Quantitative Applications for Data Analysis: Monte Carlo methods

Erik Spence

SciNet HPC Consortium

24 March 2026

SciNet

Today's slides

Today's slides can be found here. Go to the "Quantitative Applications for Data Analysis" page, under Lectures, "Monte Carlo methods".

`https://scinet.courses/1399`

SciNet

Today's class

Today we will visit the following topics:

- Monte Carlo methods, in general,
- Monte Carlo integration,
- Markov Chain Monte Carlo.

SciNet

Why randomness?

Why are we interested in randomness?

- To simulate some physical phenomenon that has noise. E.g. Brownian motion, Nyquist noise. On the level of their description, this is real randomness.
- To perform averages or integrals in systems with many degrees of freedom. E.g. Stat. Phys. computations, path integral calculations. Here, the main objective is to get the converged answer quickly.
- To test a statistical model.

We've seen this already in some of our assignments.

SciNet

Creating randomness

True random number generators include

- Lava lamps,
- Radioactive decay,
- Various quantum processes,
- Atmospheric noise,
- Random computer hardware noise signals (thermal noise).

These are generally slow, expensive, impossible to reproduce for debugging. Hard to characterize underlying distribution.

In contrast, examples of Pseudo-Random Number Generators (PRNGs) include

- Come up with a algorithm that produces random numbers
- But wouldn't such an algorithm would be deterministic?
- Only has to **act** random, i.e., give fair and uncorrelated sequence.

SciNet

Pseudo-Random Number Generators

Recipe:

- Define some ‘state’, initialized by some ‘seed’ value(s).
- Produce a number from this state.
- Advance the state determistically.
- As long as the numbers produces behave as if they are
  - independent
  - identically distributed
  - according to a predefined distribution (eg uniform)

```
In [1]: import numpy as np
In [1]: import numpy.random as npr
In [1]:
In [1]: npr.seed(111)
In [2]: npr.choice(np.arange(10), 2, replace = True)
Out[2]: array([7, 0])
In [3]:
In [3]: npr.seed(111)
In [4]: npr.choice(np.arange(10), 2, replace = True)
Out[4]: array([7, 0])
In [5]:
In [5]: npr.seed(111)
In [6]: npr.choice(np.arange(10), 2, replace = True)
Out[6]: array([7, 0])
In [7]:
```

SciNet

Monte Carlo analysis

Monte Carlo analyses are a collection of techniques whose unifying feature is the use of random sampling to generate results. These analyses generally fall into one of three categories:

- Adding randomness to otherwise-deterministic dynamics, and studying how the dynamics are changed, or the resulting data distributions.
- Generating samples from a given probability distribution, $\boldsymbol{P}(\boldsymbol{x})$, usually a distribution that is complicated and can't be dealt with nicely in closed form.
- Estimating expectation values under this distribution, e.g.

$$\langle \boldsymbol{A}(\mathbf{x}) \rangle = \int \boldsymbol{P}(\mathbf{x}) \boldsymbol{A}(\mathbf{x}) d\mathbf{x}$$

  where $\mathbf{x}$ is typically high dimensional.

These depend on having a good random number generator!

Monte Carlo example: integration

The basic idea of Monte Carlo integration is very simple and only requires elementary statistics. Suppose we want to find the value of

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

.

The quantity S is usually estimated using the expression

$$S \simeq \frac{(b-a)}{n} \sum_{i=1}^{n} f(a + U_i(b-a))$$

where U is the uniform distribution sampled n times between b and a. As you can see, this estimation is simply calculating the average value of f in the interval and then multiplying by $(b-a)$ to get the value of the area.

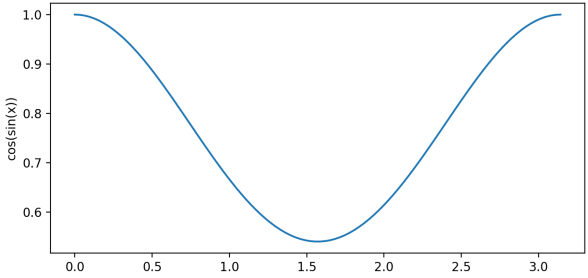
Integration, example

Let's integrate the function

$$f(x) = \cos(\sin(x))$$

from 0 to π.

```
In [7]:
In [7]: import numpy as np
In [8]: import matplotlib.pyplot as plt
In [9]:
In [9]: x = np.linspace(0, pi, 100)
In [10]: plt.plot(x, np.cos(np.sin(x)))
In [11]:
```



Integration, example, continued

Let's integrate the function
$f(x) = \cos(\sin(x))$ from 0 to π.

```
# int_MC.py
import numpy.random as npr
import numpy as np

def f(x):
  return np.cos(np.sin(x))

def int_MC(num, a, b):
  results = 0
  for i in range(num):
    results += f(a + npr.uniform(0, b - a))

  return results * (b - a) / num
```

```
In [11]:
In [11]: import int_MC
In [12]:
In [12]: int_MC.int_MC(10000, 0, pi)
Out[12]: 2.4101580829908666
In [13]:
In [13]: import scipy.integrate as si
In [14]:
In [14]: si.simpson(np.cos(np.sin(x)), x)
Out[14]: 2.403936768802837
In [15]:
```

Multidimensional integration

Using a Monte Carlo integration technique on a 1D problem is inefficient. There are much more efficient techniques out there, such as Simpson's rule.

But suppose our integral is of a higher dimension, say, 4D. This is where Monte Carlo integration techniques start to become more useful. They can efficiently reach into as many dimensions as necessary.

$$S = \int_V f(\mathbf{x}) d\mathbf{x}$$

Where now we are integrating over the 3D domain V, and $\mathbf{x}$ is a 3D vector.

SciNet

Multidimensional integration, continued

Let us integrate over 3 dimensions, rather than 1.

$$S = \int_V f(\mathbf{x}) d\mathbf{x}$$

The quantity S is estimated using a similar expression to the 1D example.

$$S \simeq \frac{V}{n} \sum_{i=1}^{n} f(a_x + U_i(b_x - a_x), a_y + U_i(b_y - a_y), a_z + U_i(b_z - a_z))$$

where a_x, b_x are the limits of integration for x, and the samples from the uniform distribution must be evenly sampled throughout V. Obviously, if a data point is randomly sampled outside of V it cannot be used.

Multidimensional integration, example

Let's integrate over the 4-sphere, to calculate its volume.

Rather than limit the range of npr.uniform to V, we keep sampling points until all points are within V. This makes sure the points are spaced evenly.

Note that the volume of a 4-sphere is $\pi^2 r^4/2$. We get half this value, since we're only integrating half of the 4-sphere.

```
In [15]: import MultiD_int_MC as MD
In [16]:
In [16]: MD.multiD_int_MC(1, 10000)
Out[16]: 2.4791925774411387
```

```
# MultiD_int_MC.py
import numpy.random as npr, numpy as np

def f(r, x, y, z):
  return np.sqrt(r**2 - x**2 - y**2 - z**2)

def my_samp(r):
  x = npr.uniform(-r, r); y = npr.uniform(-r, r)
  z = npr.uniform(-r, r); return x, y, z

def multiD_int_MC(r, num):
  results = 0
  for i in range(num):
    x, y, z = my_samp(r)
    while (x**2 + y**2 + z**2 > r**2):
      x, y, z = my_samp(r)
    results += f(r, x, y, z)
  return results / num * (4 / 3 * np.pi * r**3)
```

Monte Carlo integration, summary

A few notes about Monte Carlo integration.

- MC integration works under minimal assumptions (the desired mean must exist, then (law of large numbers) $\mathcal{P}(\lim_{n\to\infty} \hat{\mu} = \mu) = 1$.
- MC integration does not deliver extreme accuracy

$$\mathbf{RMSE} = E((\hat{\mu} - \mu)^2) = \sigma/\sqrt{n}$$

- MC integration is very competitive in high dimensional or non-smooth problems.
- MC integration has good error estimation.
- There are ways to improve the approach we've used, such as using using non-uniform sampling (Importance sampling).

SciNet

Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) is considered one of the most important algorithms of the 20th century.

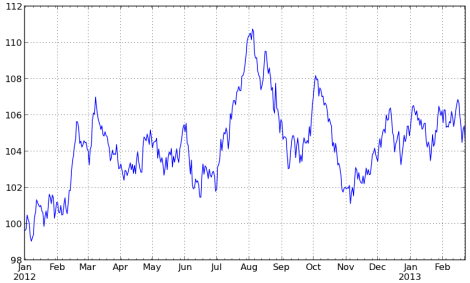
- It combines two techniques:
  - Monte Carlo: estimating a distribution's properties by randomly sampling from it, and
  - Markov Chains: the random samples are generated by a restricted sequential process.
- The resulting chain of samples is essentially a 'random walk' through a high-dimensional space.
- This walk is used for optimization problems and model fitting, in particular Bayesian inference.
- Once you have your chain, you can use this data to determine the distributions of whatever parameters you're after.

SciNet

Markov chain

A Markov chain is a chain of 'steps' through parameter space, with particular characteristics.

- Usually when we deal with random samples they are independent and identically distributed. New samples don't depend on previous samples.
- A *Markov chain* is a sequence of random numbers $\boldsymbol{\theta_0, \theta_1, ..., \theta_n}$ where the probability of $\boldsymbol{\theta_{i+1}}$ depends on $\boldsymbol{\theta_i}$, but does NOT depend on $\boldsymbol{\theta_{i-1}}$.
- Distribution is $\boldsymbol{P(\theta_{i+1}|\theta_i)}$ not $\boldsymbol{P(\theta_i)}$.
- A classic example of a Markov chain is a random walk: $\boldsymbol{\theta_{i+1} = \theta_i + \epsilon}$



Bayesian Inference

Bayesian inference is a process for updating our beliefs when we acquire new information, using Bayes theorem:

$$P(\theta|\mathbf{x}) = \frac{P(\mathbf{x}|\theta)P(\theta)}{P(\mathbf{x})}$$

Terminology:

- $\mathbf{x}$ is the data, θ are the model parameters.
- $P(\theta|\mathbf{x})$ is called the *posterior*, the probability distribution of θ after knowing $\mathbf{x}$.
- $P(\theta)$ is called the *prior*, $\pi(\theta)$, the *a priori* probability distribution of θ (our beliefs about θ prior to knowing $\mathbf{x}$).
- $P(\mathbf{x}|\theta)$ is called the *likelihood*, $\mathcal{L}(\theta, \mathbf{x})$, the probability distribution of $\mathbf{x}$ given θ.
- $P(\mathbf{x})$ is called the *model evidence*: $P(\mathbf{x}) = \int P(\mathbf{x}|\theta)P(\theta)d\theta$.

SciNet

MCMC

MCMC is most often used to determine the posterior distribution, $\boldsymbol{P(\theta|\mathbf{x})}$, for some problem. So what do we need to do this?

- Data, $\mathbf{x}$. Presumably you already have this.
- A model with which you can represent your data. The model parameters, $\boldsymbol{\theta}$, are contained herein.
- A sampling algorithm, with which you generate new values for $\boldsymbol{\theta}$, your model parameters.
- An equation for your likelihood, $\boldsymbol{\mathcal{L}(\theta, \mathbf{x})}$.
- An equation for the prior, $\boldsymbol{\pi(\theta)}$. This is needed, as you need to describe what you think the parameters look like before you have any data.

Once you have these pieces you can begin to perform MCMC.

SciNet

Bayesian Inference, likelihood

So how do we calculate the likelihood, $P(\mathrm{x}|\theta)$ ($\mathcal{L}(\theta, \mathrm{x})$),? We make the usual assumption, that our data contains noise, $y_{\mathbf{obs}} = y_{\mathbf{true}} + \epsilon$, and then assume that the noise is Gaussian, and the data are uncorrelated. As a result, we have

$$P(\mathrm{x}|\theta) = \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} e^{-\frac{1}{2}\left(\frac{y_i - f(x_i)}{\sigma_i}\right)^2}$$

Recall that $y_i - f(x_i)$ are just the residuals of the model, f, and σ_i is the uncertainty for data point i. This is sometimes modelled as the log of the likelihood.

$$\log P(\mathrm{x}|\theta) = \sum_i -\log(2\pi) - \log(\sigma_i^2) - \left(\frac{y_i - f(x_i)}{\sigma_i}\right)^2$$

where we've multiplied the right side by 2. Logs are easier to deal with and are more numerically stable.

Bayesian Inference, prior

So how do we calculate the prior, $\pi(\theta)$? This is subjective, based on the prior knowledge of the researcher. There are several commonly used options:

- uniform prior: prior is a constant over some range, no particular value of θ is any better than any other.
- log-uniform prior: useful if your parameter many orders of magnitude (doesn't work near zero).
- posterior prior: use a previously-calculated posterior as your prior, if the posterior is related to the problem you're working on.
- observation-based prior: using observations to create a distribution to use as your prior.

It's not uncommon to use the log of the prior, since it's more numerically stable.

SciNet

MCMC, sampler

So what does the sampling algorithm do?

- It generates a new value for $\boldsymbol{\theta_{i+1}}$, given $\boldsymbol{\theta_i}$.
- This is done by sampling from the "proposal distribution", $\boldsymbol{q(\theta_{i+1}|\theta_i)}$, (typically Gaussian, for continuous model parameters). This value is added to $\boldsymbol{\theta_i}$ to get $\boldsymbol{\theta_{i+1}}$. The proposal distribution should be symmetric and centred at zero.
- This value of $\boldsymbol{\theta_{i+1}}$ is then passed to the model, which, using the data, calculates the likelihood, $\boldsymbol{\mathcal{L}(\theta, \mathrm{x})}$.
- The value is also passed to the prior, $\boldsymbol{\pi(\theta)}$, to calculate its value.
- The likelihood and prior are then combined to calculate the posterior, $\boldsymbol{P(\theta|\mathrm{x})}$.

$$\boldsymbol{P(\theta|\mathrm{x}) \propto \mathcal{L}(\theta, \mathrm{x})\pi(\theta)}$$

We ignore the denominator $\boldsymbol{P(\mathrm{x})}$, as this is just a normalization constant.

SciNet

MCMC, continued

Ok, now that we've got the pieces, how do we actually do MCMC?

We will use a particular type of MCMC called the Metropolis algorithm.

1. Choose a starting position for the model parameters, $\boldsymbol{\theta_0}$.
2. Propose the next data point using the sampler.
3. Compare the posterior's new value, $\boldsymbol{P(\theta_{i+1}|\mathbf{x})}$, to the previous value, $\boldsymbol{P(\theta_i|\mathbf{x})}$. If $\boldsymbol{P(\theta_{i+1}|\mathbf{x}) > P(\theta_i|\mathbf{x})}$ then take $\boldsymbol{\theta_{i+1}}$ as the next value in our Markov chain.
4. If $\boldsymbol{P(\theta_{i+1}|\mathbf{x}) < P(\theta_i|\mathbf{x})}$ then randomly take $\boldsymbol{\theta_{i+1}}$ to be the current value in the chain with probability $\boldsymbol{P(\theta_{i+1}|\mathbf{x})/P(\theta_i|\mathbf{x})}$.
5. Repeat, starting at step 2.

The resulting chain, $\boldsymbol{\theta_0, \theta_1, ..., \theta_n}$, is our Markov Chain. It is the data used to calculate $\boldsymbol{P(\theta|\mathbf{x})}$.

SciNet

Bayesian inference, example

Suppose we've got some data and, having plotted it, have decided that the data follows a linear relationship. Let's use MCMC to generate the distributions of our model parameters.

What do we need to do MCMC?

- A model: $\boldsymbol{y = mx + b + \epsilon}$. Let us assume that the noise is Gaussian, with a mean of zero and a standard deviation of $\boldsymbol{\sigma}$.
- This means there are 3 model parameters: $\boldsymbol{\theta = (m, b, \sigma)}$.
- A likelihood, $\boldsymbol{\mathcal{L}(\theta, \mathbf{x})}$: we will use a Gaussian of the residuals.
- A prior, $\boldsymbol{\pi(\theta)}$: we will use uniform priors for this calculation.

We will also take the log of everything, as this makes things more numerically stable, and simplifies the coding. We will code this ourselves, as it's not too difficult.

SciNet

Bayesian inference, example, continued

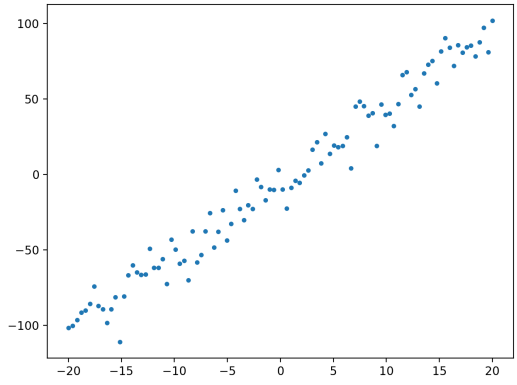
```
# my_mcmc.py
import scipy.stats as ss, numpy as np

def likelihood(params, x, y):
  m = params[0];  b = params[1];  sd = params[2]
  pred = m * x + b
  # We want the log of the Gaussian.  We take
  # the sum because we are taking the log.
  return np.sum(ss.norm.logpdf(y, loc = pred,
                scale = sd))

def prior(params):
  m = params[0];  b = params[1];  sd = params[2]
  # We take the prior to be log(P(m)P(b)P(sd)).
  # Assume uniform probabilities from -50 to 50.
  return ss.uniform.logpdf(m, -50, 100) +
    ss.uniform.logpdf(b, -50, 100) +
    ss.uniform.logpdf(sd, -50, 100)
```

```
# my_mcmc.py, continued
def posterior(params, x, y):
  return likelihood(params, x, y) +
    prior(params)

def proposal_func(params):
  # We propose 3 new values based on the
  # existing values, using hard-coded
  # standard deviations.
  return ss.norm.rvs(size = 3, loc = params,
    scale = 0.1)

def calc_chisq(params, x, y):
  m = params[0];   b = params[1]
  sd = params[2]
  return np.sum(((m * x + b - y) / sd)**2)
```

SciNet

Bayesian inference, example, continued more

```
In [17]:
In [17]: import matplotlib.pyplot as plt
In [18]:
In [18]: trueM, trueB, trueSd = 5, -5, 10
In [19]:
In [19]: n = 100
In [20]: x = np.linspace(-20, 20, n)
In [21]:
In [21]: y = trueM * x + trueB
In [22]: y += ss.norm.rvs(size = n, loc = 0,
                  scale = trueSd)
In [23]:
In [23]: plt.plot(x, y, '.')
In [24]:
```



Bayesian inference, example, continued some more

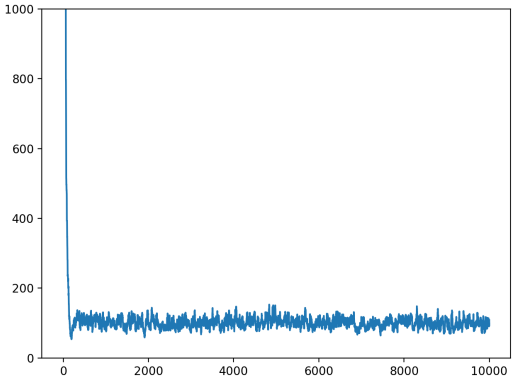
```
In [24]:
In [24]: import my_mcmc
In [25]:
In [25]: startvalue = np.array([1, 0, 1])
In [26]:
In [26]: num = 10000
In [27]:
In [27]: chain = my_mcmc.run_mcmc(startvalue,
                                  x, y, num)
In [28]:
In [28]: chain = np.array(chain)
In [29]:
```

```
# my_mcmc.py, continued
def run_mcmc(startvalue, x, y, n):
  chain = [startvalue]

  for i in range(n):
    proposal = proposal_func(chain[i])
    old_p = posterior(chain[i], x, y)
    new_p = posterior(proposal, x, y)

    if new_p > old_p:
      chain.append(proposal)
    else:
      p = np.exp(new_p - old_p)
      if ss.uniform.rvs() < p:
        chain.append(proposal)
      else: chain.append(chain[i])
  return chain
```

SciNet

Bayesian inference, example, burn in

```
In [29]: chisq = np.zeros(num + 1)
In [30]:
In [30]: for i in range(num + 1):
...:     chisq[i] = my_mcmc.calc_chisq(chain[i,],
...:                         x, y)
In [31]:
In [31]: plt.plot(chisq)
In [32]: plt.ylim(0,1000)
In [33]:
In [33]: trueM, np.mean(chain[2000:,0])
Out[33]: (5, 4.973537491078847)
In [34]: trueB, np.mean(chain[2000:,1])
Out[34]: (-5, -4.719488741813247)
In [35]: trueSd, np.mean(chain[2000:,2])
Out[35]: (10, 10.325602452263714)
```



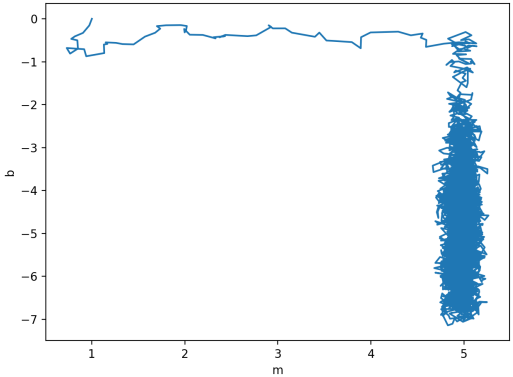
"Burn in" is the period before the MCMC lands near the correct values, as can be seen in the chi-squared values for the chain.

Bayesian inference, example, burn in, continued

We can plot the walk as it tries to find the correct values of the parameters.

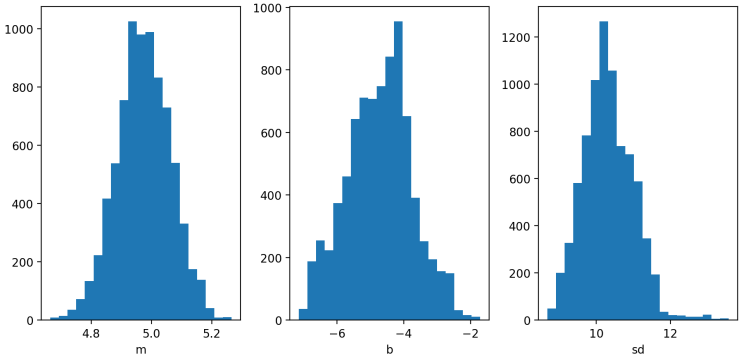
We can see that the chain has a much better sense of the value of m than b.

```
In [36]:
In [36]: plt.plot(chain[:,0], chain[:,1])
In [37]:
```



Bayesian inference, example, burn in, continued

```
In [37]: ans = chain[2000:,]
In [38]:
In [38]: plt.subplot(131)
In [39]: h = plt.hist(ans[:,0])
In [40]: plt.xlabel('m')
In [40]:
In [40]: plt.subplot(131)
In [41]: h = plt.hist(ans[:,1])
In [42]: plt.xlabel('b')
In [42]:
In [42]: plt.subplot(131)
In [43]: h = plt.hist(ans[:,2])
In [44]: plt.xlabel('sd')
In [44]:
```



Summary

Some notes from today's class.

- MCMC is a powerful technique, but it's not foolproof.
- How to know if the chain has adequately sampled the distribution (aka **converged**)?
  - Run multiple chains with different starting points, and compare the inter-chain and intra-chain variances (*Gelman-Rubin test*).
- an MCMC can be used for approximating a multi-dimensional integral by using an ensemble of "walkers" moving around randomly. At each point where a walker steps, the integrand value at that point is counted towards the integral.
- the random samples of the integrand used in a conventional MC integration are statistically independent, those used in MCMC methods are correlated.
- A Markov chain is constructed in such a way as to have the integrand as its equilibrium distribution.

SciNet