Thus far we have emphasized:

1. Writing the likelihood equation. $L(\theta) = \mathbb{P}(X|\theta)$.
2. Finding the parameter values that maximize the likelihood. Usually by:
   (a) Converting to the log-likelihood equation: $\ln L(\theta)$ rather than $L(\theta)$.
   (b) Taking the derivative of $\ln L(\theta)$ with respect to every parameter (every element of the parameter vector $\theta$.
   (c) Setting each derivative of the log-likelihood to be equal to zero.
   (d) Solving this system equations for the set parameter values, $\hat{\theta}$. These parameter values will be the MLEs[1]
   (e) Calculating the $\ln L(\hat{\theta})$
3. If we want to do hypothesis testing, then we:
   (a) Repeating the steps in bullet point #2, while enforcing any constraints made by the null hypothesis (resulting in $\hat{\theta}_0$ rather than $\hat{\theta}$).
   (b) Determine a likelihood ratio test statistic, $\Lambda$, by comparing the global ML solution to the ML solution under the null: $\Lambda = 2\left[\ln L(\hat{\theta}_0) - \ln L(\hat{\theta})\right]$
   (c) Evaluating the significance of $\Lambda$ by either:
      i. assuming a $\chi^2$ distribution with the appropriate degrees of freedom, or
      ii. using a computer to simulate lots of realizations of the data sets under the null hypothesis, and estimating the $\Lambda$ for each one of the realizations.

Step #1 above is unavoidable in likelihood-based inference, we have to be able to articulate the probability of the data given a model and set of parameter values. Sometimes we can't write the equation out in any compact form, we can just articulate its fundamental structure in a fairly abstract way (e.g. using summations and functions that we define for common combinations of parameters).

Step #2b can be tricky. Even if we can accomplish that, we may not be able to do #2d. Sometimes both of these steps are technically possible, but inordinately difficult.

Fortunately we can use numerical optimization techniques to try to find a set of parameter values that approximates $\hat{\theta}$.

Optimization is a entire field that is distinct from likelihood - indeed it is a major branch of applied mathematics. Traditionally optimization routines are discussed in the context of minimizing a function. We will simply minimize the negative-log-likelihood, to find the parameter values that maximize the likelihood.

Numerical optimization refers to the use of techniques that work on the numerical values of the problem rather than relying on solving the system with all of the variable treated analytically. In general the methods that we will talk about:

1. start from an initial guess,
2. investigate some points nearby in parameter space,

[1] We have to check the second derivatives to make sure that they are maxima not minima, and we may have to evaluate the endpoints of the range of parameters values, too.

3. repeat step #2 until the score (the log-likelihood in our case) stops changing much.

That description is too vague to be very helpful, but even so we can see that the methods:

1. can be starting-point dependent.
2. require a termination condition. Note that computers cannot store real numbers to arbitrary precision, so rounding error is an issue that we will have to confront. Most numerical methods have a "tolerance" (often referred to as 'tol' in software) that represents the smallest value that is considered notable when we consider changes in the objective function. Improvements in the log-likelihood of less than 'tol' will not be enough to keep the algorithms repeating their computation. tol will often be set to around $10^{-8}$ to avoid slow optimization caused by rounding error.

Designers of numerical optimization routines want to find $\theta_0$ such that $f(\theta_0) < f(\theta)$ for all values of $\theta$ that are relevant. In particular, they try to design algorithms that do this that:

1. Evaluate the function $f$ as few times as possible (because the function can be computationally expensive);
2. Work under a wide variety of circumstances (e.g. not just on well-behaved normal-like surfaces);
3. Require the user to supply as little information about the function as possible.

These criteria are in conflict. In general an algorithm that requires second derivatives, $f''$ will be faster than one that requires only first derivatives, $f'$. And functions that don't use information from derivatives at all will be the slowest variety. But the need to supply first and second derivatives constrains the type of problems that we can use with an algorithm.

## Single-variable optimization

These techniques are used when:

1. there is only one parameter to optimize, or
2. You have a multiparameter problem, but you already have a current point in parameter space, $\theta_i$, and you have already picked a direction to move in, $\Delta\theta$. You would like to try points:

$$\theta_{i+1} = \theta_i + \alpha\Delta\theta$$

In this case we want to find the optimal value of $\alpha$ (or the "step-size"), so we can view this a single-parameter optimization of $\alpha$.

### Parabolic interpolation

Imagine we have three points, $(x_1, y_1), (x_2, y_2)$ and $(x_3, y_3)$. Furthermore imagine that these points do not form a straight line.

In many problems, the shape near the minimum resembles a parabola (think of the log-likelihood of the normal as a function of $\mu$).

Recall that the formula for a parabola is:

$$ax^2 + bx + c = y.$$

Thus, we have

$$\begin{aligned} ax_1^2 + bx_1 + c &= y_1 \\ ax_2^2 + bx_2 + c &= y_2 \\ ax_3^2 + bx_3 + c &= y_3 \end{aligned}$$

and these three equations let us solve for $a, b$ and $c$.

$$\begin{aligned} a &= \frac{x_2y_1 - x_3y_1 - x_1y_2 + x_3y_2 + x_1y_3 - x_2y_3}{(x_2 - x_1)(x_2 - x_3)(x_3 - x_1)} \\ b &= \frac{x_2^2y_1 - x_3^2y_1 - x_1^2y_2 + x_3^2y_2 + x_1^2y_3 - x_2^2y_3}{(x_2 - x_1)(x_1 - x_3)(x_2 - x_3)} \\ c &= \frac{-x_2^2x_3y_1 + x_2x_3^2y_1 + x_1^2x_3y_2 - x_1x_3^2y_2 - x_1^2x_2y_3 + x_1x_2^2y_3}{(x_3 - x_2)(x_1^2 - x_1x_2 - x_1x_3 + x_2x_3)} \end{aligned}$$

The maximum/minimum of a parabola occurs when $x = \frac{b}{2a}$, so the extreme point should be at:

$$x_4 = \frac{x_3^2(-y_1 + y_2) + x_2^2(y_1 - y_3) + x_1^2(-y_2 + y_3)}{2(x_3(y_1 - y_2) + x_1(y_2 - y_3) + x_2(-y_1 + y_3))}$$

In this way we can predict a new value of $x$ form the preceding 3. Then we can use the $x_2, x_3$ and $x_4$ to construct a new triple of points so that we can continue another round (as long $f(x_4)$ more than 'tol' better than any of the other $f(x)$ values)

Demo of successive parabolic interpolation

**Golden section search**

If $x_1$ and $x_2$ bracket the minima then you can put $x_3$ in 38.2% of the way in between them (0.382, and 0.618 are the distances to $x_1$ and $x_2$, if we call the distance from $x_1$ to $x_2$ "one unit"). We'll assume that $y_3 < y_2$ and $y_3 < y_1$, so that $x_3$ is the reason that we known the interval brackets the minimum point.

We can continue to subdivide the larger interval remaining (and dropping more extreme points to narrow our bracketing) until we terminate.

This golden section search is more robust than the parabolic interpolation, but typically scores more points (unless the function is not close to quadratic).

**Brent's method**

Requires a bracketing, but uses parabolic interpolation when it seems to be working well (and falls back on Golden section search when it does not seem to be working well).

Brent's method is the basis of `optimize` in R.

In Python with SciPy, we use `from scipy import optimize;` at the top of the script. See `optimize.brent` for Brent's method, and `optimize.golden` for the golden section search. `optimize.bracket` can be used to find an interval that brackets a local optimum.

## 0.1 Nelder-Mead

See the nice discussion on Wikipedia's Nelder-Mead page

If you have $k$-dimensions, try out $k+1$ points and sort them by score: $x_1, x_2, \ldots x_{k+1}$.

Since $x_{k+1}$ is our worst point, we'll try to replace it.

Let $x_0$ be the mean of the points $x_1$ through $x_k$.

Let $\Delta x_{k+1}$ be the displacement from $x_{k+1}$ to $x_0$.

**Algorithm 1** Nelder-Mead Simplex method

$x_r$ is a point that is the mirror image of $x_{k+1}$ "reflected" around $x_0$ (so $x_r = x_0 + \Delta x_{k+1}$)
**if** $f(x_r) < f(x_1)$ **then**
  $x_e$ is another $\Delta x_{k+1}$ beyond around $x_r$ (so $x_r = x_0 + 2\Delta x_{k+1}$)
  **if** $f(x_e) < f(x_)$ **then**
    $x_{k+1} = x_e$
  **else**
    $x_{k+1} = x_r$
  **end if**
**else if** $f(x_r) < f(x_k)$ **then**
  $x_{k+1} = x_r$
**else**
  $x_c$ is halfway between $x_{k+1}$ and $x_0$ (so $x_r = x_{k+1} + 0.5\Delta x_{k+1}$)
  **if** $f(x_c) < f(x_{k+1})$ **then**
    $x_{k+1} = x_c$
  **else**
    Keep $x_0$ and halve the distance between all other points and $x_0$
  **end if**
**end if**

In R this is done with `optim(par, fn, method="Nelder-Mead"...)`.

In SciPy, we use we use `from scipy import optimize;` at the top of the script and then `optimize.fmin`

### Quasi-Newton methods

We can approximate the behavior of a function around a point by using a Taylor's series. Specifically, for approximating the value of a function at point $x$ given its value and derivatives at a nearby point, $x_0$:

$$g(x) \approx g(x_0) + \frac{1}{1!}\left(\frac{dg(x_0)}{dx}\right)(x - x_0) + \frac{1}{2!}\left(\frac{d^2g(x_0)}{dx^2}\right)(x - x_0)^2 + \ldots + \frac{1}{n!}\left(\frac{d^ng(x_0)}{dx^n}\right)(x - x_0)^n$$

If we just use the first two derivatives in our approximation then we get a parabolic function:

$$g(x) \approx g(x_0) + \left(\frac{dg(x_0)}{dx}\right)(x - x_0) + \frac{1}{2}\left(\frac{d^2g(x_0)}{dx^2}\right)(x - x_0)^2$$

Recall that the general formula for a parabola is

$$g(x) = ax^2 + bx + c$$

and, for a parabola

$$\frac{dg(x)}{dx} = 2ax + b$$

with a maximum/minimum at:

$$x = \frac{b}{2a}$$

For the case of the Taylor series approximation of our function we see that, our independent variable is the change in $x$ (which is $x - x_0$), and

$$\begin{aligned} a &= \frac{1}{2}\left(\frac{d^2g(x_0)}{dx^2}\right) \\ b &= \frac{dg(x_0)}{dx} \\ c &= g(x_0) \end{aligned}$$

So if we calculate the first and second derivatives our our function at step $k$ then we can predict a good value of $x$ for the next step by approximating the function using a parabola. In particular, $x_{k+1}$ should be at the minimum point of the parabola

$$\begin{aligned} x_k - x_{k+1} &= \frac{\frac{dg(x_0)}{dx}}{\frac{d^2g(x_0)}{dx^2}} \\ x_{k+1} &= x_k - \frac{\frac{dg(x_0)}{dx}}{\frac{d^2g(x_0)}{dx^2}} \end{aligned}$$

$$g(x) \approx g(x_0) + \left(\frac{dg(x_0)}{dx}\right)(x - x_0) + \frac{1}{2}\left(\frac{d^2g(x_0)}{dx^2}\right)(x - x_0)^2$$

For multiparameter problems this can be generalized as the the gradient of the function, $\nabla g$, rather then $\frac{dg(x)}{dx}$; and the Hessian matrix of partial derivatives, $H(g)$ rather $\frac{d^2g(x_0)}{dx^2}$. For an $n$-dimensional vector of parameters $x_1$ up to $x_n$, we would have:

$$\nabla g = \left(\frac{\partial g(x)}{\partial x_1}, \frac{\partial g(x)}{\partial x_2}, \ldots, \frac{\partial g(x)}{\partial x_n}\right)$$

$$H(g) = \begin{bmatrix} \frac{\partial^2 g(x)}{\partial x_1^2} & \frac{\partial^2 g(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 g(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 g(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 g(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 g(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 g(x)}{\partial x_1 \partial x_n} & \frac{\partial^2 g(x)}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 g(x)}{\partial x_n^2} \end{bmatrix}$$

## BFGS

The BFGS algorithms is one of the most widely used and effective Quasi-Newton method. Rather than evaluating both the gradient and the Hessian, it accumulates information during the run (in particular the changes in the gradients during the progression of the algorithm) that allow the Hessian to be approximated from previously calculated information.

Because evaluation of the gradient and elements of the Hessian matrix are often as computationally-expensive as the calculation of the likelihood, this "re-use" of the information of previous calculations can be very advantageous.

"Numerical Optimization" by Jorge Nocedal and Stephen J. Wright has a much more complete description (and is available in pdf form from KU's library).

In R this is done with `optim(par, fn, method="BFGS"...)`.

In SciPy, we use `from scipy import optimize;` at the top of the script and `optimize.fmin_bfgs`

In both cases, if you don't specify a function that can calculate the gradient, then it will be approximated using finite differences.

Recall that

$$\frac{\partial f(x)}{\partial x_1} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$$

the finite difference method approximates a derivative (or gradient) by making small changes to the variable (small $h$ values) and dividing the change in function value by $h$. Small, but finite, values of $h$ are considered, as long as they result in a detectable change in the function value. In other words we want $f(x + h) - f(x) > \text{tol}$.