

Lecture 11:

Methods of Optimisation

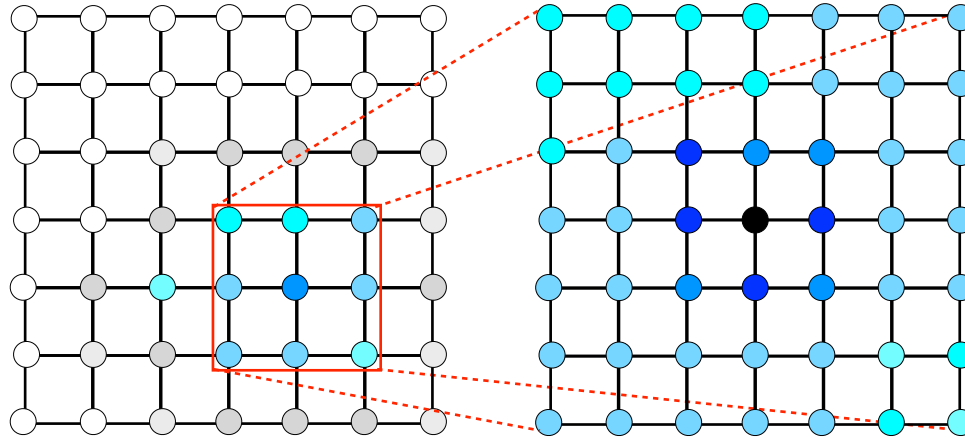
- Grid Searches
- Golden Ratio Linear Search
- Powell's Method
- Gradient Search
- Markov
- Metropolis, Hastings
- Gibbs, Gauss and Hamilton!

Numerical Optimisation (Minimisation/Maximisation)

Simplest - “Grid Search”: Systematically step through possible parameter values on an n-dimensional grid of some pre-defined resolution to find the best values.

Pros: Simple and robust

Cons: Inefficient



Various methods to do “adaptive” grid searches, where you start with a relatively coarse grid and then increase the grid sampling in the region of any identified minima (or overall if no minima found)

Other approaches involve separate line searches in each dimension that are then iteratively applied to find the optimum across multiple dimensions

Golden Ratio Search in 1-D

Assume we have a minimum that has been bounded by 3 points. What next?

Take another sample! Two possibilities:

For making an iterative number of guesses, the most effective search is binary, where the available phase space is divided into equal portions each time:

$$a + b = b + c \quad \longrightarrow \quad a = c$$

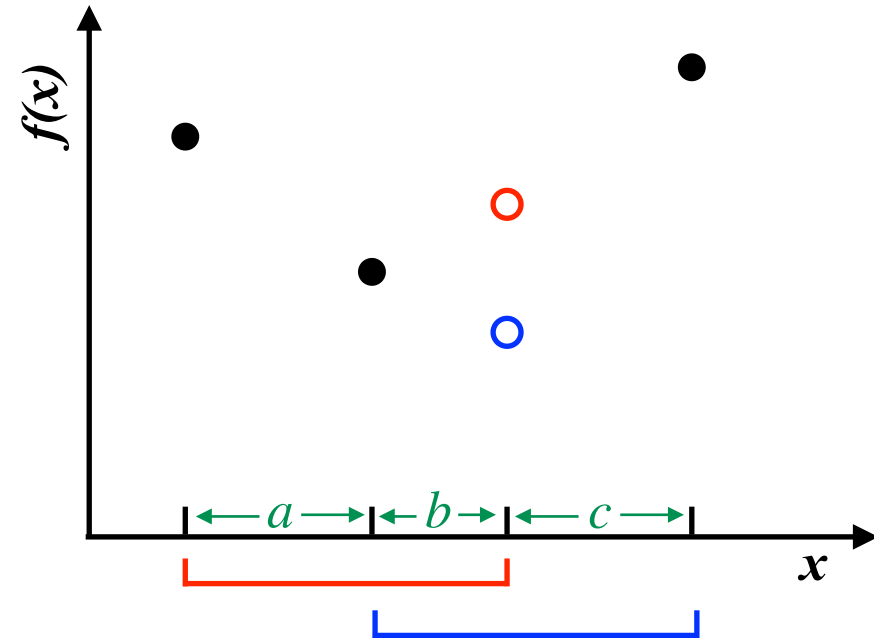
Also assume that we got here from a previous iteration, so:

$$\frac{b}{c} = \frac{a}{b+c} = \frac{c}{b+c}$$

$$\frac{b}{c} \equiv r = \frac{1}{r+1}$$

$$r^2 + r - 1 = 0$$

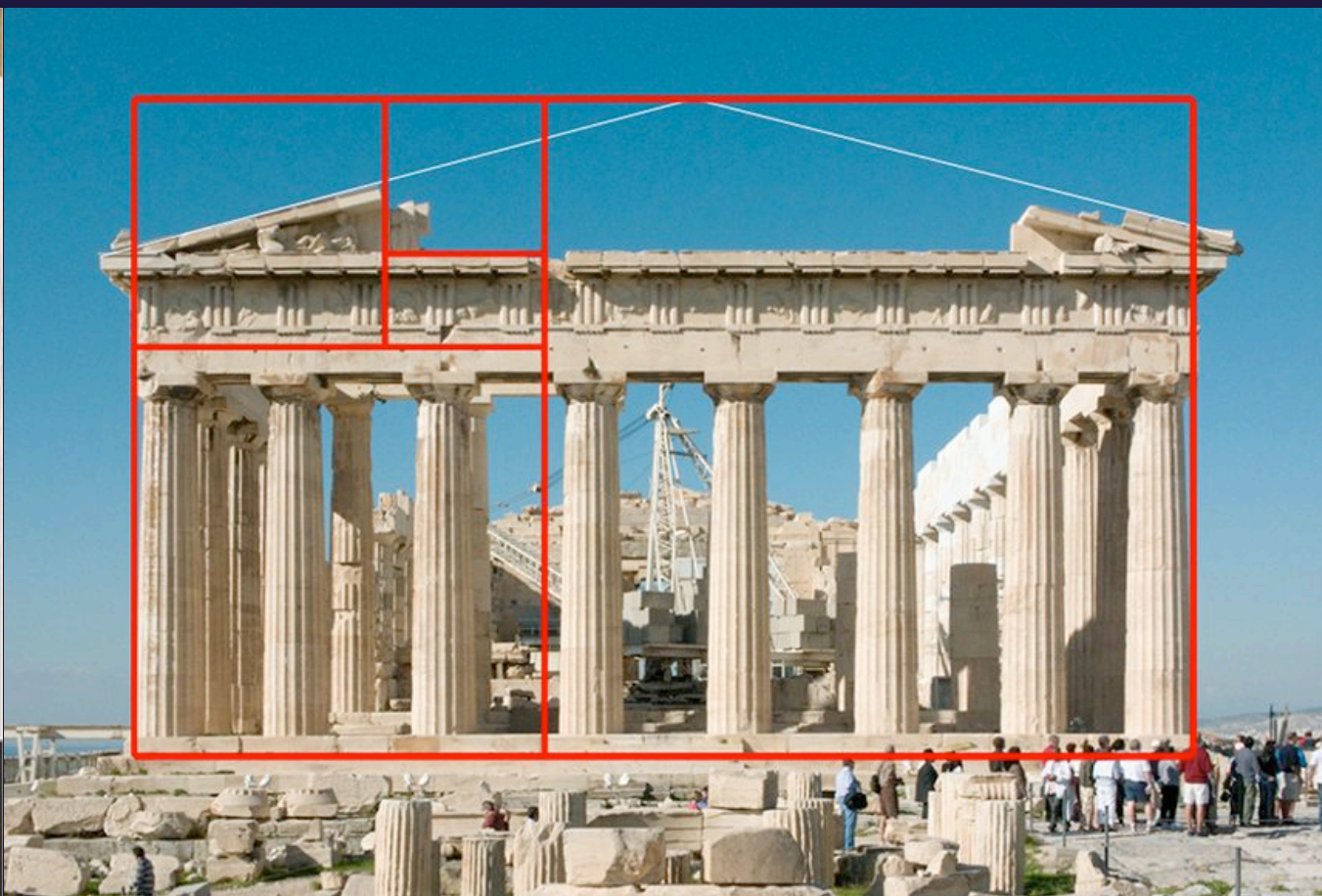
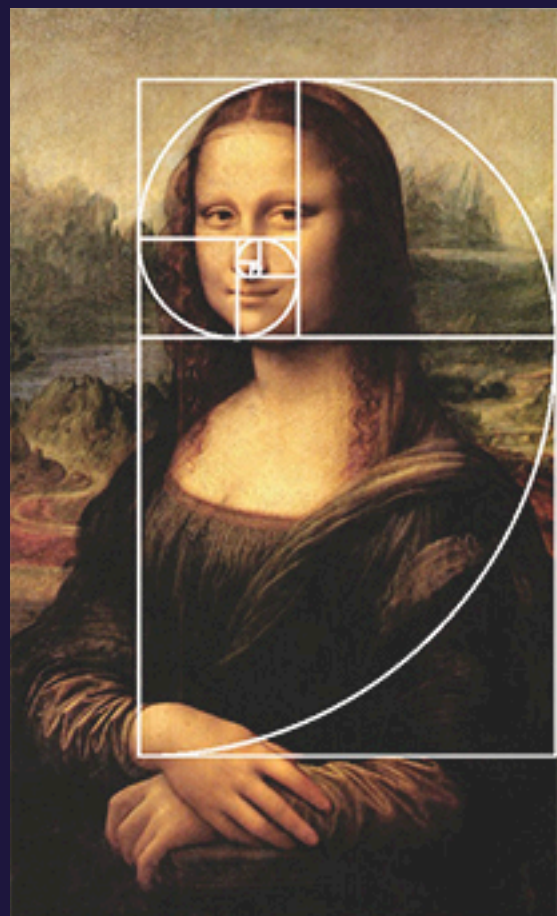
$$r = \frac{\sqrt{5} - 1}{2} \simeq 0.618$$



Golden Ratio

optimal for zooming in
via iterative elimination

(there are also other approaches,
such as parabolic interpolation)



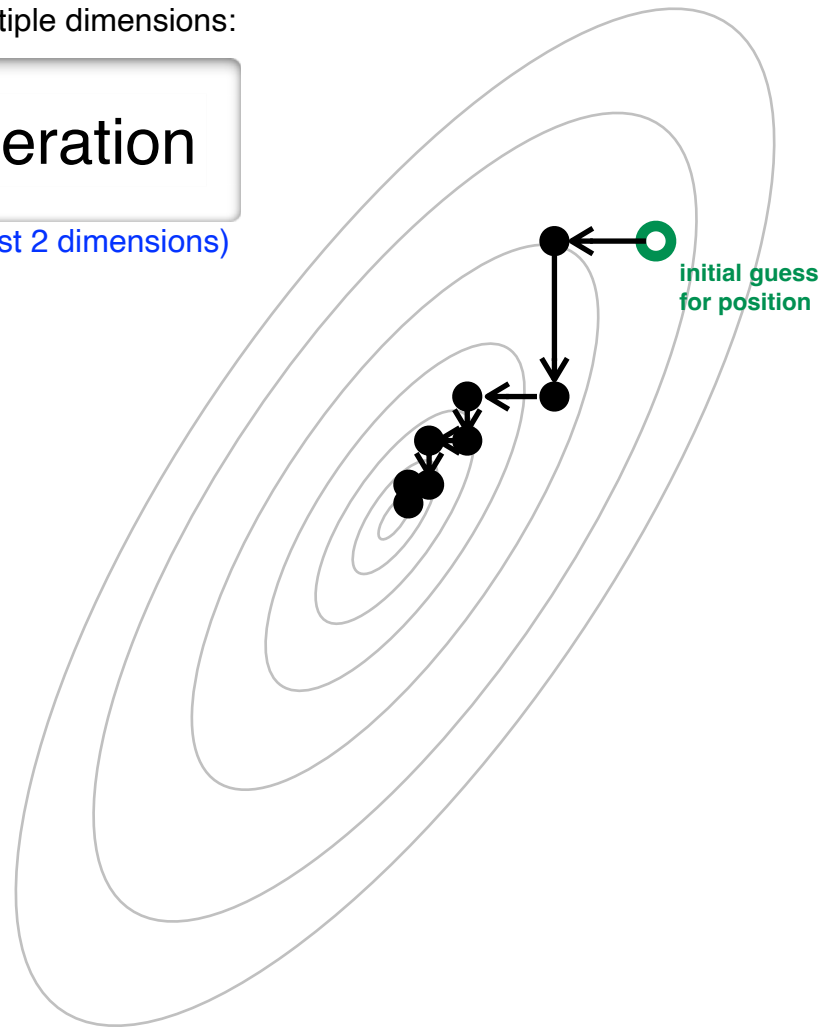
Applying 1-D searches to multiple dimensions:

Brute Force Iteration

(will simplify by considering just 2 dimensions)

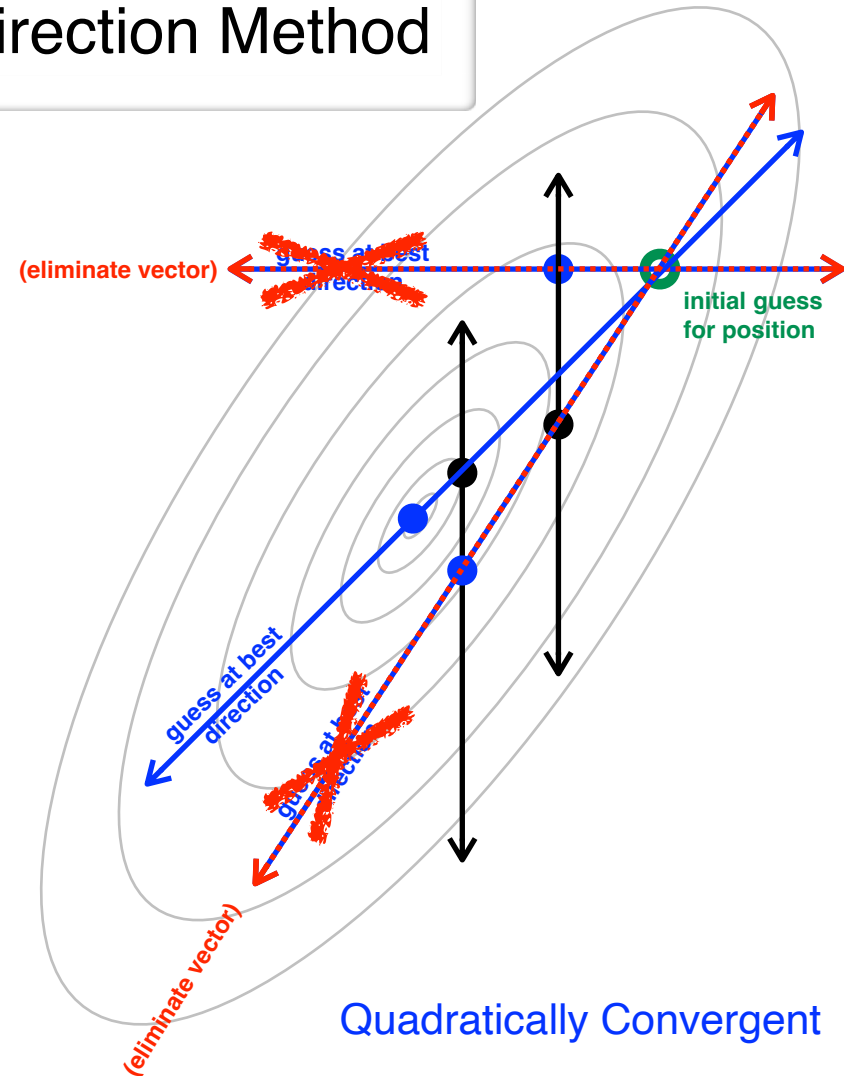
- Guess an initial position
- Cycle through each dimension doing line minimisation
- Choose the best new position each time
- Keep iterating

Not very efficient!



Powell's Conjugate Direction Method

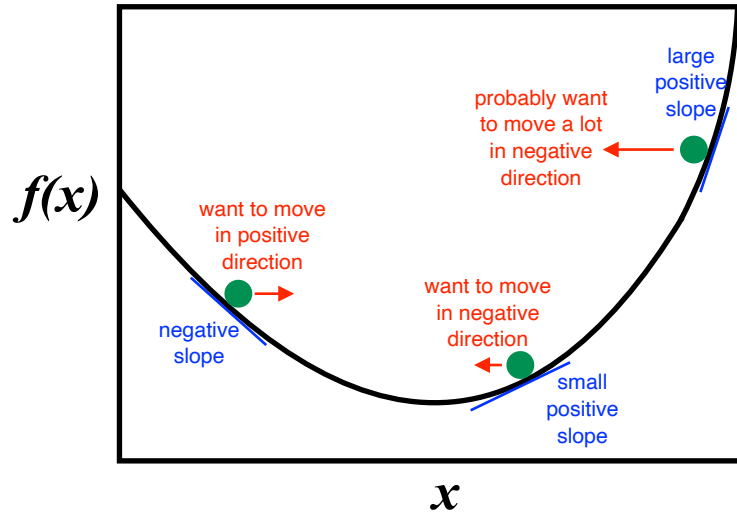
- Guess an initial position
- Cycle through each dimension doing line minimisation and choose the best new position, taking this axis as a guess at the best direction to travel
- Starting from there here, now cycle through every other dimension, doing line minimisation and choose the best new position
- Draw a line from the initial starting point to the current best minimum as the next guess at the best direction, replacing the previous best direction vector. Minimise along this
- Repeat the previous two steps until desired accuracy is reached



Gradient Descent

can be approximated numerically, i.e.

$$\frac{\partial f}{\partial x} \Big|_{x=x_n} \sim \frac{f(x_n + \Delta) - f(x_n - \Delta)}{2\Delta}$$



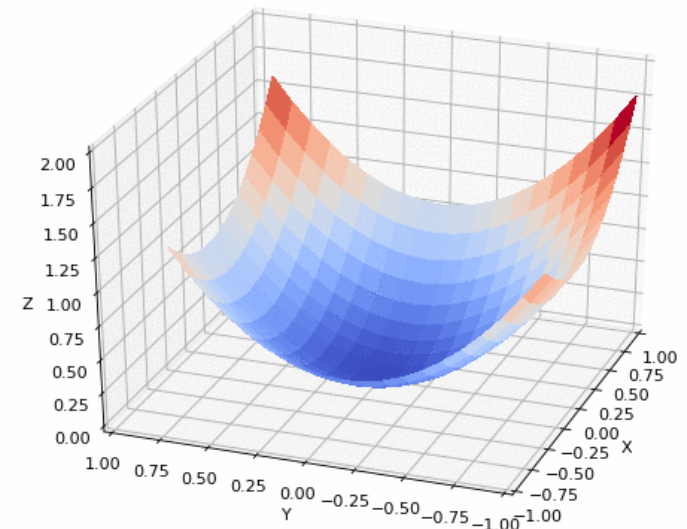
$$\vec{x}_{n+1} = \vec{x}_n - \lambda \nabla f(\vec{x}) \Big|_{x=x_n}$$

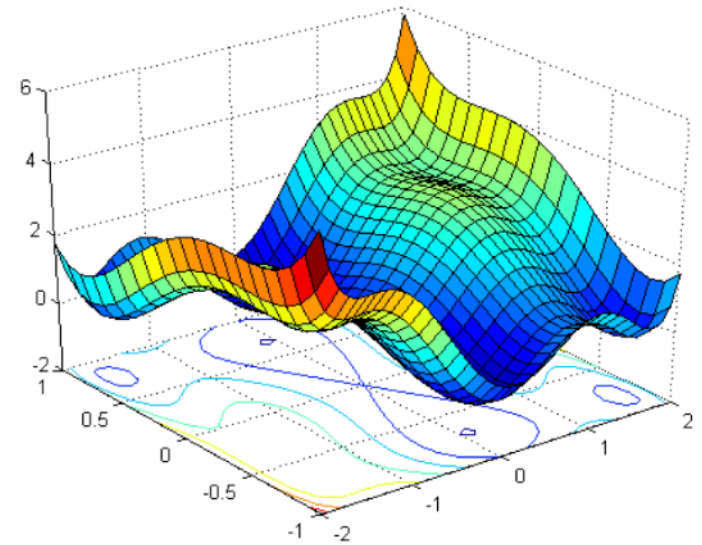
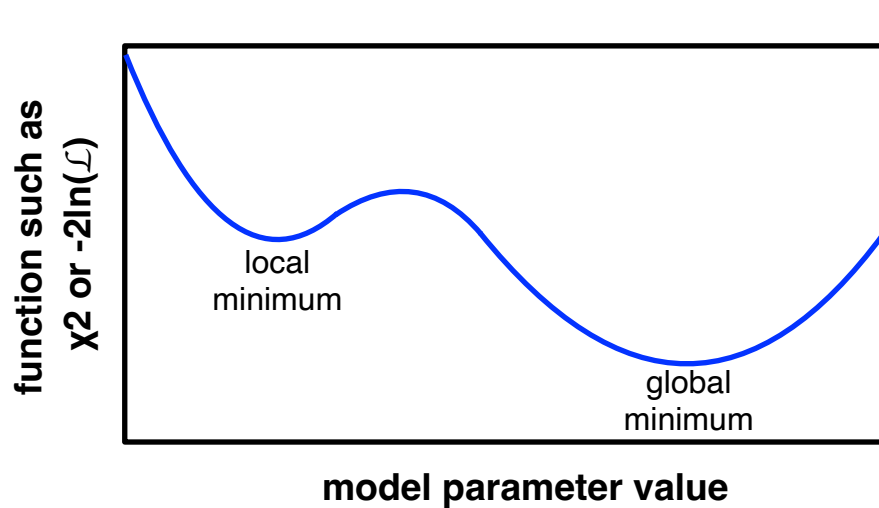
Step sizes:

Needs to be tuned for the particular problem!
("Learning Rate" in ML applications)

Often initially chosen to be some small fraction of the magnitude of the parameter you are trying to constrain. Lots of different approaches, including adaptive algorithms

Start with an initial guess for the parameter values (or "seed") and then progress through parameter space in a direction and with a step size based on successive evaluations of the gradient to follow the path of steepest descent. There is generally some convergence criteria to specify when sufficient accuracy has been achieved and/or when the function evaluations no longer seem to be changing very much (*i.e.* second derivatives are close to zero).





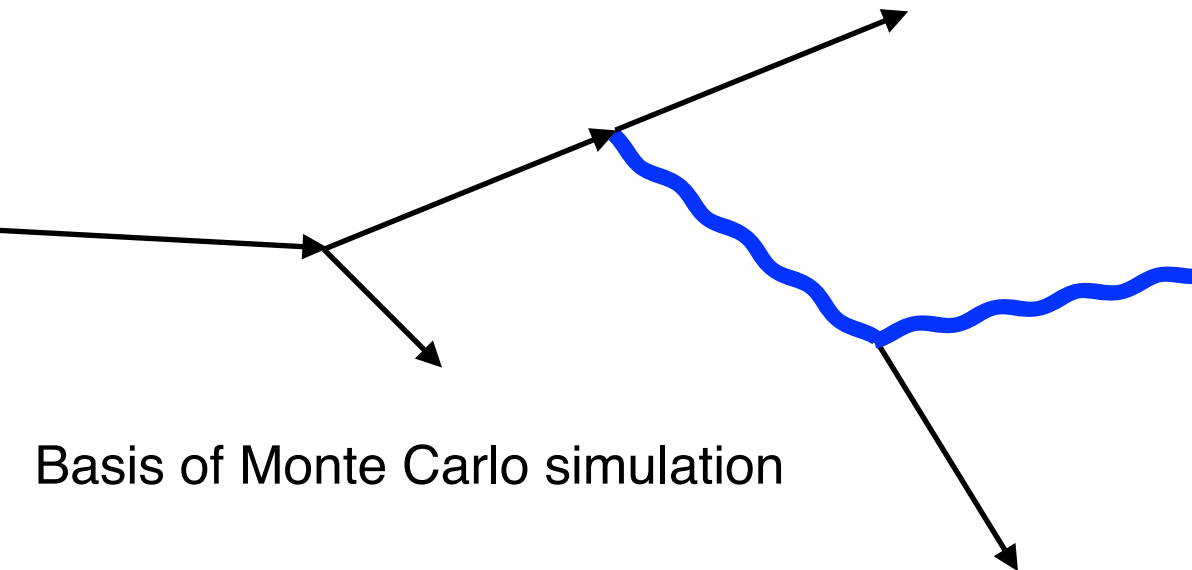
Depending on the nature of the problem, the function space can be irregular and may contain local minima, particularly when dealing with multiple dimensions and parameters have correlations or degeneracies (*i.e.* where different parameter combinations can produce similar solutions). Discontinuities such as “hard” physical boundaries can cause particular problems, as can binned PDFs created with limited statistics. **It's always a good idea to repeat the minimisation with different starting positions!**

Numerous algorithms exist to sample parameter space, bounce out of local minima, smooth out irregularities, deal with hard boundaries, etc. These may makes use of parallel processing, machine learning, Markov chains, simulated annealing... **THIS IS A VAST AREA!**

Always important to look at your parameter space

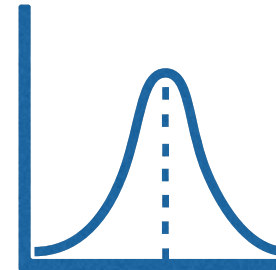
Markov Chain Monte Carlo (MCMC)

A method to numerically integrate over composite functions by probabilistically sampling the function space with a succession of linked iterations

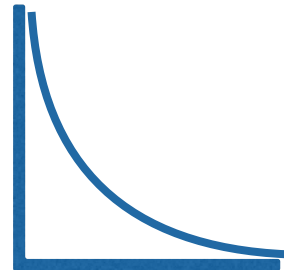


Basis of Monte Carlo simulation

Markov Chain: Each new step only depends on the previous one



Deposited Energy



Penetration Depth

Can also use such an approach to map out the parameter space of a function in the vicinity of its minimum/maximum.

Basic Idea: Ergodically explore the space of model parameters through many “guided” random walks, where movements towards the extremum is encouraged in proportion to the relative posterior probability densities .

A “Sample and Reject” method will be used to conform to these posterior densities.

A **Stationary State** is eventually reached, whereby parameter values wobble around the vicinity of the extremum in a manner that is independent of how a given chain got there.



Provides a robust approach for complex, multi-dimensional parameter space with lots of local minima and maxima.

Computationally intensive, but chains can be run in parallel.

Convergence of chains requires:

- Irreducibility: From any initial state, there is non-zero probability of reaching any other state. This prevents the chain getting stuck in local minima.
- Aperiodicity: The chain must not be periodic. This means the chain never gets stuck in a loop between the same states.
- Recurrence: All subsequent steps sample from the same stationary distribution once it has been reached. This means once a stationary state has been achieved, adding more steps gives a more accurate approximation to the target distribution.

Such chains are 'ergodic'

Say you're at some position, q (a vector of fit parameters), in the function of interest, such as the likelihood. Assume there is some proposed probability, $P(q' | q)$, for jumping to another point, q' .

$$P(q' | q) = \underbrace{g(q' | q)}_{\text{proposal}} \underbrace{A(q' | q)}_{\text{acceptance}}$$

Also apply “Principle of Detailed Balance” to ensure the chain direction is reversible so that we will reach an equilibrium “stationary” state:

$$\rho(q) P(q \rightarrow q') = \rho(q') P(q' \rightarrow q)$$

or

Bayesian
Posterior
probability

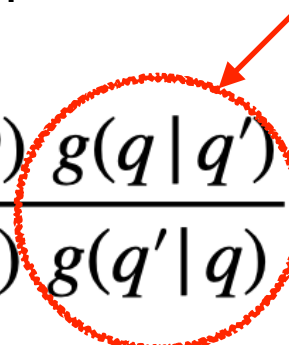
$$\leftarrow P(q | D) P(q' | q) = P(q' | D) P(q | q')$$

$$\alpha(q' | q) \equiv \frac{A(q' | q)}{A(q | q')} = \frac{P(q' | D) g(q | q')}{P(q | D) g(q' | q)}$$

$$= \frac{[P(D | q') P(q')]}{[P(D | q) \underbrace{P(q)}_{\text{Prior}}]} \frac{g(q | q')}{g(q' | q)}$$

Metropolis-Hastings Algorithm

Probability to accept the proposed jump is given by:

$$P_{accept} = \min \left[1, \left(\frac{P(q'|D)}{P(q|D)} \frac{g(q|q')}{g(q'|q)} \right) \right]$$


Hastings bit

i.e. always accept if the new point is better, but potentially accept if the new point is worse based on the balance of relative probabilities (so that you explore the parameter space around the best point).

So throw a random number between 0 and 1, and move to the new point if the number is less than this probability.

Then generate a new proposed position to jump to, and go again...

The frequency of visiting a particular point in the parameter space will be proportional to the overall posterior probability of that point as a solution

Generating Proposals: Gibbs Sampling

$$q = (q_1, q_2, q_3 \dots)$$

$$q'_1 \rightarrow P_{gen}(q'_1 | q_1)$$

$$q'_2 \rightarrow P_{gen}(q'_2 | q_2, q'_1)$$

$$q'_3 \rightarrow P_{gen}(q'_3 | q_3, q'_1, q'_2)$$

$$q'_n \rightarrow P_{gen}(q'_n | q_n, q'_1, q'_2, \dots, q'_{n-1})$$

For independent, Gaussian probabilities, this is simply:

Need to tune step sizes, guided by any parameter constraints: if too large, acceptance will be low; if too small, convergence will be slow

$$q'_1 \rightarrow \frac{1}{\sqrt{2\pi}\sigma_1} \exp\left(-\frac{(q'_1 - q_1)^2}{2\sigma_1^2}\right)$$

$$q'_2 \rightarrow \frac{1}{\sqrt{2\pi}\sigma_2} \exp\left(-\frac{(q'_2 - q_2)^2}{2\sigma_2^2}\right)$$

etc.

Generating Proposals: Hamiltonian Sampling

Analogy with system of particles at some temperature T : particles correspond to the model parameters being fit, temperature allows their values to 'jiggle about' and explore the phase space.

Boltzmann distribution: $P(E) = e^{-E}$ ← units of kT

$$= e^{-(U(q) + KE)}$$

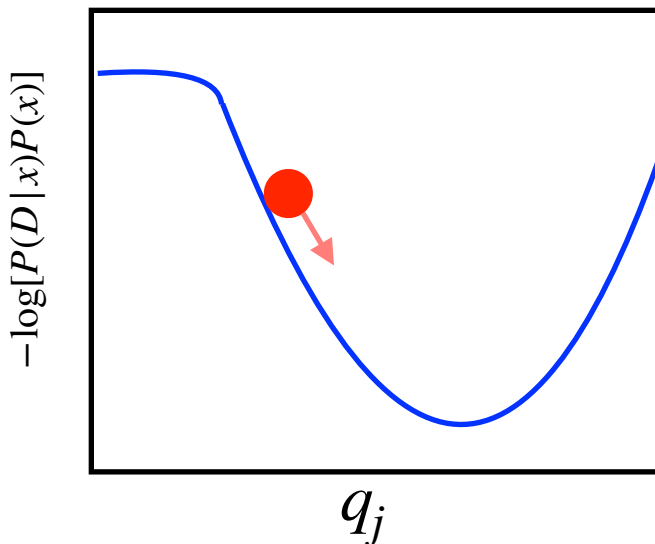
$$= \underbrace{e^{-U(q)}}_{\text{Average probability to be where we are without jiggling:}} \underbrace{e^{-\sum \frac{p_i^2}{2m_i}}}_{\text{Probability for where to move next (proposal)}}$$

Average probability
to be where we are
without jiggling:

$$P(q | D) \propto P(D | q)P(q)$$

Probability for
where to move
next (proposal)

$$U(q) \propto -\log[P(D | q)P(q)]$$



$$P(E) = P(D | q)P(q)e^{-\sum \frac{p_i^2}{2m_i}}$$

← Sample momenta from Gaussian, tuning values of $\sigma_i = \sqrt{m_i}$...as before

However, the subsequent evolution is then defined by Hamiltonian dynamics:

For a conservative system

$$H = E = U(q) + KE = -\log[P(D | q)P(q)] + \sum \frac{p_i^2}{2m_i}$$

$$\frac{dq}{dt} = \frac{\partial H}{\partial p} = \frac{p}{m}$$

$$q \rightarrow q + \frac{p}{m} \Delta t$$

← Tune step sizes

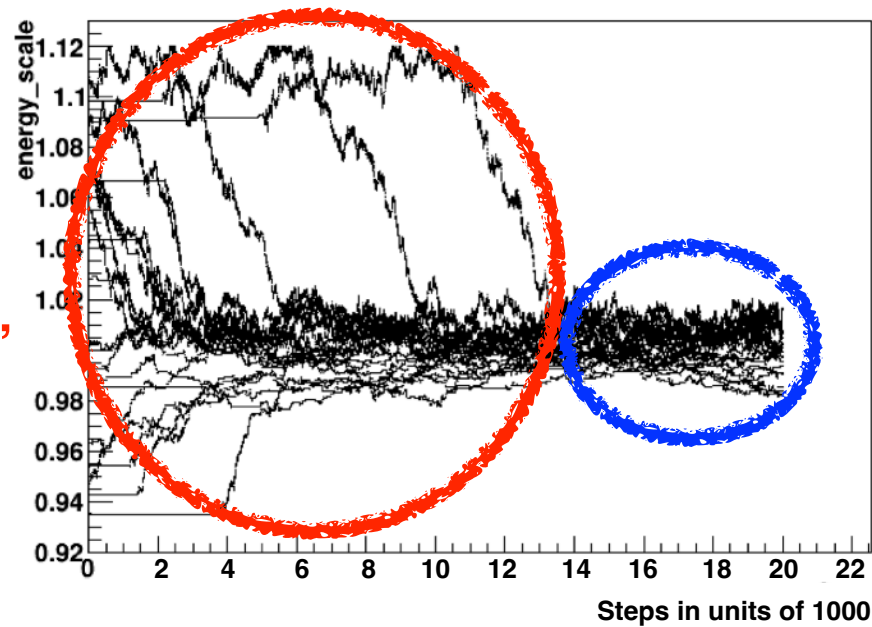
$$\frac{dp}{dt} = -\frac{\partial H}{\partial q} = -\frac{\partial U(q)}{\partial q}$$

$$p \rightarrow p - \frac{\partial U(q)}{\partial q} \Delta t$$

See “Leap Frog” algorithm for how to handle this better (basically uses interleaved average of p to compute next q)

Hamiltonian-guided path is much more efficient for reaching the stationary phase and dealing with discontinuities, but is more computationally intensive for each step. Need to tune algorithm for the specific problem at hand.

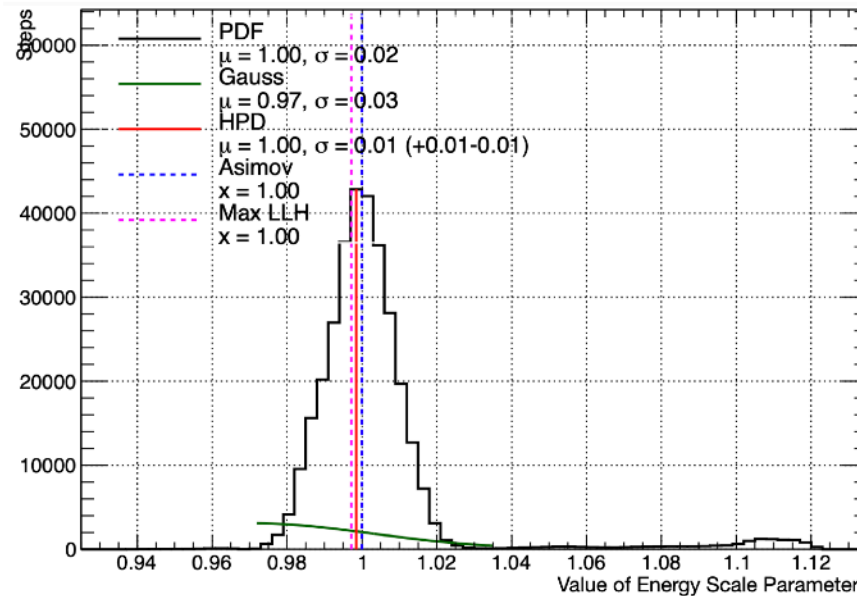
energy_scale:Step



“Burn-In”

MCMC applied to simulated SNO+ data to determine signal content, normalisations to various backgrounds and systematic uncertainties
(thanks to Will Parker)

“Stationary State”

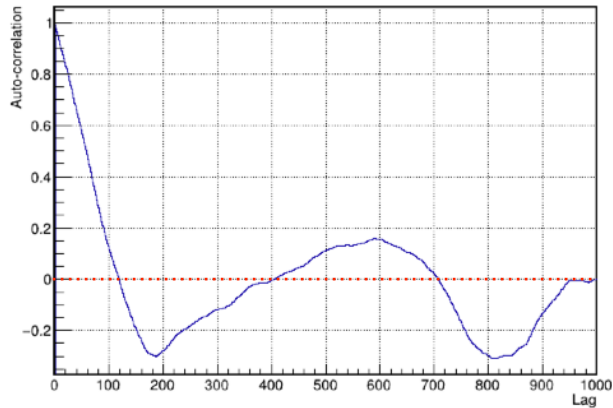


Autocorrelation as Test of Convergence

$$a = \frac{\sum_{i=1}^{N-k} (X_i - \bar{X})(X_{i+k} - \bar{X})}{\sum_{i=1}^N (X_i - \bar{X})^2}$$

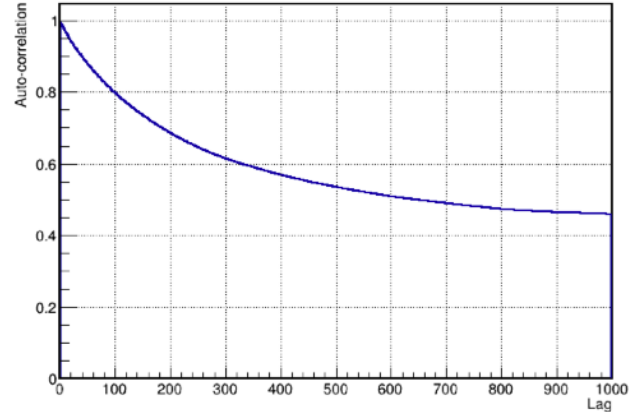
where k = “lag”

b8_numu



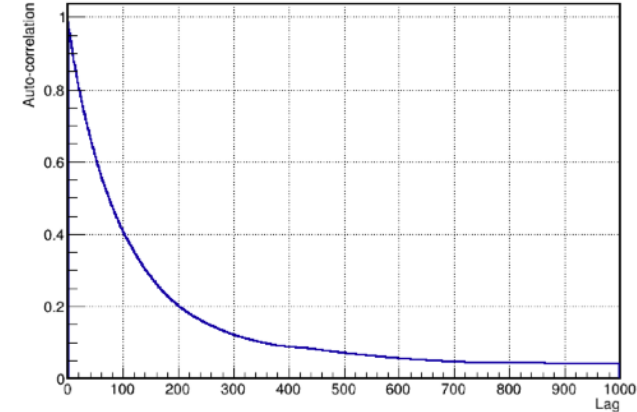
Hasn't converged yet

co60



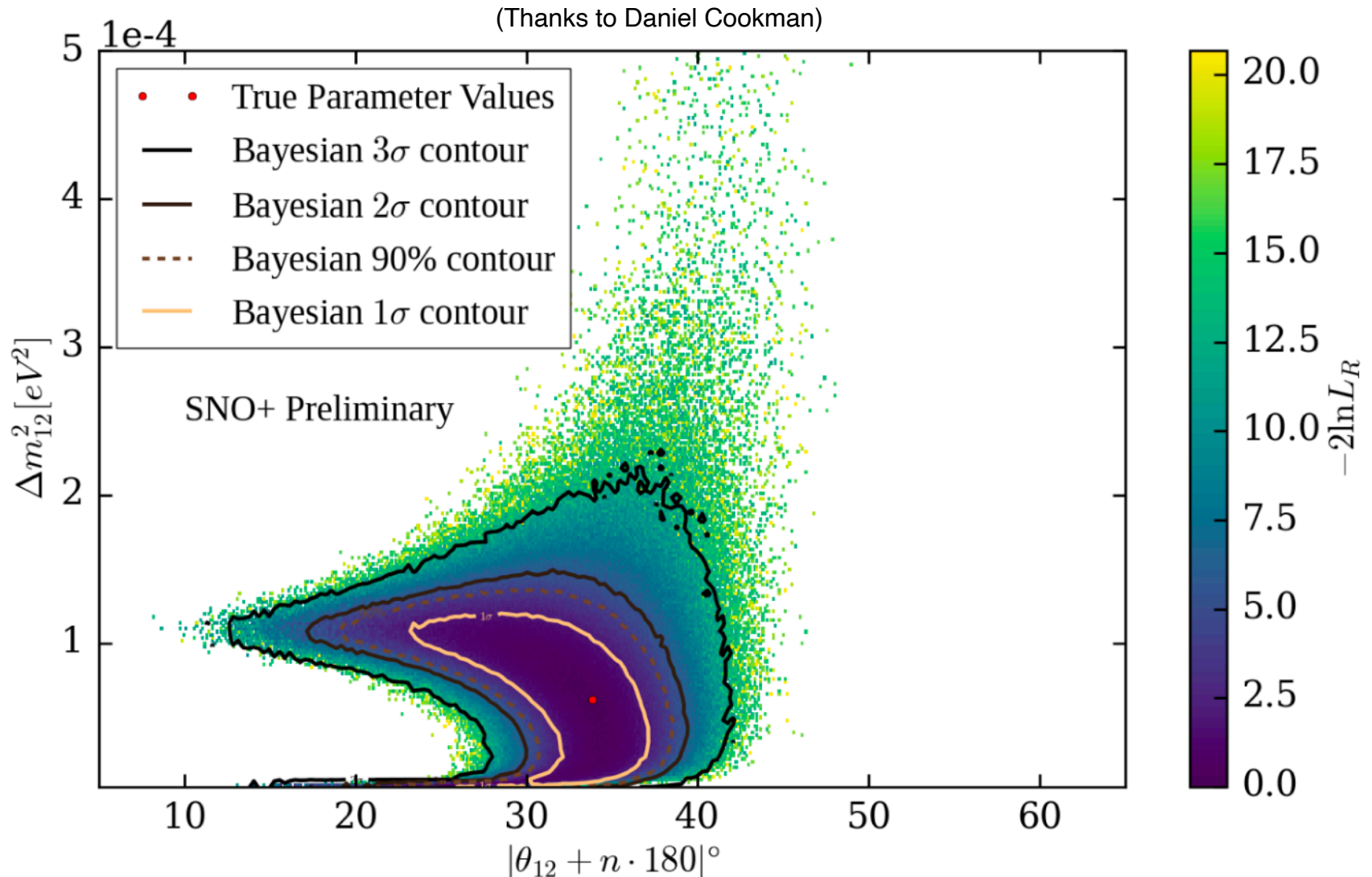
Isn't converging quickly enough
(should probably change the step size)

i130m

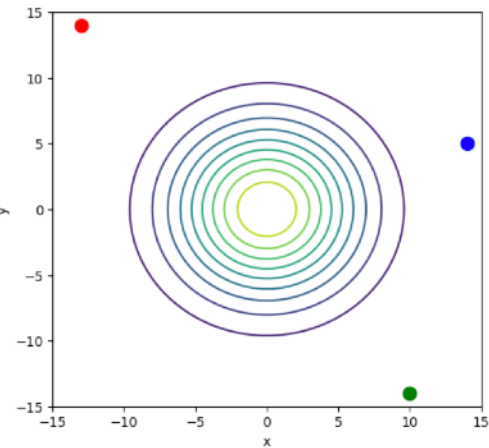


Converging nicely!

Can also use the MCMC information around the stationary phase to produce posterior density maps of the relevant model parameter space (if using parameters with uniform priors, this is also the likelihood map)



A Simple Example of MCMC Optimisation



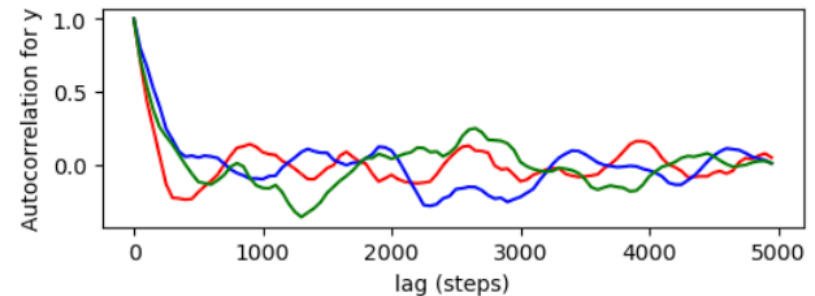
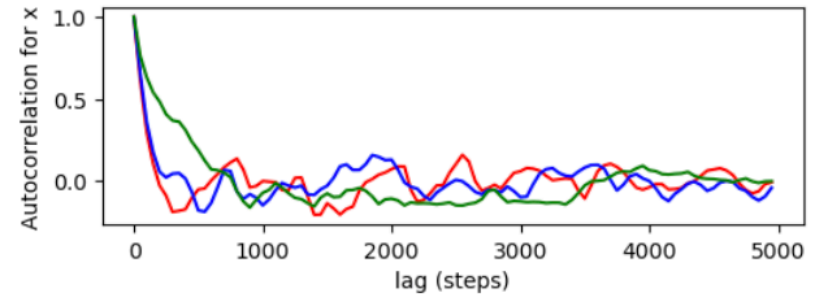
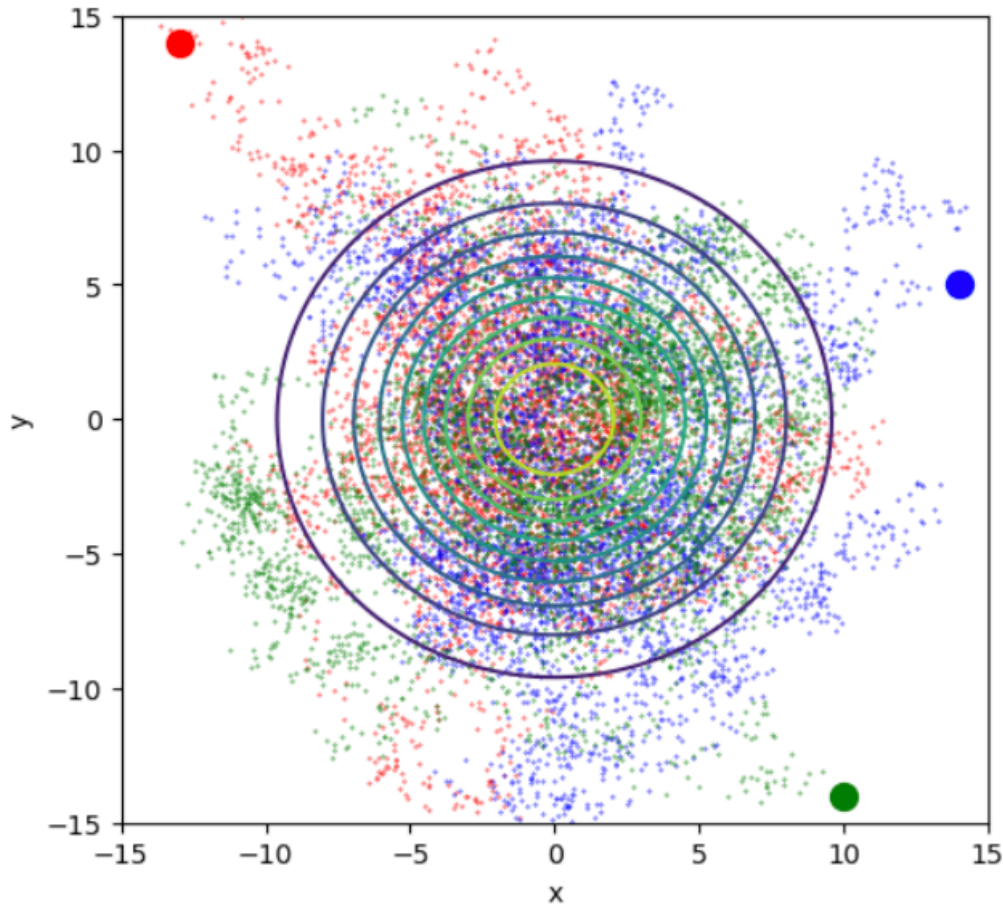
Starting points for chains
shown by coloured dots

```
for ichain in range(3):    # Use 3 chains, defining starting points for each
    if ichain==0: x[0]=-13; y[0]=14; col="red"
    if ichain==1: x[0]=14.0; y[0]=5.0; col="blue"
    if ichain==2: x[0]=10.0; y[0]=-14.0; col="green"
    ax[0].scatter(x[0],y[0],s=100,color=col)
    for istep in range(nsteps-1):    # Step loop
        accept=False    # Initialise acceptance flag
        while accept==False:    # Keep sampling if proposal has not been accepted>
            xprop=random.gauss(x[istep],sigma_x)    # Make proposal for x
            while np.abs(xprop)>15: xprop=random.gauss(x[istep],sigma_x)    # Re-sample if out of bounds
            yprop=random.gauss(y[istep],sigma_y)    # Make proposal for y
            while np.abs(yprop)>15: yprop=random.gauss(y[istep],sigma_y)    # Re-sample if out of bounds
            P_accept=Func(xprop,yprop)/Func(x[istep],y[istep])    # Define acceptance probability
            if np.random.rand()<P_accept:    # Take the step and plot if proposal is accepted
                accept=True    # Set acceptance flag to True
                x[istep+1]=xprop
                y[istep+1]=yprop
```

```
def Auto(x,lag):    # Define autocorrelation function
    N=np.size(x)
    mean=statistics.mean(x)
    Num=0.0
    Den=0.0
    for i in range(N):
        Den=Den+(x[i]-mean)**2
        if i<N-lag:
            Num=Num+(x[i]-mean)*(x[i+lag]-mean)
    return Num/Den
```

```
# Calculate autocorrelation of chains as a function of lag (from 0 up to number of steps taken)
# For calculation efficiency, do this once every hundred steps
dstep=int(nsteps/100)
auto_x=[0]*int(100)
auto_y=[0]*int(100)
step=dstep*np.arange(0,100,1)
for i in range(int(nsteps/dstep)):
    auto_x[i]=Auto(x,i*dstep)
    auto_y[i]=Auto(y,i*dstep)
ax[1].plot(step,auto_x,color=col)
ax[2].plot(step,auto_y,color=col)
```

5000 steps per chain with Gaussian proposal samplings using $\sigma_x = \sigma_y = 0.5$



Comes into its own for complex parameter space
and the ability to run multiple chains in parallel