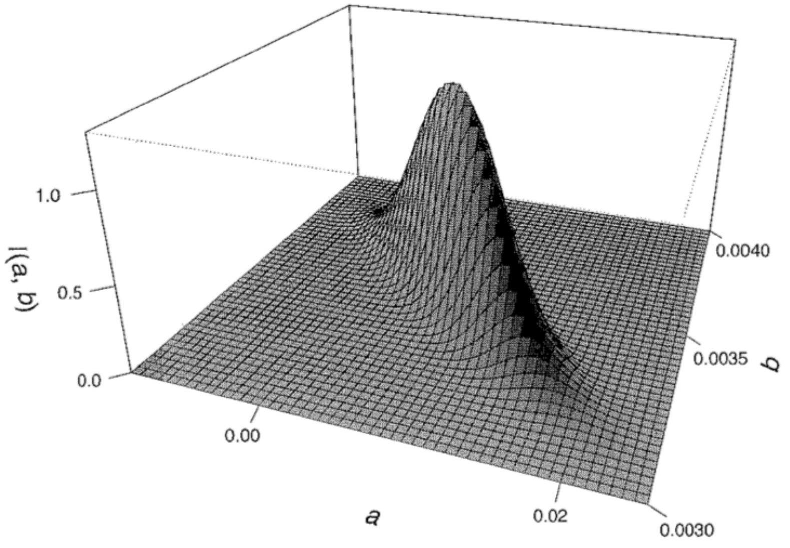
# STATISTICAL METHODS FOR THE PHYSICAL SCIENCES

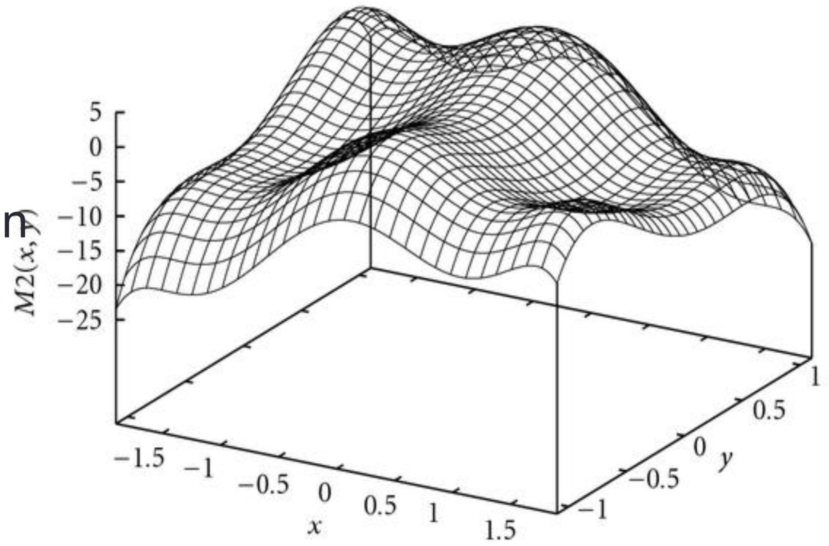
## Week 4 tutorial: Optimisation

For more detail (and the mathematics) see:
Numerical Recipes, Chapters 10 and 15 (pdfs available on the web)

# Optimisation

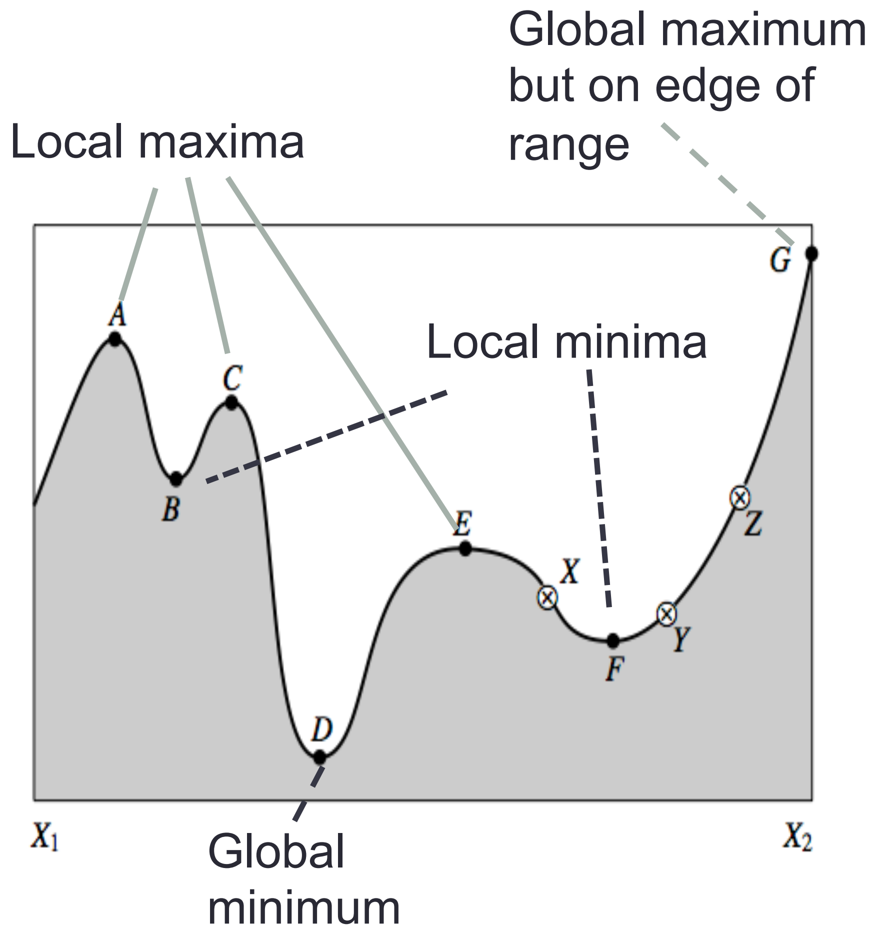
- In maximum likelihood estimation, we are trying to maximise a potentially complicated function: the (log)likelihood (or minimise the weighted least-squares equivalent for chi-squared fitting)
- How do we best find the correct ***extremum*** (minimum or maximum as appropriate)? We need to:
  - Obtain the ***global*** extremum, not a ***local*** one (don't get caught in a local 'peak' or 'valley'!)
  - Find the global extremum efficiently, with the minimum number of steps
  - Ideally, also obtain an estimate of errors on our MLEs
  - Do all this for what may be a hypersurface in a many-dimensional parameter space!
- Solving this problem (which has uses well beyond ML estimation) is a major field of numerical computing research called ***optimisation***





# The basic problem

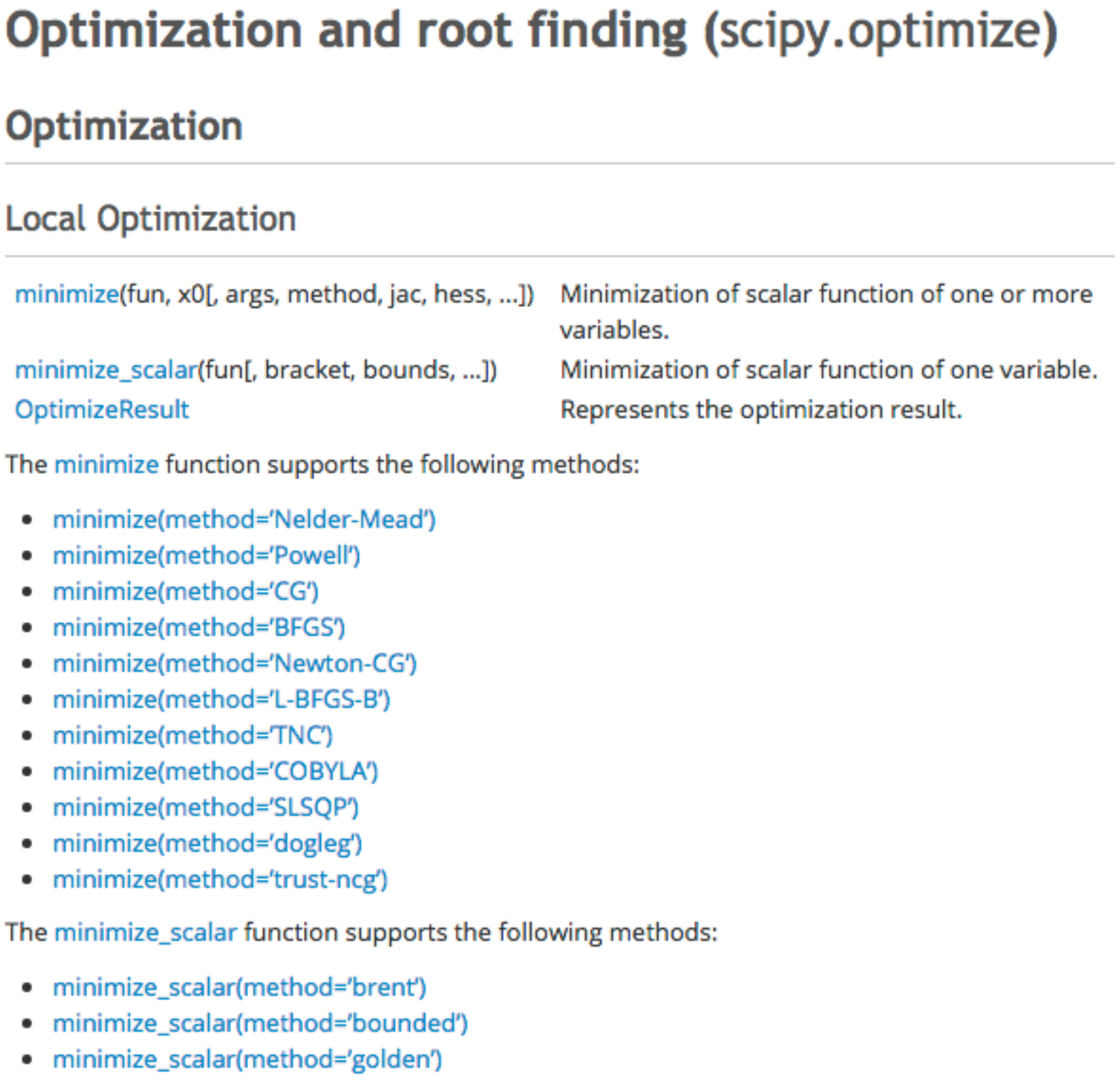
- Some methods may evaluate only the function to be optimised. Where in X to start?
- Some will evaluate also the derivative of that function.
- Evaluating derivatives alone to find the min/max can cause problems, e.g. what to do about point G? Are we looking at a minimum or maximum
- But derivatives can point in the right direction: which direction to go down/uphill? Not always reliable (e.g. point E)
- And 2nd order derivatives (the Hessian) can provide error estimates
- And how to avoid getting stuck in a local min/max…?
- Ultimately, all methods are *algorithmic*, combining different strategies to do the job. ***There is no perfect optimiser*** – which is best will depend on the problem you want to solve, and how quickly you want to solve it.



(figure from Num. Rec.)

# Optimisation in Python

Many different algorithms in scipy, here we will focus on a few of the main methods



## Optimization and root finding (scipy.optimize)

### Optimization

#### Local Optimization

| | |
|---|---|
| minimize(fun, x0[, args, method, jac, hess, ...]) | Minimization of scalar function of one or more variables. |
| minimize_scalar(fun[, bracket, bounds, ...]) | Minimization of scalar function of one variable. |
| OptimizeResult | Represents the optimization result. |

The minimize function supports the following methods:

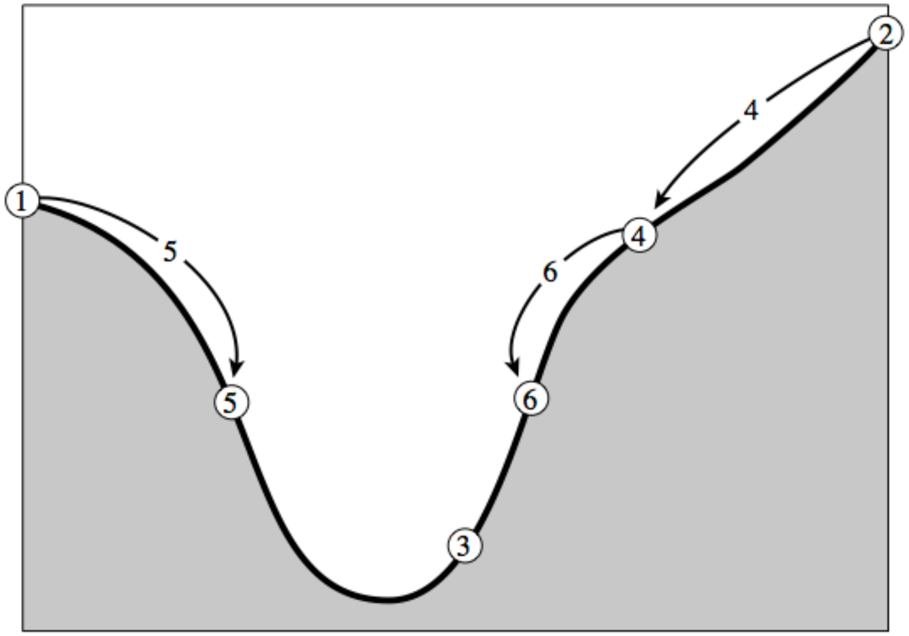
- minimize(method='Nelder-Mead')
- minimize(method='Powell')
- minimize(method='CG')
- minimize(method='BFGS')
- minimize(method='Newton-CG')
- minimize(method='L-BFGS-B')
- minimize(method='TNC')
- minimize(method='COBYLA')
- minimize(method='SLSQP')
- minimize(method='dogleg')
- minimize(method='trust-ncg')

The minimize_scalar function supports the following methods:

- minimize_scalar(method='brent')
- minimize_scalar(method='bounded')
- minimize_scalar(method='golden')

# 1D methods: bracketing and Golden Section search

- Find a minimum by ***bracketing***
- Evaluate function at points 1, 2 and 3
- Function *F* has minimum between 1 and 2 if *F(3)<F(1)* and *F(3)<F(2)*
- Now move 2 to 4. 3 is still lower than 4 and 1, so move 1 to 5. 3 is still lower, move 4 to 6…
- The rule is to keep the central point lower than the others.
- But when to stop…?



Golden Section search: due to scale similarity, the optimal bracketing interval keeps the central point a distance of 0.38197 and 0.61803 of the distance from the bracketing points. The ratio is the famous Golden Section!

# When to stop optimising: tolerance

Let's Taylor expand the function around the minimum, *b*:

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x-b)^2$$

There is no point in evaluating *f*(*x*) any further when the second term is smaller than the first by a factor equal to the numerical precision of your computer, $\epsilon$

Thus:

$$|x-b| < \sqrt{\epsilon}|b|\sqrt{\frac{2|f(b)|}{b^2 f''(b)}}$$

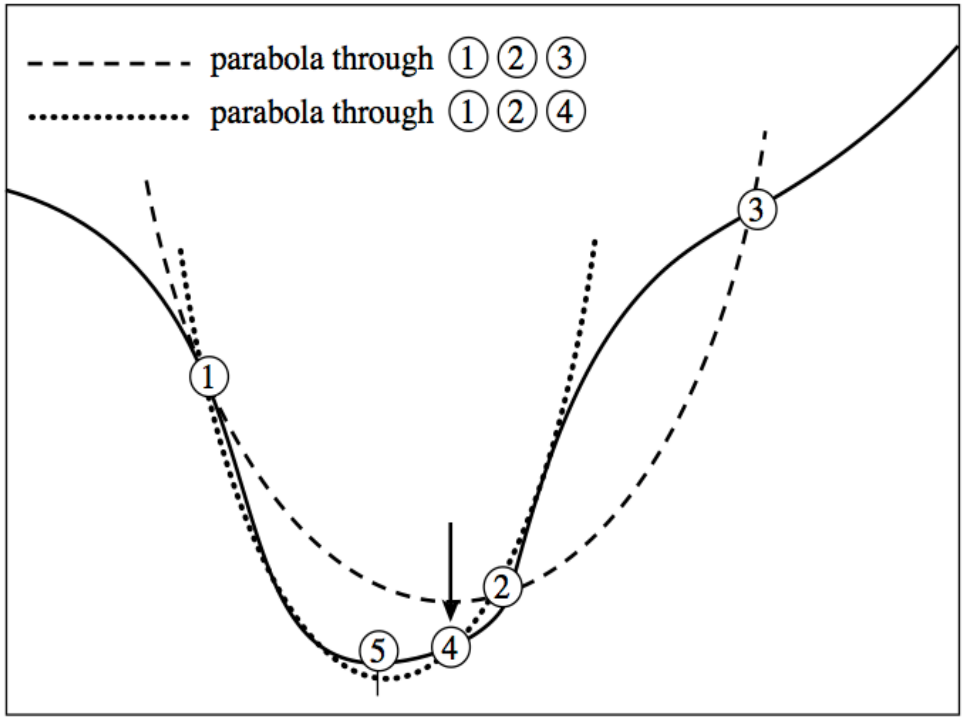
this sqrt term is of order unity for most functions

Typically $\epsilon \sim 10^{-16}$ for floating point 'double precision' on most current machines.

Thus the best possible optimisation precision or ***tolerance*** is ~1 part in $10^{-8}$
However, worse tolerances (larger values of fractional precision) can also be set, which can save computation time depending on the method used and what your requirements are.

# 1D methods: parabolic interpolation: Brent's method

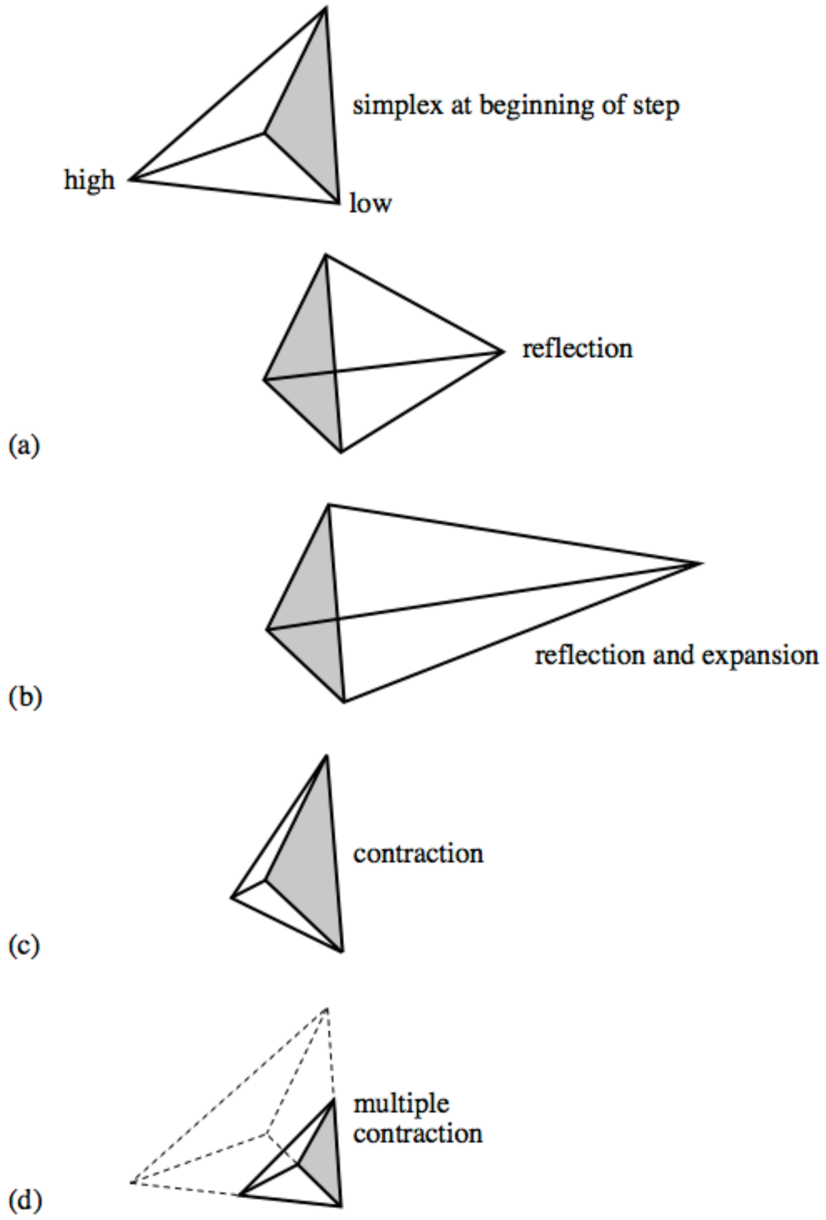
- Assumes function to be minimised is smooth and thus parabolic near the minimum
- Find parabola through points 1, 2 and 3.
- Evaluate function at parabola minimum, to get 4.
- Find parabola between 1, 2 and 4.
- Evaluate at minimum to get 5, closer to true minimum, and continue…



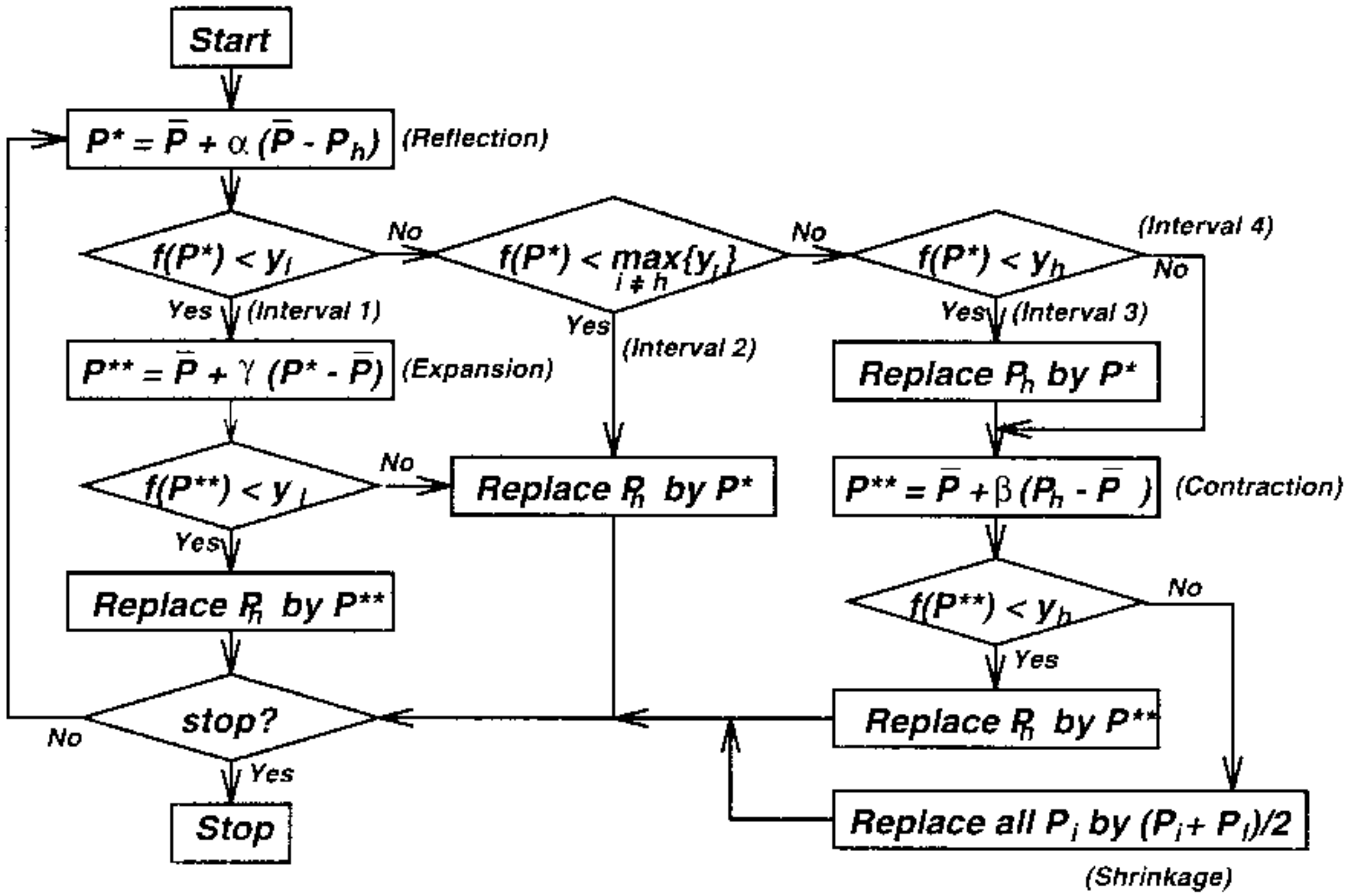
Brent's method keeps track of several steps at once to ensure ***convergence***, i.e. that the parabolas aren't causing the fit to jump wildly. To be accepted steps must show > factor 2 smaller changes in the x position of the minimum, than for the step before last

# nD methods: Downhill simplex (Nelder-Mead)

- *n*-dimensional approach to bracketing
- A ***simplex*** is an object described by a set of *n*+1 points or vertices in an *n* dimensional space. Each point corresponds to a set of parameter values for the function to be minimised.
- The ***downhill simplex*** or ***Nelder-Mead method*** moves the simplex across the (hyper)surface of the function using a variety of different steps through the parameter space.
- ***Reflections*** move the highest point through the opposite face of the simplex to a lower point on the (hyper)surface.
- ***Expansions*** allow the simplex to quickly cross to even lower parts of the surface when the path is straightforward.
- ***Contractions*** allow the simplex to 'ooze' through 'bottlenecks' in the landscape.

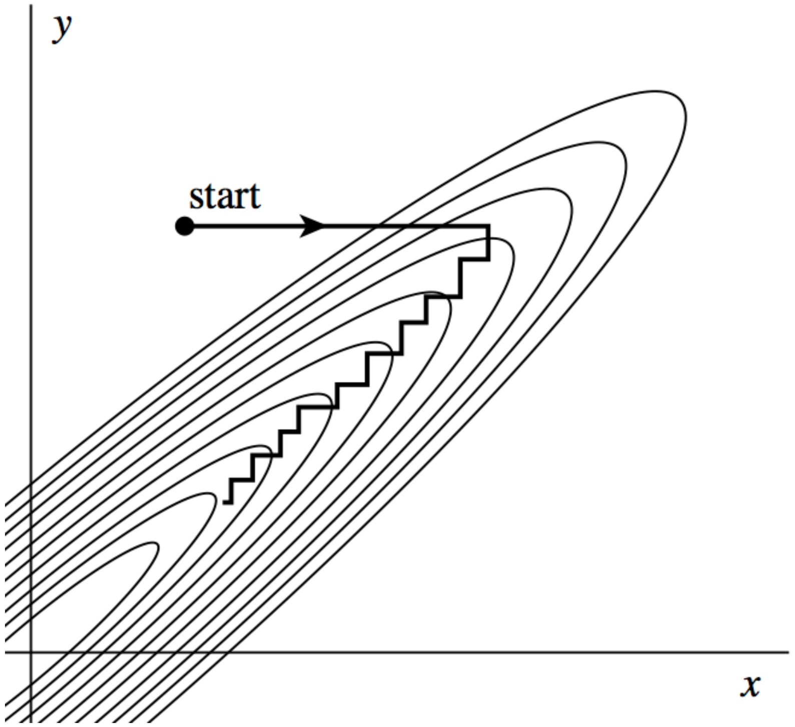


# Downhill simplex example flowchart



# nD methods: Direction set (Powell's) methods

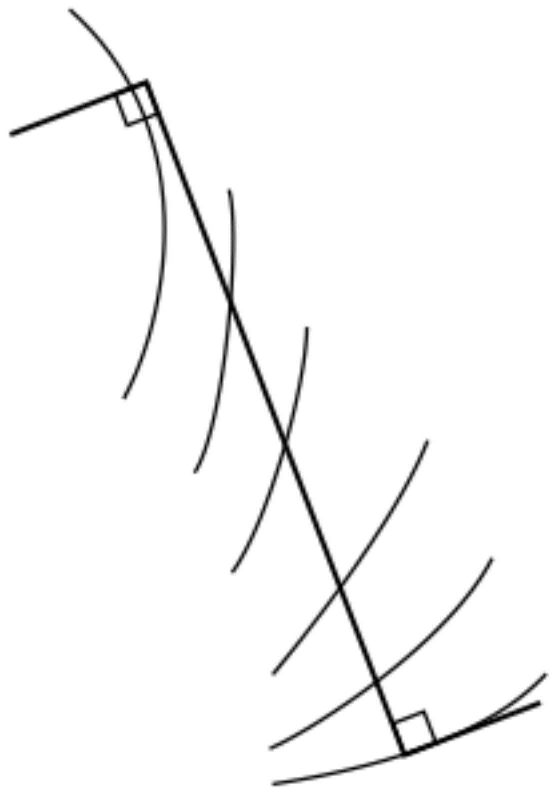
- Given the input vectors ***P*** and ***n***, and the function *f*, find the scalar λ that minimises *f*(***P***+λ***n***)
- Replace ***P*** with ***P***+λ***n*** and ***n*** with λ***n*** and continue…
- Can start with basis vectors for ***n***, but orthogonal vectors can be very inefficient, e.g. for surfaces which follow a long narrow 'valley' (right).
- Problem is that subsequent minimisations 'interfere' with each other. Need better set of ***conjugate directions***.



Powell's method uses a simple algorithm to obtain mutually conjugate direction vectors (see Num. Rec. Section 10.5), more advanced methods achieve this by calculating gradients of the function *f*

# Conjugate gradients and the Hessian

- Problem is that gradients continuously change (right) – to get downhill quickly it is more efficient to follow the steepest gradient locally than follow the direction of steepest gradient at the starting point in a straight line until a minimum is reached.
- To quickly find conjugate gradients one can use the Hessian (second partial derivative) of the function being minimised.
- Algorithmic approaches can also be applied (no Hessian required).



## nD methods general considerations

$$f(\mathbf{x}) = f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j$$

$$\approx c - \mathbf{b} \cdot \mathbf{x} + \frac{1}{2} \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x}$$

Hessian at **P**

Jacobian at **P**

where: $c \equiv f(\mathbf{P})$ $\quad \mathbf{b} \equiv -\nabla f|_{\mathbf{P}}$ $\quad [\mathbf{A}]_{ij} \equiv \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\mathbf{P}}$

- CG methods and ***variable metric methods*** all make various use of the above expansion of the function to be minimised, to obtain the best vector directions to efficiently navigate to the minimum.
- Commonly used variable metric methods: ***Broyden-Fletcher-Goldfarb-Shanno (BFGS)*** and ***Fletcher-Powell***
- For weighted least squares ('chi-squared') fitting, the **Levenberg-Marquardt method** (the curve_fit function in scipy) uses similar principles.

## Points to consider

- 1D or nD problem? 1D methods very robust, as is downhill simplex, but do not return other useful things like the Hessian.
- Speed to convergence? Some methods converge linearly, others quadratically (but need more memory, usually CG/variable metric methods – see Num. Rec. for details)
- nD but no Hessian needed, just minimum? Downhill simplex (called Nelder-Mead in scipy)
- Some methods (e.g. traditional Levenberg-Marquardt) will require at least the Jacobian to be already calculated – these are very fast and robust, but obtaining the Jacobian may not be easy.
- Other methods will estimate the Jacobian and often the Hessian as they go (it is needed for the algorithm), e.g. curve_fit in scipy, BFGS
- If you get stuck in a local minimum, try restarting from there (assumes method does not start with params it left off with, i.e. they are based on past history of iterations not uniquely on location in the fit).
- Be careful how you set up your problem, some methods can have convergence issues due to peculiarities in the data/function – sometimes just reformulating the function/parameters will help, otherwise you will need to try a different method.
- Fallback option: Monte Carlo methods