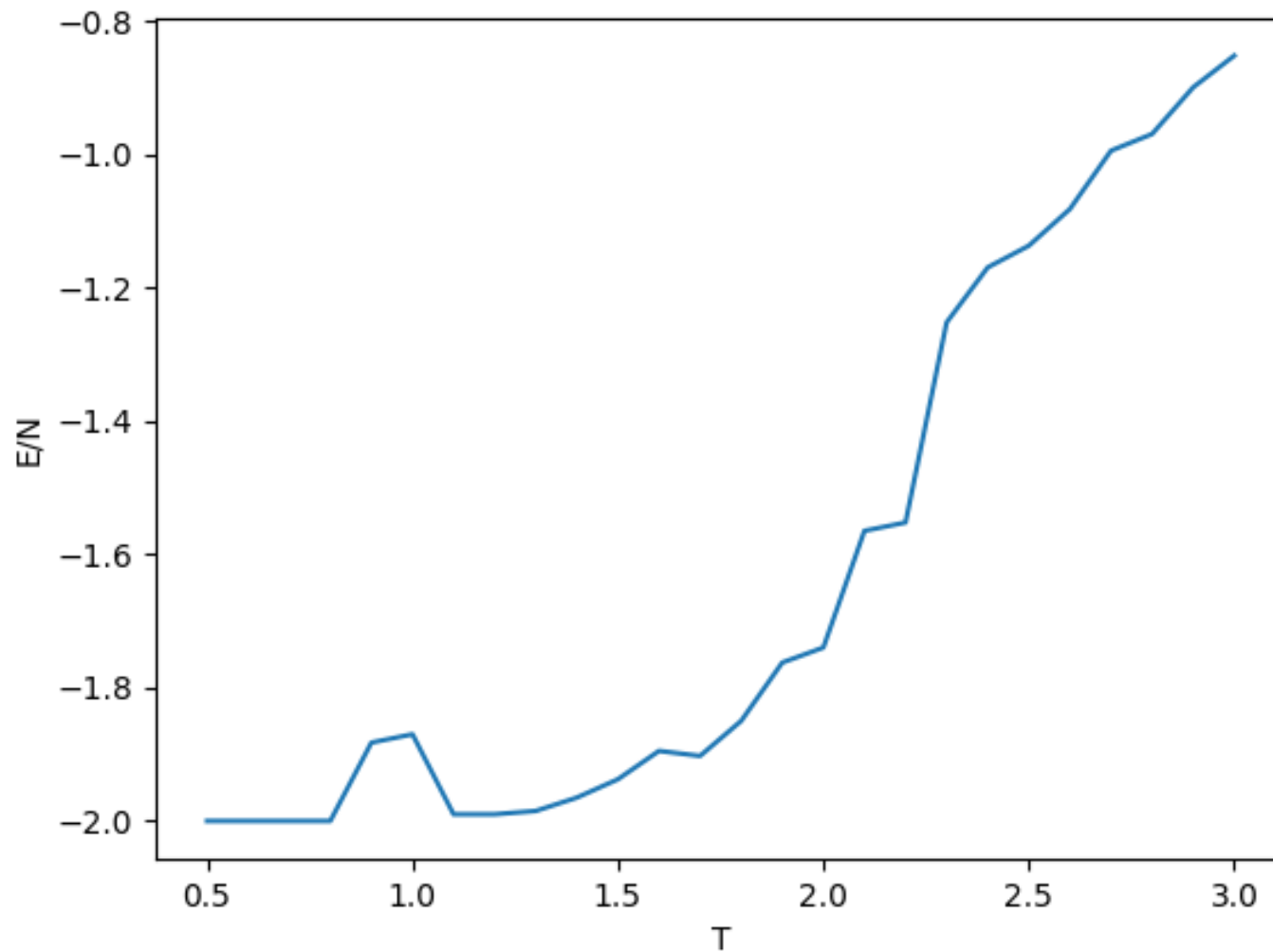
$T = 3.00$; $E = -1364.0$, $M = 292.0$



sometimes we get states like this where there are two large domains of opposite spin. Probably more iterations would have gotten rid of this.

Ising Model

- As temperature is increased, magnetization goes away



Simulated Annealing

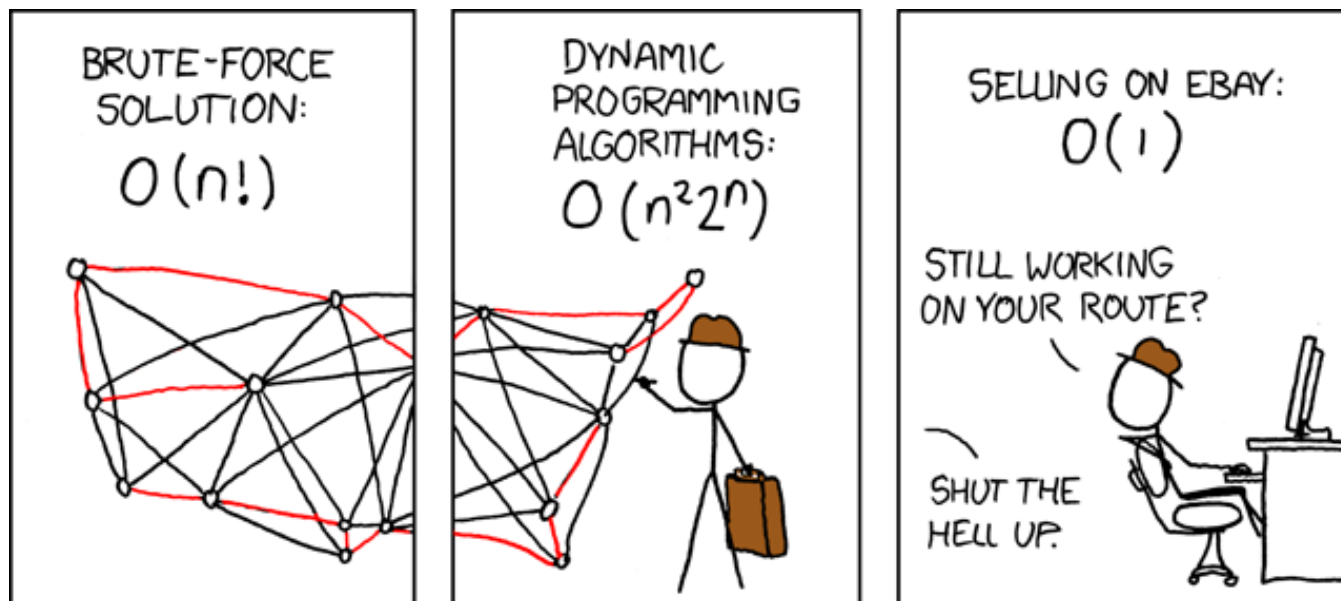
- We can apply the Markov chain Monte Carlo methods to optimization problems, e.g. find a (global) minimum or maximum
- We mimic nature
 - Quickly cool a metal or glass and it finds a local minimum of energy, but not necessarily the ground state
 - if you slowly cool a metal, it can find its lowest energy state (be strongest)—this is *annealing*
- Do the MCMC process, but slowly bring the temperature of the system down to a point where it freezes / crystalizes
 - At this cold T , no further transitions are likely

Traveling Salesman

- Consider the traveling salesman problem
 - Imagine a list of cities (you have their x, y locations on a map)
 - You want to travel to all of the cities in the shortest amount of time (total distance of your route)
 - Some variations: allow starting and ending city to float? End at the same place you start?
 - You could also assign costs to certain segments, making them most expensive to complete (e.g. no road, travel by boat, ...)

Traveling Salesman

- The traveling salesman problem is known to be a very difficult problem
- Direct solution:
 - Evaluate every possible route and keep the shortest—this is computationally expensive
 - $(N-1)!$ Possible orderings of the cities from a chosen starting point



(xkcd)

Traveling Salesman

- We'll make our “energy” be the distance:

$$d = \sum_{i=1}^N \left[(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 \right]^{1/2}$$

- We use $N+1 = 1$ here to do a closed loop
- Our move set is swapping a pair of cities at random
- We start the system off at some high temperature—this will encourage swapping
- We accept a swap if it matches the Metropolis acceptance criteria
- We anneal by slowly lowering the temperature according to a cooling schedule:

$$T(t) = T_0 e^{-t/\tau}$$

Traveling Salesman

- We choose the starting temperature to be high

- Our probability of accepting a move is

$$P = e^{-\Delta d/T}$$

- Imagine a circuit with a distance of 4, where the typical Δd involved in swapping 2 cities is ~ 0.1

Δd	T	P
0.1	10	0.99
0.1	1	0.9
0.1	0.1	0.37
0.1	0.01	4.5e-5

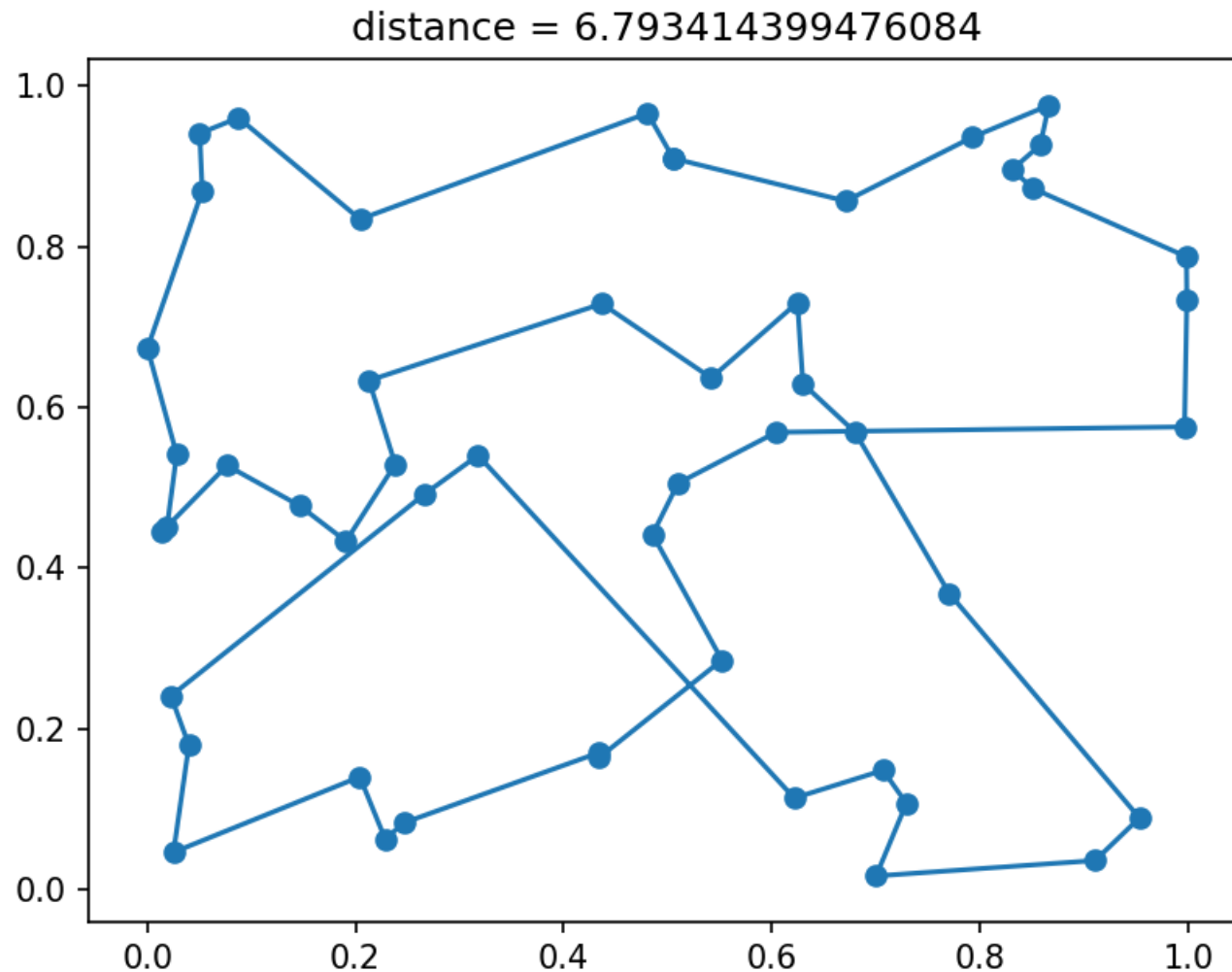
- So you should pick your initial T to be $\sim 10x$ or more the typical Δd
- The cooling timescale is $1/\tau$ —we want this to be relatively long
- Note: simulated annealing will not necessarily get *the* optimal solution, but it will find a good solution

Traveling Salesman

- You can automate the selection of the initial temperature
 - Generate an initial state, try N different moves, if at least 80% are not accepted, then double the temperature and try again (Strickler & Schachinger)
- Some simulated annealing algorithms implement restarts—go back to a previously better solution with some random probability

Traveling Salesman

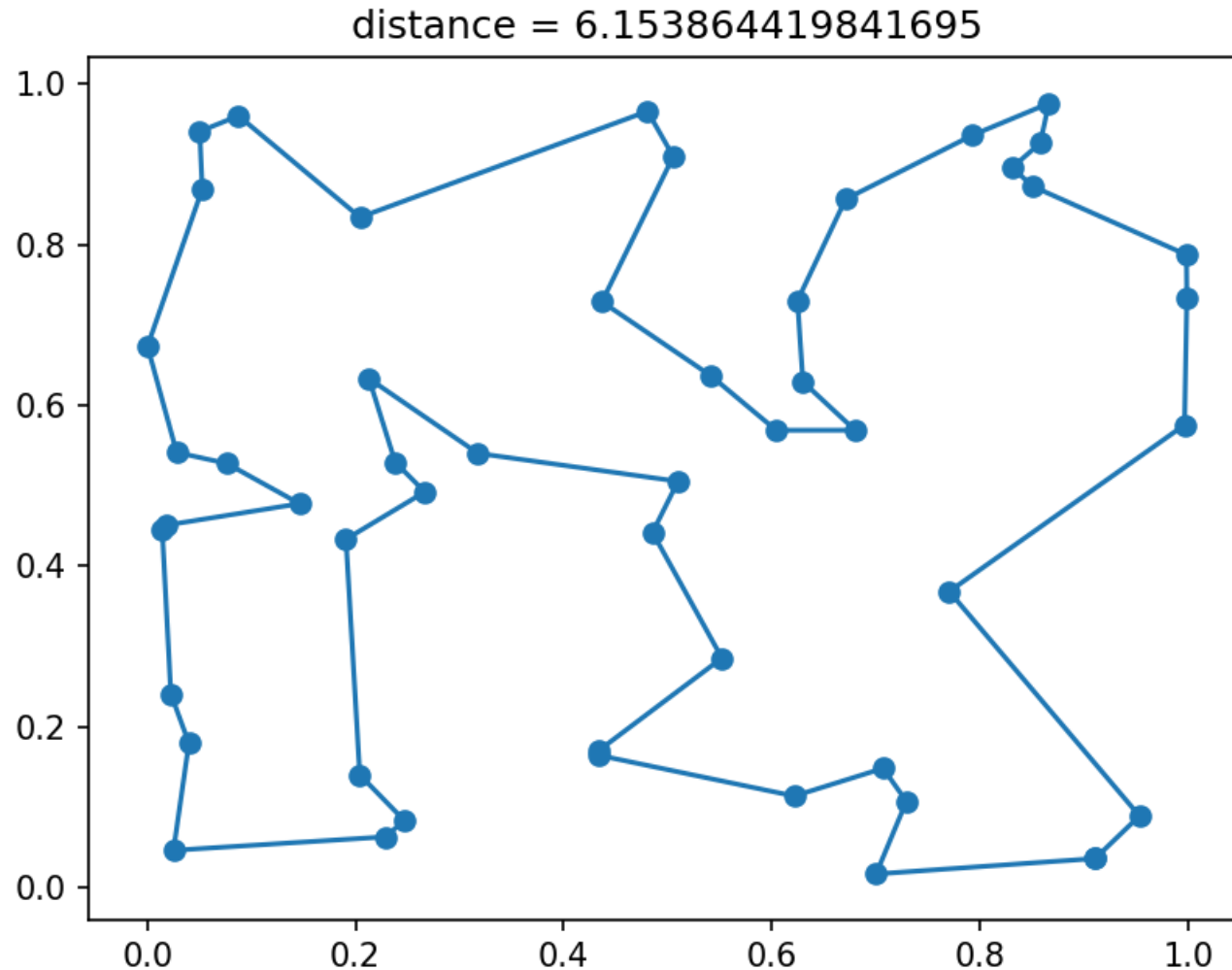
- We'll randomly choose the city locations (but with a known seed, so we can run the same configuration multiple times)



code: traveling_salesman.py

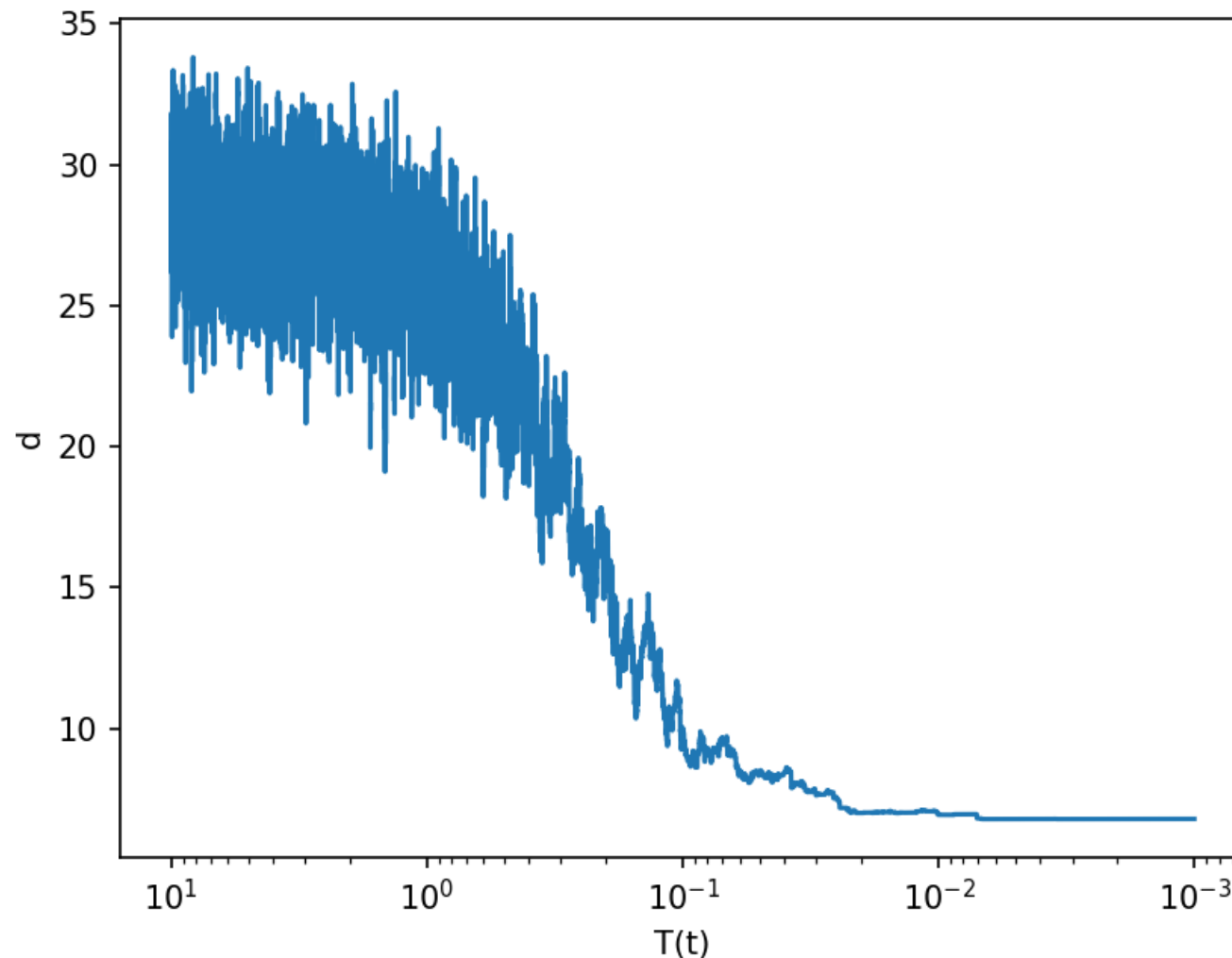
Traveling Salesman

- I ran this a lot of times, with $T_0 = 10$ and $T_0 = 100$, here's the shortest distance (seemed to get the best result, but also greatest variation, with the lower initial T)



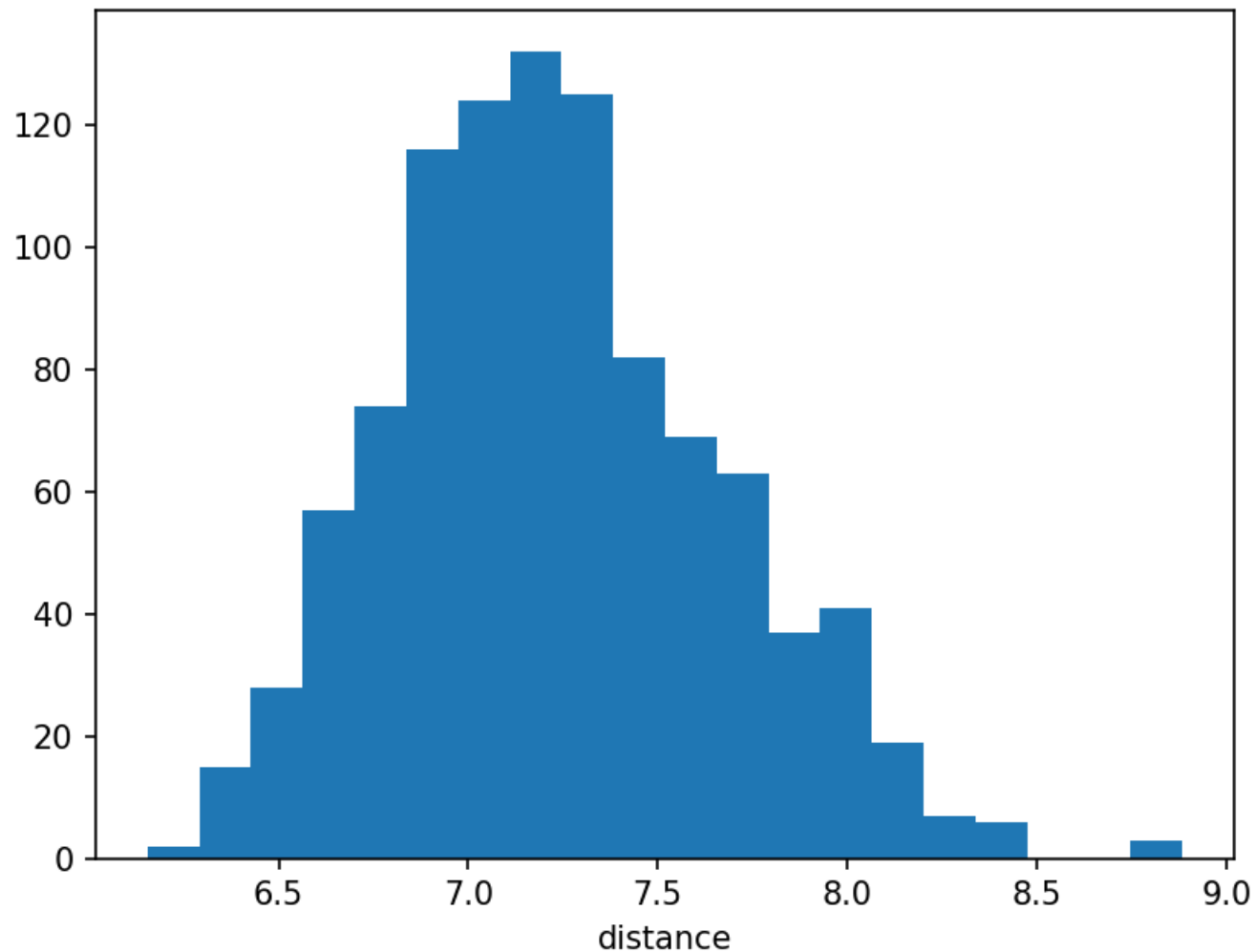
Traveling Salesman

- Monitoring the history can tell us how well our cooling function did



Traveling Salesman

- Distribution of shortest paths found from 1000 runs of the simulated annealing problem



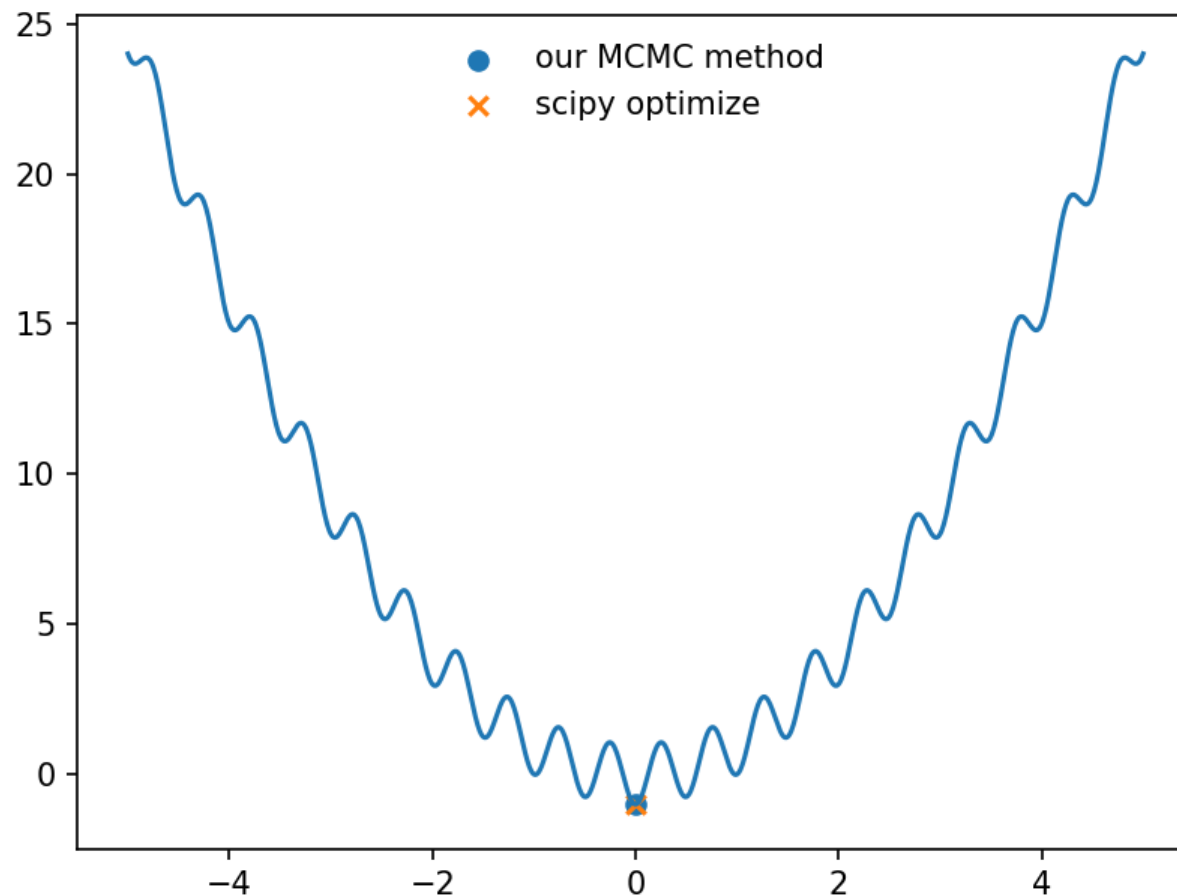
code: traveling_salesman.py

Function Minimization

- Simulated annealing can also be used to find the minimum of a function
 - Again, it may not find the lowest minimum, since there is randomness involved
- Our energy is now the function value at the current guess for the minimum
 - If we are interested in the maximum, then we use $-f$ as the energy
- Our move set is all the possible points over which we expect to find the minimum
 - Perturb the initial guess, $x_0 \rightarrow x_0 + \delta$
 - Pick δ from Gaussian normal numbers with some appropriate width (suggestion from Newman)

Function Minimization

- Example: $f(x) = x^2 - \cos(4\pi x)$
 - Find $x = 0.000683608212246$ as the minimum

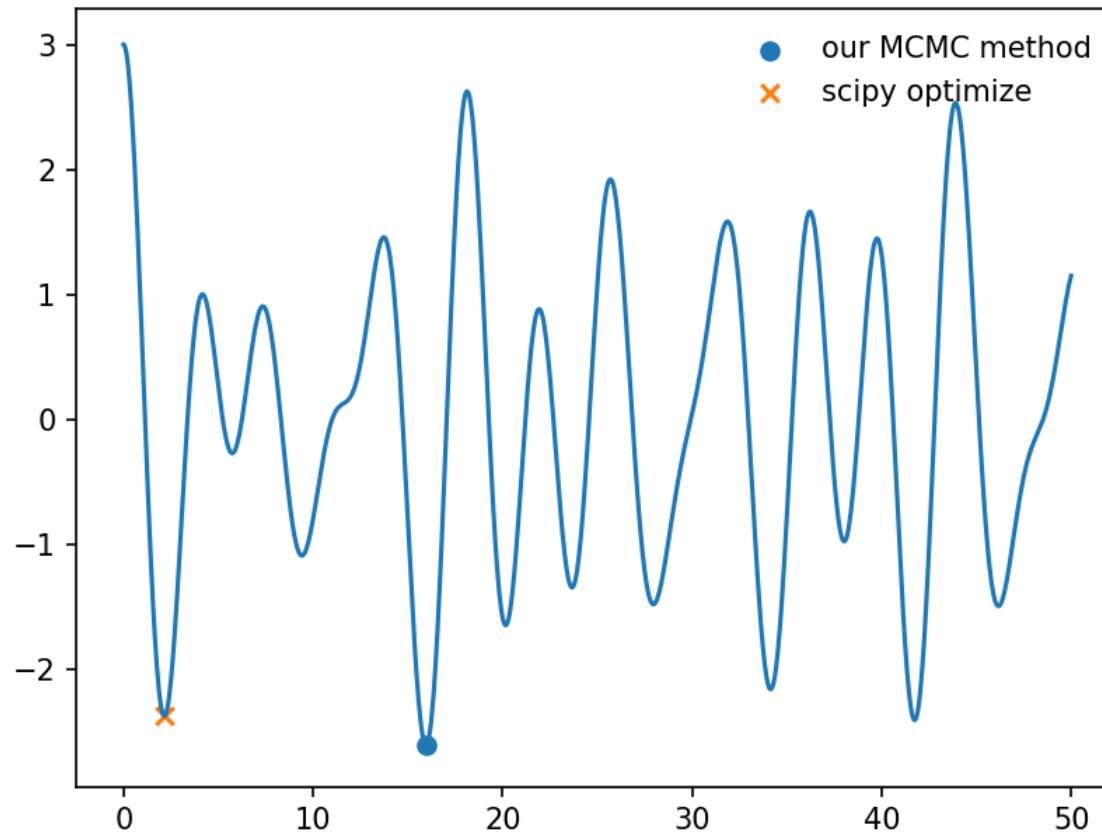


Function Minimization

- In comparison, Newton's method has a very hard time with this function (zeroing the first derivative)

Function Minimization

- Example: $f(x) = \cos(x) + \cos(\sqrt{2}x) + \cos(\sqrt{3}x)$
 - Restrict to $[0, 50]$
 - Find $x = 15.947857$ (but not always!)



Function Minimization

- The `scipy.optimize.minimize_scalar` method, which uses Brent's method, fails to find the true minimum
 - It gives -2.3775...