Random Numbers

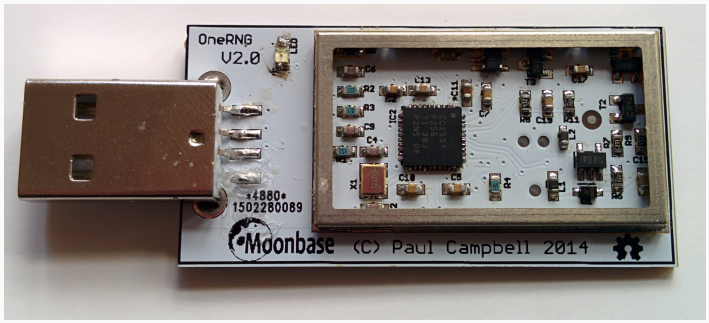
Monte-Carlo methods

We often need to generate random number *sequences* to simulate physical phenomena. Often loosely grouped together as *Monte-Carlo* methods.

- make a decision based on a probability of an event e.g. simulate a random walk, pick random points/directions in space;
- generate simulated noise according to a known distribution.

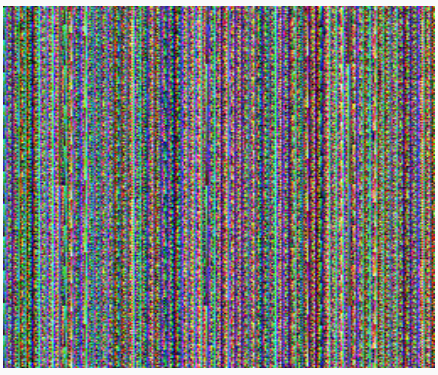
Hardware Random Number Generators

True random number generators (TRNG) use analogue processes e.g. the noise of the microwave background radiation, or a transistor; national lottery machines, etc. Their sequences are *non-deterministic and non-periodic with no correlations* (hopefully).



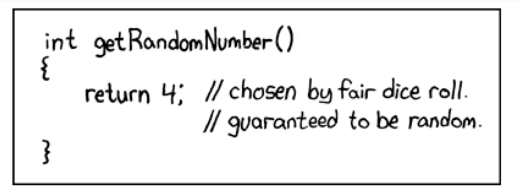
Pseudo-Random Numbers

With computers we often make do with Pseudo Random Numbers Generators (PRNG) which create deterministic sequences of numbers. We need ideally a long sequence with no clustering of values and no short-range or long-range correlations from number to number.



Pseudo Random Number Generators

There is a long and interesting history of PRNG algorithms, and a large technical iterature on the subject. It is easy to make a bad PRNG (and potentially dangerous to do so). Bottom line: Make sure you use a well tested generator for any serious work.



From xkcd

See also: Hill et al. 1999 *How we Learned to Cheat in Online Poker: A Study in Software Security*.

The linear congruential PRNG

- The linear congruential method generates a sequence of integers according to

$$I_{n+1} = (I_n * A + C) \mod M \tag{4}$$

  starting from a seed value I_0. Good choices of C, A and M lead to sequences which have length up to M.
- A good sequence must have uniform probability distribution, and no correlations between successive numbers.
- Remember that the sequence is determined by the seed. Call a generator in a program and you get the same sequence each time unless you change the seed.

Much better algorithms exist. The GSL algorithm has a *period of* $2^{19937} - 1$ *(about* 10^{6000}*) and is equi-distributed in 623 dimensions. It has passed the diehard statistical tests.*

Basic Linear Congruential Generator

Do not use this!

```
// Returns a random integer in the range [0,6074]
int basic_random(int &state) {
  const int A = 107;
  const int C = 1283;
  const int M = 6075;
  state = (state * A + C) % M;
  return state;
}
int main() {
  int state=1234567; // Set the seed
  for (int i = 0; i < 100; i++)
    std::cout << basic_random(state) << "\n";
}
```


Pseudo Random Numbers using GSL

But do use this:

```
#include <iostream>
#include "gsl/gsl_rng.h"
int main() {
  gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
  gsl_rng_set(rng, time(NULL)); // Different seed every time
  for (int i = 0; i < 5; i++)
    std::cout << gsl_rng_uniform(rng) << ", ";
  std::cout << "\n";
  gsl_rng_free(rng);
}
```

```
0.31875, 0.123422, 0.509051, 0.261434, 0.949369,
```

Setting the seed to a fixed value at the start will still give the same "random" sequence every time the program is run. However the repeat period is much longer than for the linear congruential method and long-range correlations between numbers have been stringently tested.

Random Numbers from Given Distributions

Often we want a random deviate y drawn from non-uniform distributions — Poisson, Gaussian, Gamma, ...— described by a probability density $p(y)$, normalised such that $\int p\,\mathrm{d}y = 1$.

Consider $y = f(x)$. Now, y will have a probability distribution such that $|p_y(y)\,\mathrm{d}y| = |p_x(x)\,\mathrm{d}x|$, i.e.

$$p_y(y) = p_x(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right|$$

For example, if $y = -\log(x)$ and the distribution of x is uniform, we have

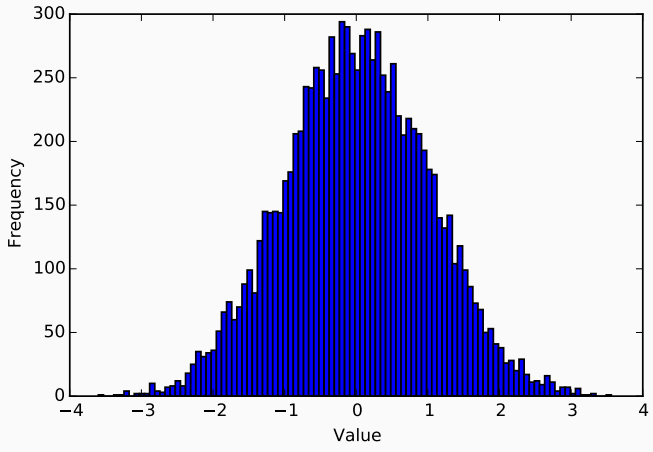
$$p_y(y) = \exp(-y)$$

i.e. y has an exponential distribution.

Other methods to generate arbitrary distributions are available, e.g. the "rejection method".

Most standard distributions are available

```
import numpy.random as random
import matplotlib.pyplot as plt
a=random.normal(size=10000)
plt.hist(a,bins=100)
```



Routines exist also for Poisson, Binomial, etc distributions, and also in GSL. See example `prng3.cc` for an example.

Random Numbers: summary

- “Random number” generators make pseudo-random number sequences
- Same sequence for each seed — beware!
- Use a high-quality library for serious computation — don’t write your own
- GSL or `numpy.random` will probably provide all the generators you will ever need.

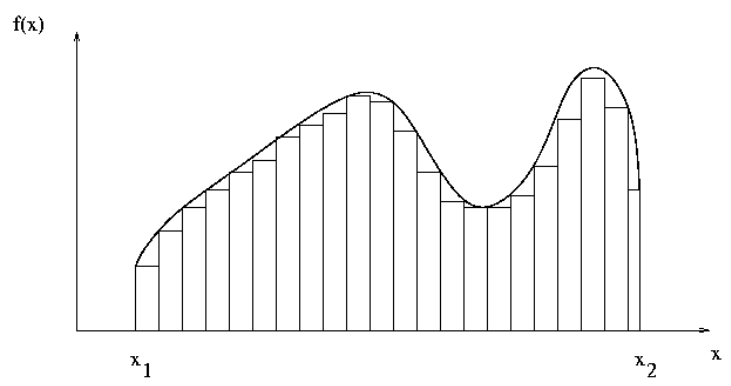
Numerical Integration

Numerical Integration

Many integrals are not analytic. A pendulum swinging with amplitude θ_m has $\ddot{\theta} = -(g/l)\sin(\theta)$. The period of the swing can be expressed as an elliptical integral

$$T = \sqrt{\frac{8l}{g}} \int_0^{\theta_m} \frac{d\theta}{\sqrt{\cos(\theta) - \cos(\theta_m)}}, \tag{5}$$

which must be evaluated numerically. The techniques required have much in common with those developed for ODEs.



- We sample systematically the function at N equally-spaced abcissae, each of which has width $h = (b - a)/N$. The techniques are intrinsically nested in that we can further subdivide each interval to get higher accuracy, and reuse the function evaluations already made.
- Often uses "Newton-Cotes" methods, evaluating the function at regular intervals. For example, sampling at the start of N panels:

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{N-1} f(x_i) \tag{6}$$

- The error per panel is $\mathcal{O}(h^2)$, with $N \propto (1/h)$ panels, yielding a total error $\mathcal{O}(h)$ which is $\mathcal{O}(N^{-1})$. (c.f. Euler method for ODEs).

- As with ODEs, the best routines use higher-order methods. For an n'th order method (Trapezium Rule/Mid Point: $n = 2$, Simpson's Rule $n = 4$) the error is $\mathcal{O}(h^{n+1})$ per panel, yielding a total error is $\mathcal{O}(h^n)$ i.e. $\mathcal{O}(N^{-n})$.
  - So Simpson's Rule error scales like that in the Runge-Kutta 4th order method for ODEs
- There are very clever ways of choosing non-uniform abcissa spacings, called Gaussian quadrature, which can yield very high accuracy for many types of functions.
- GSL or `scipy.integrate.quad()` provides enough for your needs. Simply write an evaluator function and pass this to the integration function.
- Examples `quad1.cc` and `quad.py` are available in examples directory.
- Example class interface in `cavlib` header file `gslint\_integ.hh`

Monte-Carlo Techniques

Monte-Carlo Integration

- Conventional quadrature methods are most efficient in small numbers of dimensions. Higher-dimensional integrals are harder to evaluate accurately by these methods.
- In d dimensions, if we do a total of N function evaluations, we have $N^{1/d}$ panels in each dimension, yielding a total error of $\mathcal{O}(N^{-n/d})$ for an n'th order method. Using Simpson's rule, $n = 4$, and the error is $\mathcal{O}(N^{-4/d})$. As d increases, the number of evaluations N required for a given error increases rapidly.
- We often meet multi-dimensional integrals in physics, e.g. partition function of m particles with potential $\phi(r_{ij})$,

$$Z = \int \cdots \int \exp\left[-\sum_{\text{pairs}} \frac{\phi(r_{ij})}{k_B T}\right] d^3 r_0 \cdots d^3 r_{m-1},$$

  is a $3m$-dimensional integral. If we do 10 evaluations in each dimension we need to make 10^{3m} function evaluations. And the error is $\mathcal{O}(N^{-n/(3m)})$ which falls rather slowly with N.

- Such integrals also arise when doing parameter estimation where we want to integrate over ("marginalise") a likelihood or probability function of many model parameters.
- We might not need a very accurate result.
- Monte-Carlo integration is a simple to code, robust technique which generalises easily to many dimensions.

Monte Carlo Integration

Recall the estimation of π by choosing N random points in a unit square. We expect $\pi/4 = M/N$ where M points lie inside the unit circle. (Coded up in the example `pi.cc`).

In Monte-Carlo integration we use the fact that the integral is equal to the mean function value times the (hyper)volume of the integral.

Rather than systematically evaluating the integrand throughout the volume as in classical quadrature techniques, we can make an estimate of the average by choosing random points throughout the hypervolume:

$$Z = \left\langle \exp\left(-\sum_{\text{pairs}} \frac{\phi(r_{ij})}{k_B T} \right) \right\rangle V^n$$

where V is the three-dimensional volume in this example.

Simple Monte-Carlo Integration Formula

To integrate a function f of n parameters (i.e. an n-dimensional vector $\mathbf{r}$) over a multi-dimensional hypervolume V, we take N samples distributed at random points $\mathbf{r}_i$ throughout the hypervolume V, and use:

$$\int f \, \mathrm{d}V \approx V \langle f \rangle \pm V \left(\frac{\langle f^2 \rangle - \langle f \rangle^2}{N} \right)^{1/2}$$

with

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f(\mathbf{r}_i)$$

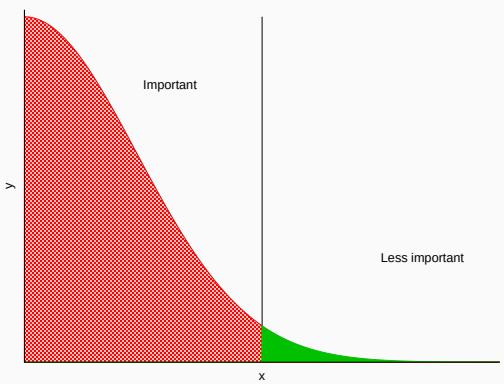
$$\langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f^2(\mathbf{r}_i)$$

Note that this error estimate is not guaranteed to be very good, and is not Gaussian-distributed: treat it as indicative only of the error.

- This is an example where we need extremely good random numbers. We may create many million sample points in the hypervolume V: if there is any clustering along hyperplanes, we will get a biased sampling of the integral and the wrong result.
- Note that the error goes as $N^{-1/2}$ independent of the number of dimensions, so that this method can beat Newton-Cotes quadrature methods in higher dimensions e.g. $d = 8$ and $n = 4$ yields the same $N^{-1/2}$ dependence. For higher d, Monte-Carlo is more efficient.

Reducing Errors with Importance Sampling

- As long as our PRNG is good, our result should converge to the right answer with the error falling off as $N^{-0.5}$
- But convergence can be improved by taking more samples where the integrand's value is larger. Sampling lots of points where the integrand is small is wasteful. (Think about estimating π by placing balls randomly onto a football pitch and counting the ones in the centre circle: it works but it is slow as most balls miss the circle).
- This is generally known as importance sampling, or sometimes variance reduction in the case of MC integration.



Importance Sampling in Monte-Carlo Integration

We don't generally know *a priori* where the integral has a large value, so the algorithm needs to learn about the function as it progresses.

Example Algorithm

1. Divide integration region into 2 disjoint subregions.
2. Use simple MC technique with a small number of samples to estimate value of the integral in each subregion and the estimated error.
3. Use these estimates to decide how to divide optimally the future samples between the two volumes — we will concentrate on the one where the integral has a higher value.
4. Break the volumes up once more, now into 4 subvolumes, and use recursion to break the problem into ever smaller regions, doing most evaluations where the integrand is large.

Monte-Carlo Integration in Practice

- It is easy and instructive to code your own simple Monte-Carlo method (Exercise 1). You can also investigate the error on your estimate by repeating the evaluation n_t ($\sim$ 20?) times using different random points, and estimating the sample variance of these n_t estimates.
- The GSL provides functions for both simple MC and two importance sampling methods, called `gsl_monte_plain_integrate`, `gsl_monte_miser_integrate`, and `gsl_monte_vegas_integrate`.
- `scipy` does not provide an importance sampling algorithm, but there are open-source Python packages to do so.

The Ising Model of Ferromagnetism

An example of real physics insight gained using Monte-Carlo methods.

The Problem

- Ferromagnetism is an intrinsically many-body quantum phenomenon: electron spins align creating a macroscopic dipole moment.
- The magnetisation disappears above some critical temperature.
- How do the spin-spin interactions cause this effect?
- Can we predict the magnetisation **M**, energy E, heat capacity C, and and magnetic susceptibility χ as a function of temperature?

In the Ising model, it is assumed that only adjacent spins interact, and we write the energy of the system as

$$E = -J \sum_{<ij>} s_i s_j - \mu H \sum_{i=1}^{M} s_i \tag{7}$$

where the sum $< ij >$ is over nearest neighbour pairs of atoms only.

We are assuming a two-dimensional array of M spins ($M = N \times N$), so that each spin has 4 nearest neighbours.

J is the exchange energy, μ the magnetic moment, and H the applied magnetic field. We use spin values $s_i = \pm 1$ (could use 1/2 by rescaling J).

More elaborate models treat interactions in more detail (next-nearest neightbours etc) but simple model captures the essence of the behaviour.

Ising Model

- The energy is negative for aligned spins: tendency for alignment to either $s_i = +1$ or $s_i = -1$ for all spins
- But when in contact with a heat bath at temperature T, individual spins will flip back and forth, exchanging energy with the bath.
- With M spins in total there are 2^M microstates which the system can explore. For $M = 32 \times 32 = 2^{10}$ say, we have $2^{2^{10}} \approx 10^{308}$ microstates to explore.
- They are all equally likely but there are too many to compute.
- Onsager developed an analytic solution to the Ising model in 2D by evaluating the partition function directly, but generally analytic techniques are intractable. Need to simulate, but how?

The Metropolis Algorithm

We can't compute all the microstates — there are too many! But the probability P of finding a system in a state with energy E should be proportional to the Boltzmann factor

$$P \propto \exp\left(-\frac{E}{k_B T}\right) \tag{8}$$

We want our simulation to generate a sample of the microstates with this probability distribution. We can then derive thermodynamic quantities by a simple average over the microstates visited by the simulation. Metropolis et al (1953) invented just such an algorithm (N.C. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, J. Chem. Phys. 21 (1953) 1087-1092)

"Instead of choosing configurations randomly, then weighting them with exp(-E/kT), we choose configurations with a probability exp(-E/kT) and weight them evenly."

Metropolis Algorithm (aka Metropolis-Hastings algorithm)

- Given a system in a known microstate A with energy E_A, consider a nearby microstate B of energy E_B.
  - if $E_B < E_A$, allow the system to make a transition to the new microstate (because it has lower energy);
  - else if $E_B > E_A$, either make the jump with a probability $p = \exp(-(E_A - E_B)/(k_B T))$

So as the temperature increases, the system can explore more distant (higher energy) states.

The algorithm works (that is, is thermodynamically sensible) because it encodes the principle of detailed balance between states when thermal equilibrium applies: $p_A T_{AB} = p_B T_{BA}$ where T_{AB} is the transition rate $A \to B$.

$$\frac{p_A}{p_B} = \frac{T_{BA}}{T_{AB}} = \frac{\exp(-E_A/(k_B T))}{\exp(-E_B/(k_B T))} \tag{9}$$

This is another example of importance sampling.

Ising Model using Metropolis Algorithm

1. Create a system of spins in a particular microstate.
2. Repeat a large number of sweeps through the lattice. For each sweep (a single Monte-Carlo "step"):
   - Select a lattice site, either randomly or in a sequence
   - Find the energy ΔE required to flip the spin at the given lattice point using Equation 7
     - if $\Delta E < 0$, flip the spin;
     - else if $\exp(-\Delta E/(k_B T)) > p$, where p is a uniform random number in the range $[0, 1]$, then flip the spin;
     - else do nothing.
   - Repeat for another lattice site, until you have visited all the sites in turn, or visited a large fraction of the total number of sites.
   - Record relevant variables (energy, magnetisation,...), plot, save state.

Some technicalities

1. Spins on the lattice edges don't have 4 neighbours, and this will introduce artefacts. It is easiest to use periodic boundary conditions, so that every spin has 4 neighbours: imagine the lattice on a torus. (This is easy to implement by carefully looking after the indexing of the array used to store the spins.)
2. You need to let the simulation run for a 'long time' to ensure equilibrium is approached. How long will depend on the initial set up.
3. Visiting lattice sites randomly or in a regular sequence will both work, but the time to approach equilibrium will vary again depending on inital conditions and temperature.
4. The compute time varies steeply with N, but desktop PCs should be able to handle $N \sim 10 \times 10$ and somewhat larger in reasonable time. Memory requirements are small as long as the data are not stored.
5. Numerical results will depend on N; should investigate how...
6. Easy to generalise to higher dimensions, or add more interactions.

Other Applications using Monte-Carlo Methods

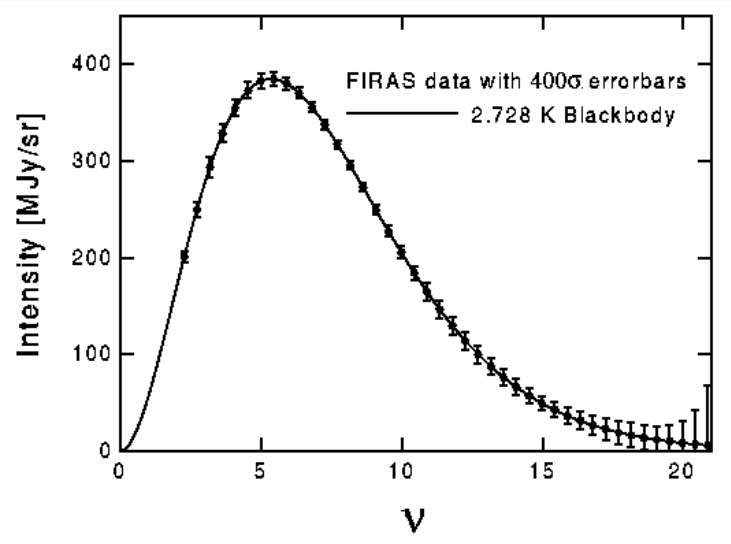
1. Estimating the error on parameters determined by experiment.
2. Solving complex systems of differential equations. For example, radiative transfer codes in 3-dimensions.
3. Random walks e.g. polymer folding
4. "Travelling Salesman"/simulated annealing problems

Many others...

Fitting Data

Inference: Fitting Models to Data

Given some measurements, what can we learn about the world? Often we fit a theoretical model to data to see (a) if the model is valid and (b) constrain any parameters of the model.



Linear regression and chi-squared

Given N data points y_i sampled at points x_i with errors σ_i, find "best" values of m and c in a straight-line model Y:

$$Y_i = mx_i + c$$

We often use a goodness of fit statistic called chi-squared:

$$\chi^2(m,c) = \sum_{i=0}^{N-1} \left(\frac{y_i - mx_i - c}{\sigma_i} \right)^2$$

and choose m, c which minimises χ^2. What we are doing is maximising the likelihood of the data given the model:

$$\begin{aligned} L = Pr(\{y_i\} \,|\, m, c) &\propto \prod_{i=0}^{N-1} \exp\left(-\frac{(y_i - mx_i - c)^2}{2\sigma_i^2} \right) \\ &= \exp(-\chi^2/2) \end{aligned}$$

Likelihood and Bayesian inference

Note that what we really want to know is the inverse of this, i.e. the probability of our model parameters m, c given the data y_i, which is proportional to the likelihood:

$$Pr(m, c \,|\, \{y_i\}) \propto Pr(\{y_i\} \,|\, m, c) Pr(m, c)$$

This is Bayes' Theorem — beyond the scope of this course, but Bayesian data analysis is becoming extremely popular in many areas of science, and provides a consistent framework for inference.

For the rest of this analysis, we assume that the *prior* $Pr(m, c)$ is uniform and so the least-squares solution gives us the model parameters which maximise the *posterior* probability $Pr(m, c \,|\, \{y_i\})$.

General least-squares fitting

The least-squares solution for fitting a straight line can be derived analytically. A more general model could have M parameters $\theta = \{\theta_i\}$, $i = 0 \ldots (M-1)$, so that χ^2 becomes:

$$\chi^2(\{\theta_i\}) = \sum_{i=0}^{N-1} \left(\frac{y_i - f(\{\theta_i\}; x_i)}{\sigma_i} \right)^2$$

where the function f expresses the model.

We now need to find the minimum value of χ^2 as a function of the M parameters θ_i.

General linear least squares

There are many cases where the model is not a straight line, but nevertheless problem is linear in the model parameters. The most common case is where the model can be expressed as

$$y(x) = \sum_{k=1}^{M} \theta_k \phi_k(x)$$

where $\{\theta_k\}$ are the model parameters and $\{\phi_k(x)\}$ are the basis functions for the problem and can be *non-linear* functions of x.

Matrix formulation

For this case, we require a least-squares solution to the linear problem

$$\mathbf{A}\theta = \mathbf{b} \tag{10}$$

where A is the design matrix for the problem

$$A_{ij} = \frac{\phi_j(x_i)}{\sigma_i}$$

and

$$b_i = \frac{y_i}{\sigma_i}$$

Note that in general the problem is overdetermined and $\mathbf{A}$ is not square.

Singular value decomposition

There are multiple ways to solve this linear least-squares problem, but the most robust involves the use of Singular Value Decomposition (SVD).

SVD decomposes an arbitrary matrix $\mathbf{A}$ into three matrices $\mathbf{U}$, $\mathbf{V}$, and w such that

$$\begin{pmatrix} & & \\ & \mathbf{A} & \\ & & \end{pmatrix} = \begin{pmatrix} & & \\ & \mathbf{U} & \\ & & \end{pmatrix} \cdot \begin{pmatrix} w_1 & & & \\ & w_2 & & \\ & & \cdots & \\ & & \cdots & \\ & & & w_N \end{pmatrix} \cdot \begin{pmatrix} & & \\ & \mathbf{V}^T & \\ & & \end{pmatrix}$$

The matrices $\mathbf{U}$ and $\mathbf{V}$ are *unitary* i.e. $\mathbf{U}^T \cdot \mathbf{U} = \mathbf{I}$ and $\mathbf{V} \cdot \mathbf{V}^T = \mathbf{V}^T \cdot \mathbf{V} = \mathbf{I}$ and the diagonal matrix $\mathbf{w}$ contains the so-called singular values w_j.

The Moore-Penrose pseudo-inverse

The least-squares solution to equation (10) is given by

$$\theta = \mathbf{A}^{+}\mathbf{b} \tag{11}$$

where $\mathbf{A}^{+}$ is the pseudo-inverse of $\mathbf{A}$ given by

$$A^{+} = \mathbf{V} \cdot [\mathrm{diag}(1/w_j)] \cdot \mathbf{U}^{T} \tag{12}$$

Small or zero values of w_j indicate a singularity/degeneracy in the problem and in this case $1/w_j$ should be replaced by zero.

The SVD-derived matrix inversion is very robust — it tells us when the problem is malformed and can still give sensible solutions. There are many more uses for SVD — see “Numerical Recipes”.

Numpy example

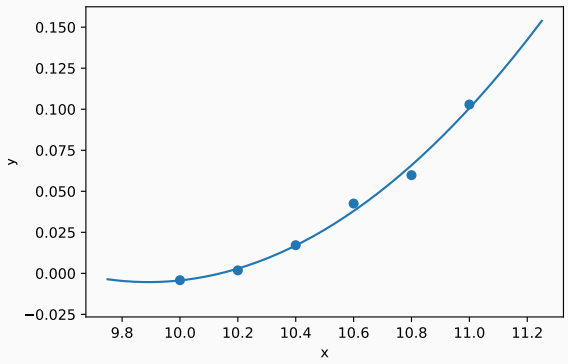
```
import numpy as np
import matplotlib.pyplot as plt
ndata=6
x=np.linspace(10,11,ndata)
y0=0.1*x**2-2*x+10  # noise-free data
y1=y0+np.random.normal(0,0.01,ndata) # noisy data
# Set up design matrix, assume unit errors
A=np.ones((ndata,3))
A[:,0]=x**2
A[:,1]=x
# Pseudo-inverse
Ainv=Np.linalg.pinv(A)
theta0=np.dot(Ainv,np.transpose(y0))
theta1=np.dot(Ainv,np.transpose(y1))
print(theta0,theta1)
```

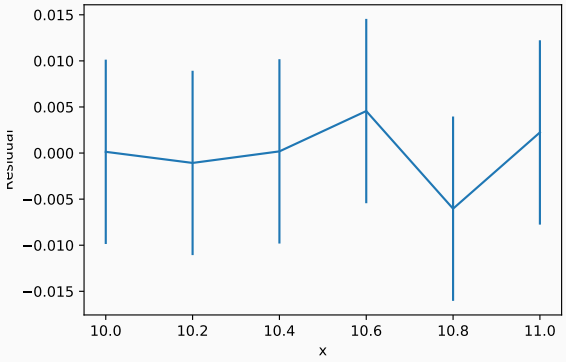
```
 [ 0.1 -2.  10. ] [ 0.086233   -1.70593319  8.43171695]
```

Numpy example (cont'd)

```
px=np.linspace(9.75,11.25,200)
plt.scatter(x,y1)
plt.plot(px,theta1[0]*px**2+theta1[1]*px+theta1[2])
plt.ylabel("y")
plt.xlabel("x")
```

```
plt.errorbar(x,y1-(theta1[0]*x**2+theta1[1]*x+theta1[2]),yerr=0.01)
plt.ylabel("Residual")
plt.xlabel("x")
```

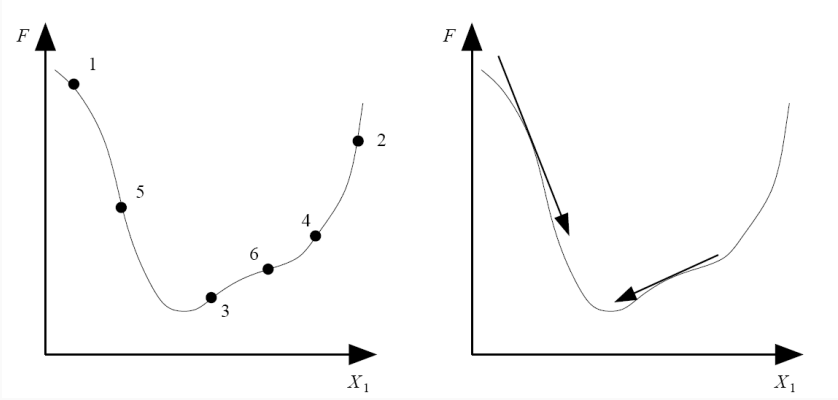




Non-linear problems: minimisation

The most general least-squares problem is the multi-dimensional minimisation of the function $\chi^2(\theta)$, possibly subject to some constraints (e.g. "all the θ_i are positive", or $\theta_1 + \theta_2 > 0$.).

If we have no derivative information, we use a bracketing method (left). But if we have derivative information, the convergence is usually much more rapid (right).



Beware local minima — can be hard to treat in general.

Minimisation in Multiple Dimensions

Many algorithms developed to minimise functions of many variables. Don't code your own! Specialised techniques for least squares also exist, and they often distinguish between linear and non-linear models

The Levenberg-Marquardt algorithm is often used for minimisation. Typical implementations of this algorithm require the user to provide functions that evaluate the model i.e. $f(\theta_i, x_i)$, and the gradient information $df(\{\theta_i\})/d\theta_j$ (the **Jacobian** matrix). See Numerical Recipes for details.

The coding can be a little elaborate in **C++**, but I have provided an example that fits a polynomial by least-squares to a set of data in `fitpoly.cc`. It uses the Levenberg-Marquardt algorithm found in the **GSL** routines. There are a range of least-squares functions in `scipy.optimize` and also in other parts of `scipy` which tackle linear and non-linear data fitting problems.

Least-squares notes

For linear models, it can be shown that χ^2 follows the chi-squared distribution with $\nu = N - M$ degrees of freedom. For large values of ν, χ^2 has mean ν and standard deviation $\sqrt{(2\nu)}$.

Rule of thumb: expect $\chi^2 \sim (N - M)$ for a good fit. If we don't know $\{\sigma_i\}$, we sometimes assume the best fit has $\chi^2 = (M - N)$ then estimate σ, assuming it is the same for all data points.

We usually need the errors on the fitted parameters, both the standard deviation (error) on each parameter, and the co-variances. Estimates of the covariance matrix are returned by most quality algorithms.

Programming

Practical programming

We want to write programs which have the following attributes (in priority order):

- They produce results which are reliable.
- They produce results of acceptable accuracy.
- They produce results in an acceptable amount of time.

The process of writing programs which achieve these aims is helped by using some best practices:

- Break the problem into small, well-defined parts.
- Write readable code.
- Test the parts.
- Test the whole.

Program design

- Most people start out writing code using the unstructured programming style:
  - Put everything in the main program.
  - All the data are defined there and used when needed.
- This is tolerable for “Hello, World” programs, say up to 20-30 lines, and probably OK for the class exercises, but not much more.
- It is much better to use a procedural programming style: we write functions/methods/procedures which solve small, well defined parts of the problem and combine these to solve the problem as a whole.
- This helps us to deal with complexity: smaller functions are both easier to comprehend and easier to test.
- Try to reduce hidden coupling between functions (e.g. global variables, other “side effects”) as this increases complexity and decreases comprehensibility.

Writing readable code

There is a world of difference between good and bad code, and there are many books/webpages about ’good’ programming style, but in brief:

- Write your code primarily to explain to humans (for example, your future self) what you are trying to do; a secondary aim is to explain to the *computer*.
- Use names consistently and with meaning. Use descriptive variable names for important quantities, e.g. `calculate_derivative( double x )` or `calculateDerivative( double x )` ; trivial loop variables may not always need a long name though.
- Use comments where they add value. Too many comments can distract the reader from what the code itself is saying.
- Make the code neat and tidy. Line it all up in columns — editors (e.g emacs) do this for you. Use whitespace generously (I have squeezed my examples for display purposes...).

- Avoid ‘hard-wired’ numbers in your code. Use names for these even if they are trivial — they will enhance readability. e.g.

```
const double rad_to_degree = C::pi/180.0;
x *= rad_to_degree;
```

  is much better than

```
x /= 57.2958;
```

- Aim for general, re-usable code. Avoid fixed-sized arrays, e.g. in Ising model, your code should have as a parameter the number of lattice sites on each side — don’t hard-wire this number into the code.

Testing

- Review what you have written for correctness.
- Test early: write small functions and test that these work.
- Test often: put the functions together and test the result.
- Test by running the code in situations where the answer is known.
- You will find bugs. Use debugging tools:
  - Print out intermediate results
  - Divide and conquer
  - Learn to use a debugger (`gdb` for `C++` and `pdb` for `Python`)

Numerical programming in python

- The speed of a well-coded `Python` program can be similar to that of a `C++` program, but it can be hundreds of times slower if you write the python program as though it was in `C++` without the curly braces.
- Try to avoid explicit python loops addressing array elements individually: `numpy` functions are optimised for vectorised problems: doing the same thing to a large number of elements in a single function call. Try where possible to cast your problem to make use of this property.
- Vectorised programs are typically also more compact and easy to see the intention of the code.

Vectorisation speed-up

```
def MultiplySlow(a,b):
    n=len(a)
    c=zeros(n)
    for i in range(n):
        c[i]=a[i]*b[i]
    return c

def MultiplyFast(a,b):
    return a*b

a=arange(10000)
b=a**2
```

```
%timeit MultiplySlow(a,b)
```

```
100 loops, best of 3: 5.88 ms per loop
```

```
%timeit MultiplyFast(a,b)
```

```
100000 loops, best of 3: 13.3 µs per loop
```