Imperial College London

Computational Physics: Random Numbers and Monte Carlo Methods

Blackett Laboratory

17 October 2018

Previously: Summary

- Introduction
- Review of Fourier Transforms
- Discrete Fourier Transforms
- Sampling and Aliasing
- Fast Fourier Transforms
- An Introduction to Pseudocode

Outline

- Random Numbers–in the real world and in a computer
- Pseudo-Random Numbers
- Non-Uniform Distributions
  - Transformation Method
  - Rejection Method
- Monte Carlo Minimisation
- Monte Carlo Integration

Random Numbers

In the Real World

- A coin toss
  - effectively generating an integer out of $[0, 1]$
- A die roll
  - " " $[1, 6]$
- Roulette
  - selecting an element from $\{0, 00, [1, 36]\}$
- A pack of cards
  - selecting from 52 cards from four suits and a joker or two
- Lottery Draws
  - an integer from $[1, 59]$, six times: $59!/53!/6!$ outcomes?
- FA Cup/League Cup Draw
  - balls drawn from a bag

All of the above are (should be) designed to be unbiased

- how would we test this?
- which two broad groupings do the above examples fall into?

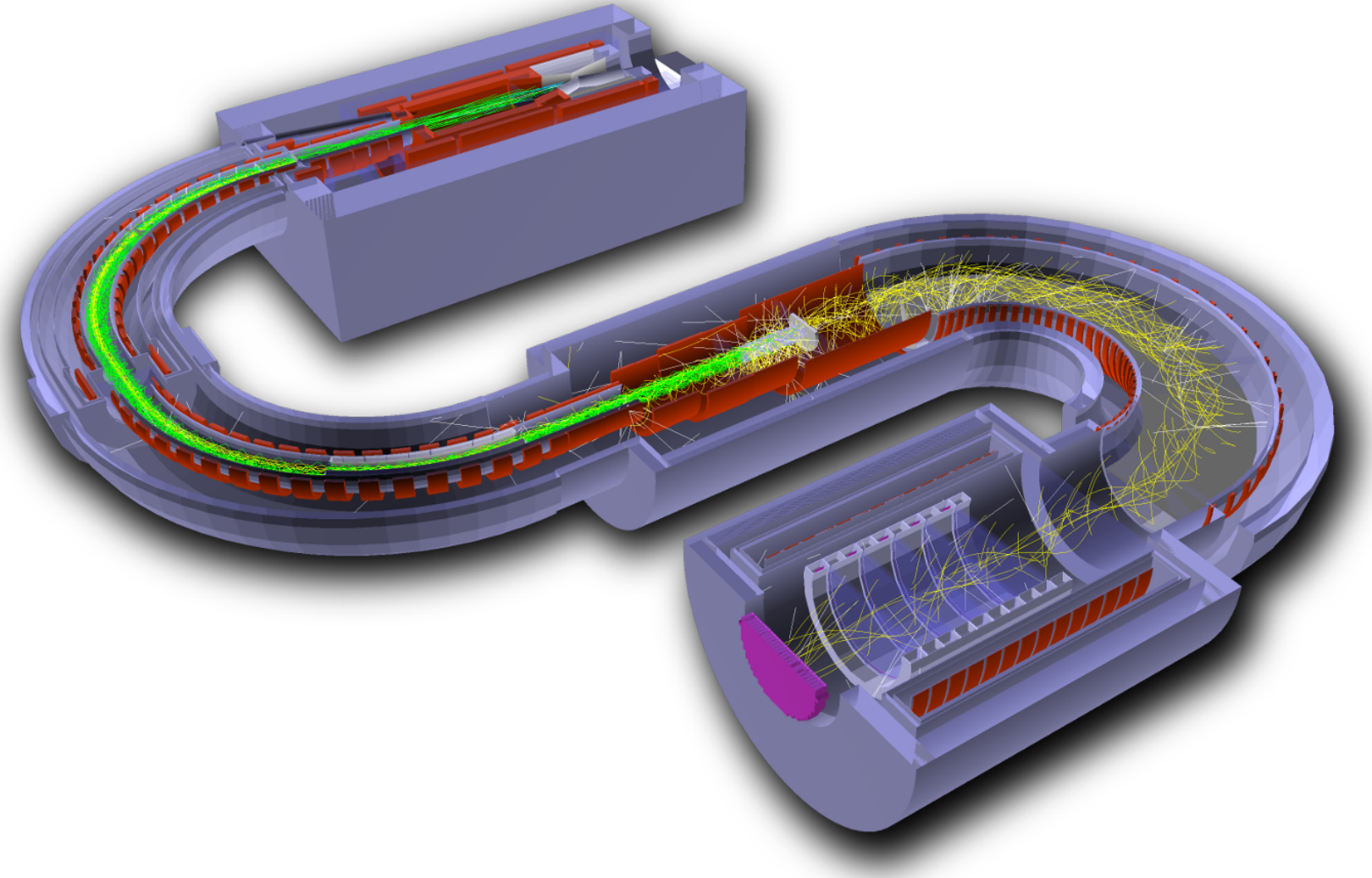
Random Numbers

uses in Computational Physics

- Modelling of inherently random processes
  - predictions are of probability densities
- Deterministic systems (for enormously complicated systems, exact solutions can be infeasible)
- A combination of the above...
  - a particle passing through matter etc
- but these are relatively trivial (while being very important) uses...

Random Numbers

In Particle Physics



The COMET Experiment: PhD Thesis, B. Krikler

Random Numbers

uses in Computational Physics (continued)

- ...
- Mathematical calculations (mathematically, there is nothing random about these, but the use of random numbers can be highly advantageous)

The use of random numbers to help perform such mathematical calculations is extremely valuable

Random Numbers

In a Computer

True Random Numbers

- based on physical processes
  - imagine sampling the nth digit of a digital display of a photodiode voltage in ambient surroundings
  - thermal noise, radio waves, radioactive sources
  - real-world examples:
    `http://moonbaseotago.com/onerng/theory.html`,
    `http://www.entropykey.co.uk/tech`
- not touched upon further in this course, but you should be aware of these

Random Numbers

in a computer (continued)

Pseudo Random Numbers

- a sequence of *seemingly* random numbers, generated according to an algorithm (a *random number generator*)
- usually with (a) *seed* number(s) to initiate the algorithm
- simple example: the **Linear Congruential** generator
- many refinements: the "Mersenne Twister" is the most widely used; RANLUX is also known to produce very high quality uncorrelated numbers
  - Mersenne Twister: Matsumoto, Nishimura, January 1998 `doi:10.1145/272991.272995`; python `random()` uses this; MT is still not "perfect"–this is a much discussed topic
  - RANLUX: Martin Lüscher, February 1994 `doi:10.1016/0010-4655(94)90232-1`

A pseudo random number generator is able to **reproduce the same sequence** from the same seed–an important property not possible (of course) with a true random number generator.

Pseudo Random Number Sequence Generation

the Linear Congruential algorithm

With x being a random variable, and the *deviate* being the values that it can take, we can define:

Uniform Deviate

a random number lying within a range where any number has the same probability as any other

$0 \leq x < 1$ is commonly used as the range.
The **Linear Congruential Generator** algorithm is given by:

$$I_{n+1} =$$

where I_n are the random numbers, a is a multiplier, c is the increment, and m is the modulus.
($a\%b$ denotes modulo division; i.e. the remainder of dividing a by b.)

Pseudo Random Number Generation I

Code snippets

Linear Congruential Generator

$$I_{n+1} =$$

```
# define the LC-PRNG algorithm
def nplusone(a,c,m,n):
    return (a*n+c)%m

# test it
print nplusone(5,3,10,5)

# recommended parameters from "Numerical Recipes"
a = 1664525; m = 2**32; c = 1013904223
n = 0
for i in range(10):
n = nplusone(a,c,m,n)
print(n)
```

Pseudo Random Number Generation II

Code snippets

Linear Congruential Generator

$$I_{n+1} =$$

```
# one choice of settings (not recommended)
a = 1664525; c = 1013904223; m = 10001

# other settings (not recommended)
a = 151; c = 99; m = 100

# generate a uniform deviate in [0,1)
totN = 100000;
x = np.zeros(totN, dtype=float)
for i in range(totN):
n = nplusone(a,c,m,n)
x[i] = n/float(m) # normalise to [0,1)

(hist, bins, patches) = plt.hist(x, bins=100,range = (0.0,1.0))
```

Pseudo Random Number Generation III

Linear Congruential Generator

$$I_{n+1} =$$

- What are some problems with the algorithm?

But for a very long time the LCG was the standard way of generating random numbers, and it illustrates well the pitfalls a pseudo random number generator algorithm can encounter

Non-Uniform Distributions

- The generation of a uniform deviate is best left to the professionals; we just need to **choose wisely, especially if speed/memory use versus quality is important**
- Often do not want to use a uniform $[0, 1)$ distribution for our random number
- Need transform the uniform deviate to a new deviate (random variable) y which is distributed according to a ***probability density function*** $P(y)$.

General Probability Density Function (PDF) $P(y)$

Must be Positive: $P(y) \geq 0$
and Normalised: $\int_{-\infty}^{\infty} P(y)\, \mathrm{d}y = 1$

We have

PDF for Uniform Deviate in $[0, 1)$

$$U(x)\,\mathrm{d}x = \begin{cases} \mathrm{d}x & 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}$$

This is something that the individual physicist has to implement for their own purposes How would we go about this?

Non-Uniform Distributions

Transformation method

- Want a new random variable y to be a function of x, i.e., $y(x)$.

$$|P(y)\,\mathrm{d}y| = |U(x)\,\mathrm{d}x|$$

- Assuming $\frac{\mathrm{d}y}{\mathrm{d}x} \geq 0$, and since $U(x) = 1$,

$$\frac{\mathrm{d}x}{\mathrm{d}y} = P(y).$$

- Then we can integrate up and define

$$F(y) = \int_{-\infty}^{y} P(\tilde{y})\,d\tilde{y},$$

- and therefore

$$F(y) = \int_{-\infty}^{y} \frac{d\tilde{x}}{d\tilde{y}}\,d\tilde{y} = \int_{0}^{x} d\tilde{x} = x \quad \Rightarrow \quad \boxed{y = F^{-1}(x)}$$

Non-Uniform Distributions

Transformation method

Non-Uniform Distributions

Rejection method

If $P(y)$ cannot be obtained with the Transformation Method, we can use the Rejection Method:

- Define a new function $f(y)$ (the *comparison function*) which satisfies $f(y) \geq P(y)$ for all y
  - this can be anything, so for now, we use a constant $f(y) = C$
- Then follow these steps:
  - Pick a random number y_i, uniformly distributed in the range between y_{min} and y_{max}.
  - Pick a second random number p_i, uniformly distributed in the range between 0 and C.
  - **If $P(y_i) < p_i$ then reject y_i** and go back to the beginning Otherwise accept.

The accepted y_i will have the desired distribution $P(y)$.

Non-Uniform Distributions

Rejection method

Non-Uniform Distributions

Rejection method (properties)

- Efficiency ϵ: $1/\epsilon$ is the

- The ratio of areas under the desired PDF $P(y)$ and the comparison function $f(y)$
- By definition the PDF's area is $A_P = 1$. For the constant case, $A_f = C \times (y_{\text{max}} - y_{\text{min}})$. Thus $\epsilon = \frac{A_P}{A_f}$

$$\epsilon = \frac{1}{C}\frac{1}{y_{\text{max}} - y_{\text{min}}}$$

- The rejection algorithm above effectively randomly picks a pair of coordinates (y_i, p_i) uniformly over the area A_f.
- A fraction of these 'throws' fall outside the area A_P under the PDF, in which case the coordinates are rejected and another throw is taken.

Non-Uniform Distributions

Rejection method (improved)

- Use a comparison function $f(y)$ that conforms more closely to the desired $P(y)$ (but is always greater than P, of course)
- Use this in the transformation method (i.e. it has invertible definite integral $\int_{y_{min}}^{y} f(\tilde{y})d\tilde{y}$)
- The first step changes:
  - ~~Pick a random number y_i, uniformly distributed in the range between y_{min} and y_{max}.~~
  - Using the transformation method, pick a random deviate y_i in the range between y_{min} and y_{max}, distributed according to the *PDF obtained by normalising* $f(y)$.
  - Pick a second random number p_i, uniformly distributed in the range **between 0 and $f(y_i)$.**
  - **If $P(y_i) < p_i$ then reject y_i** and go back to the beginning Otherwise accept.

Monte Carlo Minimisation

We will come back to this once we have covered minimisation more generally

- in the meantime, have a think about how one might use MC methods to help perform minimisation (which, mathematically, is a deterministic problem)
  - the non-minimisation MC techniques that follow should serve as good hints

Monte Carlo Integration

- The integral of a d-dimensional function $f(\vec{x})$ over some volume V defined by boundaries a_i and b_i

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \ldots f(\vec{x}) = \int_V f(\vec{x})\, d\vec{x}$$

- The integral I is related to the mean value of the function in the volume. This can be written as

$$\langle f \rangle = \frac{1}{V} \int_V f(\vec{x})\, d\vec{x}$$

- Then I can be obtained from the mean of the function as

$$I \equiv V \langle f \rangle$$

Monte Carlo Integration

(continued)

- This means we can *estimate* I by taking N random samples of the function in the volume

$$\hat{I} = \frac{V}{N} \sum_{i=1}^{N} f(\vec{x}_i)$$

- Estimate the error in $\hat{I}$ by considering the estimate of the parent variance of the samples $f(\vec{x}_i)$, i.e.,

$$\sigma_{f_i}^2 = \frac{1}{N-1} \sum_{i=1}^{N} (f(\vec{x}_i) - \langle f \rangle)^2$$

- The variance of the mean $\langle f \rangle$ is $\sigma_{\langle f \rangle}^2 = \sigma_{f_i}^2 / N$ and given $I \equiv V \langle f \rangle$, we can write the variance in the estimate of I as

$$\sigma_{\hat{I}}^2 = V^2 \sigma_{\langle f \rangle}^2 = \frac{V^2}{N} \sigma_{f_i}^2$$

Monte Carlo Integration

(continued)

- Therefore we can state the estimate of I (including its error) as

$$\hat{I} = \frac{V}{N}\sum_{i=1}^{N} f(\vec{x}_i) \pm \frac{V}{\sqrt{N}}\sigma_{f_i}$$

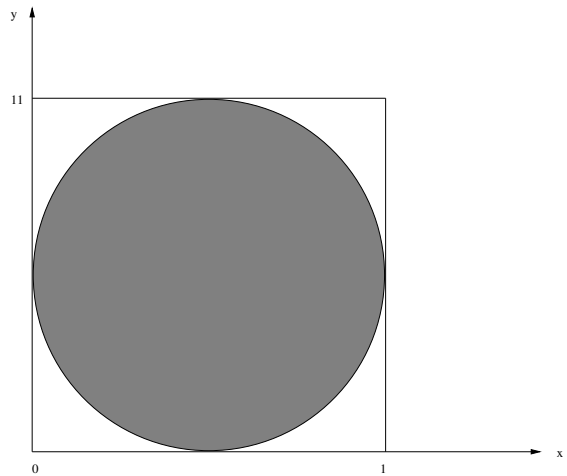
- A key point is that the error in MC integration scales as $\mathcal{O}(\sqrt{h} \propto 1/\sqrt{N})$.

Monte Carlo Integration

A simple example

- Consider the integral of a uniform circle of radius 1/2:

$$f(x,y) = \begin{cases} 1 & \text{if } (x-\frac{1}{2})^2 + (y-\frac{1}{2})^2 < (\frac{1}{2})^2 \\ 0 & \text{otherwise} \end{cases}$$



- In this example, we have the luxury of knowing that

$$I = \int f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \text{circle area} = \pi r^2 = \frac{1}{4}\pi$$

Monte Carlo Integration

A simple example (continued)

- Sample N random, uniformly distributed points in the range $0 \leq x < 1$ and $0 \leq y < 1$.
- $f(x_i, y_i)$ will either be 1 (***success***) or 0 (*fail*) to hit the circle.
- The ***probability of success***, p for any given sample is the fraction of the square (area $A = \int_0^1 \int_0^1 \mathrm{d}x\,\mathrm{d}y = 1$) covered by the circle

$$p \equiv \frac{I}{A}$$

- The successes follow a binomial distribution. Let N_1 be the number of successes
- We can define an estimator for p as $\hat{p} = \frac{N_1}{N}$ The variance of N_1 when binomial is

$$\sigma_{N_1}^2 = N\,p\,(1-p)$$

Monte Carlo Integration

Code snippets

```
totN = 10
def incircle(x, y):
    return (x - 0.5)**2 + (y - 0.5)**2 < 0.5**2

total = 0
accepted = 0
for i in range(totN):
    x = np.random.uniform()
    y = np.random.uniform()
    print x, y
    if incircle(x, y):
        accepted += 1
    total += 1

ratio = (accepted / float(total))

np.set_printoptions(precision=20)
print((accepted/float(total))*4), np.pi
```

Monte Carlo Integration

A simple example

Integration of a radius-$1/2$ circle

- the above example is trivial to extend to the case of the integration of an arbitrary function $f(x, y)$ that is defined in the region defined by $0 \leq x < 1$ and $0 \leq y < 1$
- We can also extend it to any number of dimensions, for example, a sphere in a three-dimensional box
- Another way to find the volume of this sphere was given in lectures, by assigning a value $f(x, y)$ that corresponds to the height of a hemisphere centred on the point $(1/2, 1/2)$

Monte Carlo Integration

A simple example (continued)

- The variance of N_1 when binomial is

$$\sigma^2_{N_1} = N\,p\,(1-p)$$

- We can then have our estimate for the value of the integral I and its error

$$\hat{I} = \hat{p}\,A = \hat{p} = \frac{N_1}{N} \pm \sqrt{\frac{p\,(1-p)}{N}}$$

- Again we see that the error on the integral scales as the square root of the number of samples

Monte Carlo Integration

Comparison to the trapezium method

- Will cover other integration methods fully later in the course
- A linear expansion, so the truncation error goes as $\mathcal{O}\left(h^2\right)$
- For d-dimensions, need to evaluate at $N \sim h^{-d}$ points
- So $h \sim N^{-1/d}$, giving the error:

$$\epsilon \sim \mathcal{O}\left(h^2\right) \sim \mathcal{O}\left(N^{-2/d}\right)$$

- So **for four or more dimensions, the MC method is as accurate or more accurate than the trapezium method,** for the same number of samples:

Error scalings $\longrightarrow$

d	Trapezium	MC
1	N^{-2}	$N^{-1/2}$
2	N^{-1}	$N^{-1/2}$
3	$N^{-2/3}$	$N^{-1/2}$
4	$N^{-1/2}$	$N^{-1/2}$
5	$N^{-2/5}$	$N^{-1/2}$
6	$N^{-1/3}$	$N^{-1/2}$

Taking further advantage of MC methods

The Metropolis Algorithm

- For many practical integrals, the integrand is not "flat" across all space, but rather, highly weighted to particular regions of $\vec{x}$
- More efficient to bias the random samples to where the integrand is not vanishingly small
- For an integral

$$I = \int_V f(\vec{x}) d\vec{x} = \int_V Q(\vec{x}) P(\vec{x}) d\vec{x} = \langle Q \rangle, \textbf{where} P(\vec{x}) \textbf{ is a PDF},$$

  the Metropolis Algorithm can concentrate the samples on where the integrand is largest.
- Instead of completely independent random sampling points, it **guides a "trajectory" of $\vec{x}$ to seek the maximum of $P(\vec{x})$ and "wander about" there**, sampling proportionally to $P(x)$

Taking further advantage of MC methods

The Metropolis Algorithm

- An example is a typical integration in Statistical Physics
- Average a quantity Q over all states $\vec{x}$
- For a classical system where the probability is governed by the Boltzmann energy distribution $P(\vec{x}) \propto \exp\left(-E(\vec{x})/k_B T\right)$, the average over states is

$$\langle Q \rangle = \frac{1}{Z} \int_{\vec{x}} Q(\vec{x}) \exp\left(-\frac{E(\vec{x})}{k_B T}\right) d\vec{x},$$

  where $Z = \int_{\vec{x}} \exp\left(-E(\vec{x})/k_B T\right) d\vec{x}$ is the partition function
- Here, with $P(\vec{x})$ defined as

$$P(\vec{x}) = \exp\left(-E(\vec{x})/k_B T\right)/Z$$

  satisfies the requirements that it be a PDF (see previous slide)
- The Metropolis algorithm has the additional advantage that it does not require you to are able to actually normalise $P(x)$

The Metropolis Algorithm

- Randomly pick a starting point $\vec{x}$
- Repeatedly use the Metropolis algorithm to move to a new point
  - Make a small trial step/change in the parameters to get $\vec{x}'$
  - Calculate $P(\vec{x}')$. **Accept or reject step using**

$$p_{acc} = \begin{cases} 1 & \text{if} \quad P(\vec{x}') \geq P(\vec{x}) \\ P(\vec{x}')/P(\vec{x}) & \text{if} \quad P(\vec{x}') < P(\vec{x}) \end{cases}$$

  (Acceptance means $\vec{x}'$ becomes the next $\vec{x}$)
  - Evaluate $Q(\vec{x})$ and add to running total $\sum Q$ (and $\sum Q^2$ if necessary)
- Repeat whole process for a series of different starting points.
- **The resulting set of values of $Q(x)$ should have sampling points with a density that is proportional to $P(x)$**
- Divide through by the total number of samples to get $\langle Q \rangle$, etc.
- The key is that while the overall trajectory is biased towards areas with a larger $P(\vec{x})$, samples with smaller values are also selected, with a smaller probability, to avoid getting stuck in a local minimum

The Metropolis Algorithm

(continued)

- We are essentially doing

$$\langle \hat{Q} \rangle = \frac{1}{N} \sum_{i=1}^{N} Q(\vec{x}_i) \pm \frac{1}{\sqrt{N}} \sigma_{Q_i},$$

  as before, but with a different strategy for picking the samples.
  - instead of sampling the entire integrand uniformly, we sample Q in a way that is proportional to P.
- Compared to if we had sampled/averaged $f = QP$ directly, the variation in $Q(\vec{x})$ is relatively flat.
- This reduces σ_{Q_i} compared to σ_{f_i} and therefore *makes the error smaller* (for a given number of samples N).
- If started far from equilibrium, the samples obtained during the process of finding the equilibrium skew the result somewhat. They should be omitted unless they are a negligibly small proportion of the total samples.
- Repeating the process for different random starting points is optional but ensures good, even coverage of the integrand.

The Metropolis Algorithm

$$\text{For } I = \int_V f(\vec{x})d\vec{x} = \int_V Q(\vec{x})P(\vec{x})d\vec{x} = \langle Q \rangle, \textbf{where} P(\vec{x}) \text{ isaPDF,}$$

- Randomly pick a starting point $\vec{x}$
- Repeatedly use the Metropolis algorithm to move to a new point
  - Make a small trial step/change in the parameters to get $\vec{x}'$
  - Calculate $P(\vec{x}')$. **Accept or reject step using**

$$p_{acc} = \begin{cases} 1 & \text{if} \quad P(\vec{x}') \geq P(\vec{x}) \\ P(\vec{x}')/P(\vec{x}) & \text{if} \quad P(\vec{x}') < P(\vec{x}) \end{cases}$$

    Acceptance means $\vec{x}'$ becomes the next $\vec{x}$)
  - Evaluate $Q(\vec{x})$ and add to running total $\sum Q$ (and $\sum Q^2$ if necessary)
- Repeat whole process for a series of different starting points.
- **The resulting set of values of $Q(x)$ should have sampling points with a density that is proportional to $P(x)$**
- **Note that "Metropolis Algorithm" refers to what allows points to be sampled in a way that is proportional to a function (here, $P(x)$). Integration is just one example of its use.**

The Metropolis Algorithm

How to pick the next point

- Previously we said, as part of the Metropolis Algorithm:
  - Make a small trial step/change in the parameters to get $\vec{x}'$
- What does this mean?
- Specifically, it means that we define the *proposal density*:

- Typically, we can use a Gaussian that is centred on the current point, $g(\vec{x}', \vec{x}) = N(\vec{x}'|\vec{\mu} = \vec{x}, \vec{\sigma})$
- Then we use the acceptance criteria:

$$p_{acc} = \begin{cases} 1 & \text{if} \quad P(\vec{x}') \geq P(\vec{x}) \\ P(\vec{x}')/P(\vec{x}) & \text{if} \quad P(\vec{x}') < P(\vec{x}) \end{cases}$$

The Metropolis-Hastings Algorithm

The more general case

- For the Metropolis Algorithm, the proposal function is
  - obviously true for a Gaussian
- This is a special case of the Metropolis-Hastings Algorithm, where the proposal function is
- To accommodate this, we modify the acceptance criteria to:

$$p_{\text{acc}} = \begin{cases} 1 & \text{if} \quad A(\vec{x}', \vec{x}) \geq 1 \\ A(\vec{x}', \vec{x}) & \text{if} \quad A(\vec{x}', \vec{x}) < 1 \end{cases}$$

- where A is given by:

Conclusion

We have looked at

- Random numbers in general
- Random numbers in a computer
  - True random
  - Pseudo random
- How to generate pseudo-random numbers
  - Uniform (flat) distributions
  - Arbitrary distributions
- Monte Carlo methods
  - focusing on integration
  - will cover minimisation soon
  - highly non-trivial: algorithms using random numbers can be (much) more effective than deterministic methods!
  - the Metropolis Algorithm