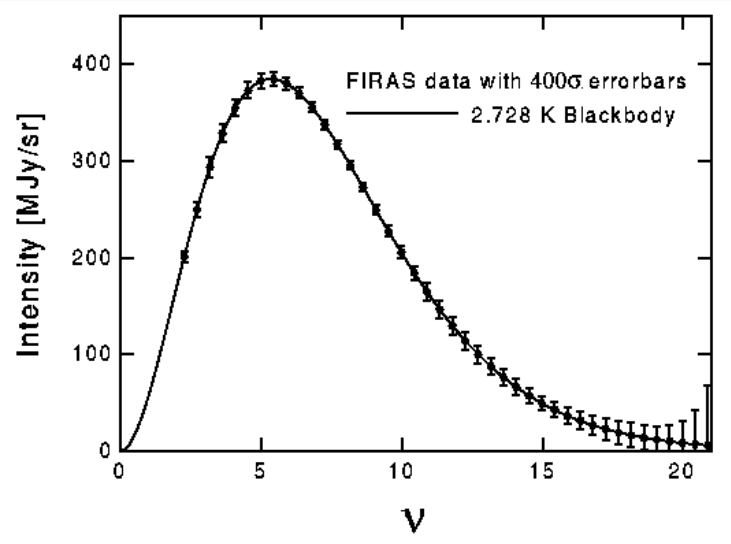
Fitting Data

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

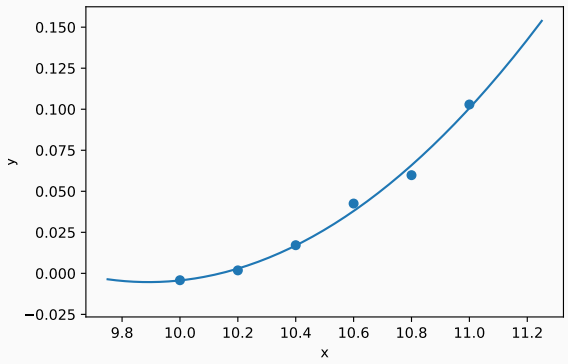
```
import numpy as np
import matplotlib.pyplot as plt
ndata=6
x=np.linspace(10,11,ndata)
y0=0.1*x**2-2*x+10  # noise-free data
y1=y0+np.random.normal(0,0.01,ndata) # noisy data
# Set up design matrix, assume unit errors
A=np.ones((ndata,3))
A[:,0]=x**2
A[:,1]=x
# Pseudo-inverse
Ainv=Np.linalg.pinv(A)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

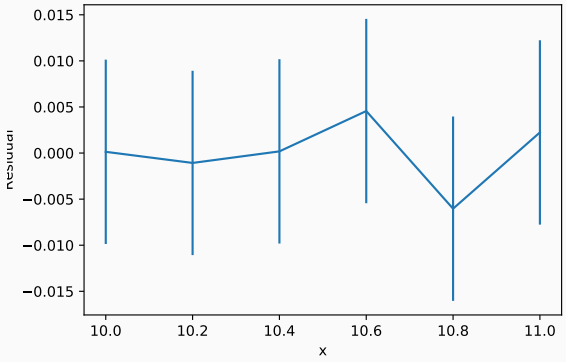
```
 [ 0.1 -2.  10. ] [ 0.086233   -1.70593319  8.43171695]
```

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")
```

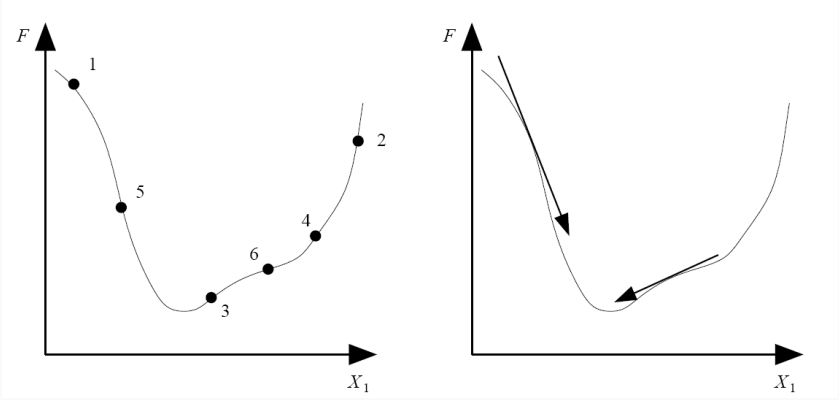




Non-linear problems: minimisation

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 $\partial f(\{\theta_i\})/\partial\theta_j$ (the **Jacobian** matrix). See Numerical Recipes for details.

For `Python`, there are a range of least-squares fitting functions and multidimensional minimisation algorithms in `scipy.optimize`.

For `C++`, see the `GSL` documentation under the "Nonlinear Least-Squares Fitting" and "Multidimensional minimisation" chapters. An example of using the `GSL` implementation of the Levenberg-Marquardt algorithm can be found in `fitpoly.cc` in the MCS examples.

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 $\sqrt{2\nu}$.

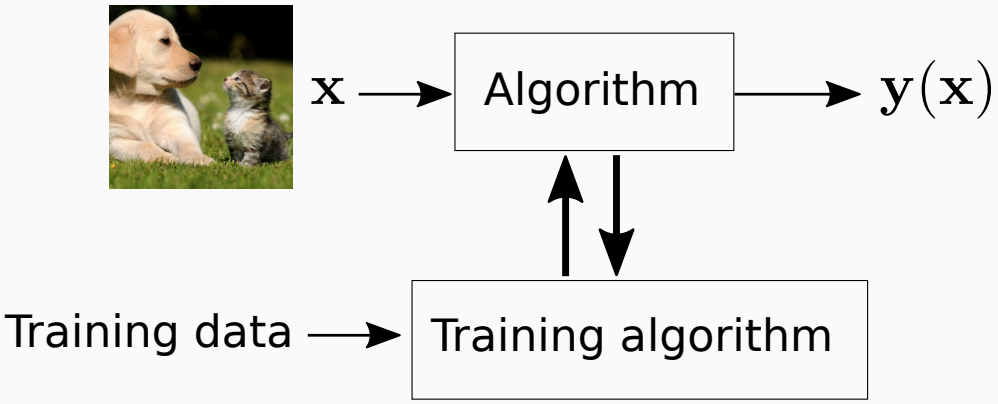
Rule of thumb: expect $\chi^2 \sim (N - M)$ for a good fit. If we don't know $\{\sigma_i\}$, we sometimes assume the best fit has $\chi^2 = (M - N)$ then estimate σ, assuming it is the same for all data points.

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.

Deep Learning

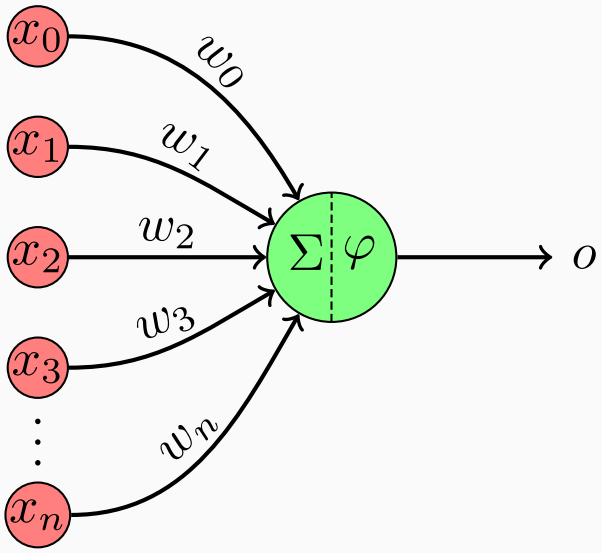
Machine learning: algorithm generation from data

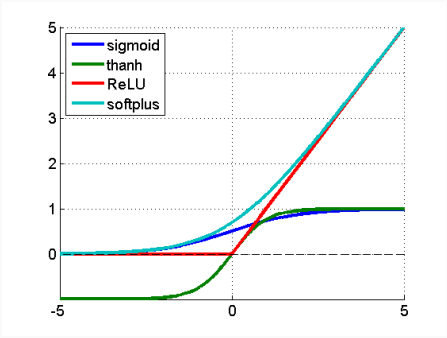
We seek an algorithm that maps an input **x** to a desired output **y**.



Note that **x** and **y** are typically vectors: for example **x** could be a set of pixel intensities in an image and **y** the probabilities of there being a cat and/or dog in the image.

A neural network is built out of artificial neurons

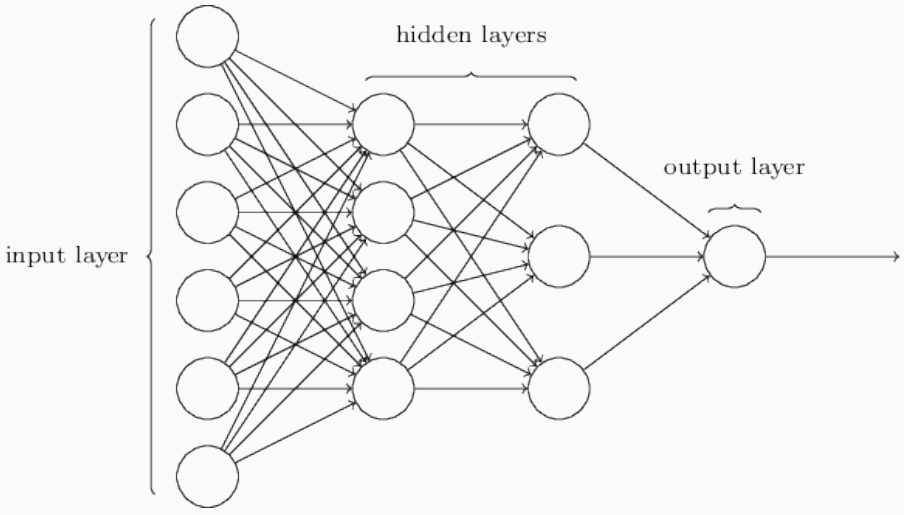




$$o(\mathbf{x}) = \phi\left(\sum_i w_i x_i\right),$$

where ϕ is a non-linear activation function.

A “deep” neural network



There are many possible network architectures adapted to different types of problems.

The universal approximation theorem states that a network with a single hidden layer can approximate any reasonably smooth function. In practice it needed the advent of fast GPUs to allow the training of systems with *several* hidden layers to solve real-world problems — deep learning.

Training

Training a neural net is the process of adjusting the set of weights $\{w_{ij}\}$ to give an output which approximates the desired output as well as possible for all the members of a (hopefully large) training set of data.

This involves applying the training data to the inputs and using gradient descent methods to minimise a cost/loss function based on the difference between the actual output and the desired output. The cost function is often quadratic, so very similar to a non-linear least-squares problem.

$$C(\{w_{ij}\}) \propto \sum_k |\mathbf{y}(\mathbf{x}_k) - \mathbf{t}(\mathbf{x}_k)|^2,$$

where $\mathbf{t}(\mathbf{x}_k)$ is the target output for a given member of the training set $\{\mathbf{x}_k\}$.

The backpropagation algorithm is often used to efficiently compute the derivatives $\partial C/\partial w_{ij}$ needed for gradient descent.

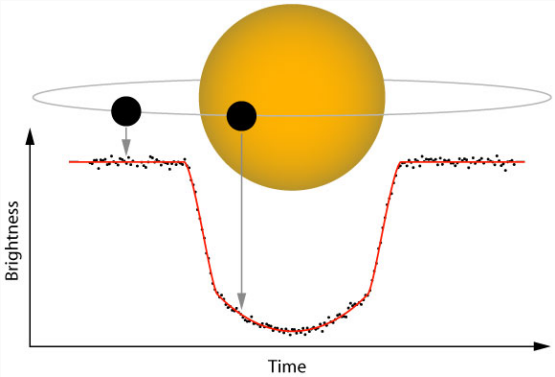
Packages

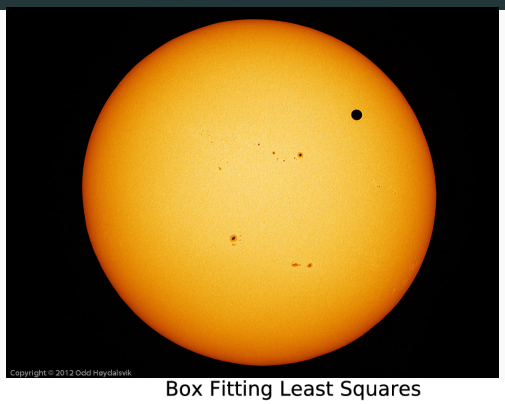
There are many packages to implement the training of deep neural networks, most are accessible from `Python`. These include:

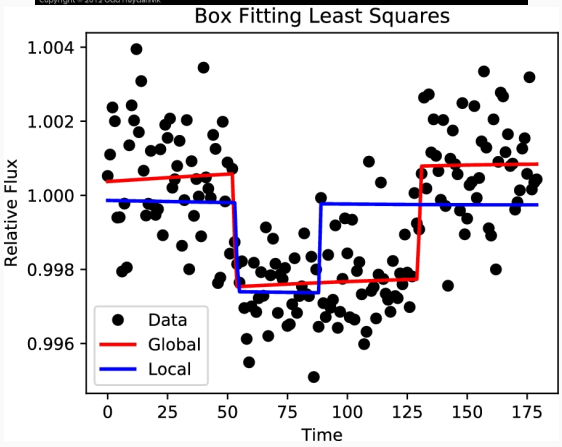
Theano	University of Montreal
TensorFlow	Google's development of Theano
Keras	Layer which can use Theano or TensorFlow
PyTorch	Developed by Facebook
MXNet	`C++` and `Python` bindings
Caffe	Berkeley Vision and Learning Center

These packages typically support the use of GPUs to allow the training of large neural nets.

Example application — detecting exoplanet transits

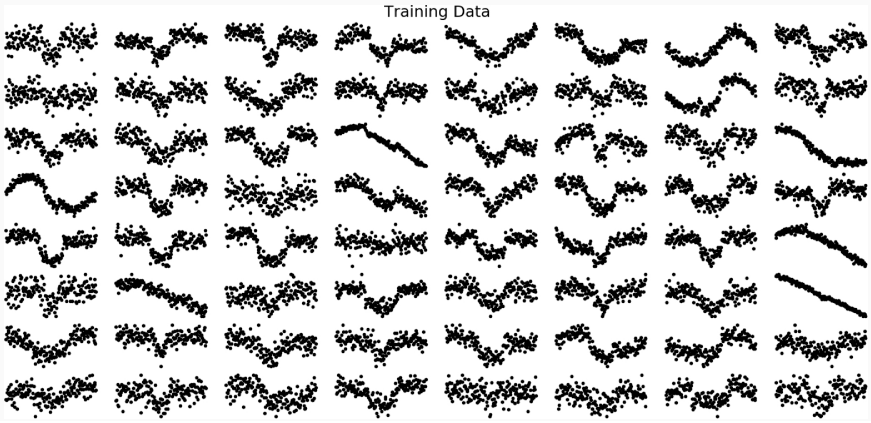


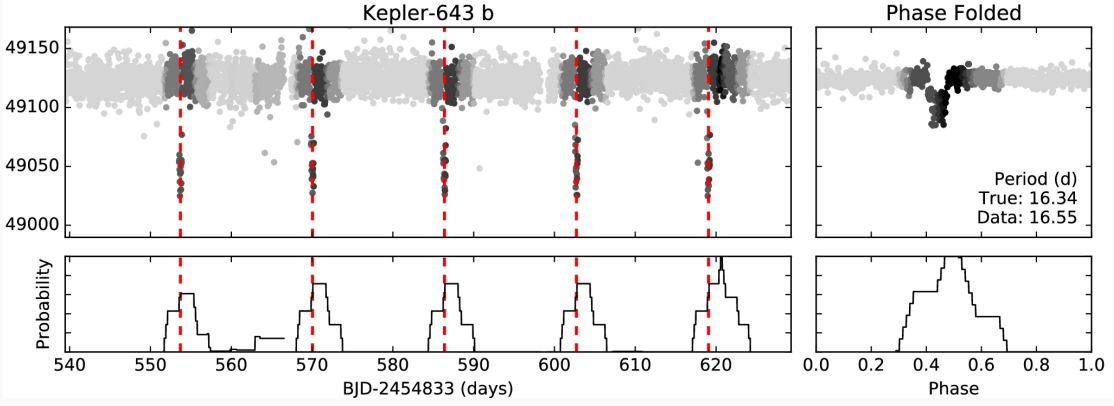




For the Earth transiting the Sun, the fractional "dip" is of the order 10^{-4}.

Training and test datasets





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(x)` or `calculateDerivative(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.

```
rad_to_degree = np.pi/180
x = y*rad_to_degree
```

  is much better than

```
x = y/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 and debugging

- 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 appropriate debugging techniques:
  - Print out intermediate results
  - Learn to use a debugger (`gdb` for `C++` and `pdb` for `Python`).
- An efficient algorithm for locating the bug once you have identified incorrect behaviour is divide and conquer — divide the program into “half” and check the correctness of the behaviour of each half individually. Subdivide the erroneous half (assuming there is only one) and so on until you find the bug.
- Beware the *Heisenbug* — the bug that disappears when you try to observe it.

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 (but not always — when in doubt, go for readability first, speed later).

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
```