Scientific Computing, Fall 2022
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2022/index.html

Scientific Computing

Jonathan Goodman, Fall, 2022

1 Optimization

The word *optimal* means the best possible. *Optimization* means finding the optimal. The mathematical optimization problem is formulated as finding the maximum or minimum of a function of one or more variables. The variables to optimize are $(x_1, \cdots, c_n)$, which may be thought of as parameters to be chosen. These form the components of $x \in \mathbb{R}^n$, which we sometimes call a "parameter combination". We write x as a column vector in matrix/vector formulas. The function being optimized is the *objective function*, or *loss* function, $f(x)$. The set of allowed values of x is the *feasible set*, which we call $\mathcal{F}$. The optimization problem is to find

$$f_* = \min_{x \in \mathcal{F}} f(x) \ . \tag{1}$$

We write x_* for an optimal x, which may or may not be unique. This is written

$$x_* = \arg \ \max_{x \in \mathcal{F}} f(x) \ . \tag{2}$$

Motives and goals

Optimization has the obvious use of finding the best possible parameter combination for some problem. Optimization methods also are used to "tune" parameters to make an algorithm work better, even with the understanding that the absolute best parameter combination cannot be found. Optimization is a systematic way to improve algorithms by "learning" good parameter settings. An engineer may be rewarded for making something work better, rather than being penalized for not making it perfect. The loss function, if it is differentiable (see below) is flat near x_*, so being somewhat close to the optimal x_* may put you very close to the optimal performance. We may care about f and f_* more than how close x may be to x_*.

The goal in other situation may be accurate estimation of x_*. For example, the optimization problem might be a *variational principle* that characterizes x_*. We saw variational principles for finding eigenvalues of symmetric matrices. Many differential equation systems used in mechanical engineering have useful variational principles (sometimes called *Dirichlet* principles). Examples include the partial differential equations that determine stresses in metal beams that hold up bridges and buildings.

Constraints, feasibility

An optimization problem is *unconstrained* if every parameter combination x is feasible, which is written $\mathcal{F} = \mathbb{R}^n$. Otherwise the problem is *constrained*. *Constraints* are conditions that define the feasible set. Most optimization problems have constraints. Different kinds of constraints lead to different kinds of optimization problems. For example, some of the variables/parameters might be constrained to be integers, as in deciding how many of which kinds of cars to buy. Inequality constraints might apply for variables that represent the amount of time spent on various tasks, which cannot be negative. Other constraints might involve combinations of parameters. Examples are budget constraints in finance, in which the parameters are amounts of money allocated to different things and there is a limit on the total spending.

An *equality* constraint is a mathematical relation that must hold among the parameters. For example, the total might be specified if the total spending must be equal to the amount budgeted. An *inequality* constraint is a mathematical relation that holds as an inequality. For example, if $x = x_1, x_2, x_3)$ is a point in three dimensional space, there might be a constraint on the size of x, as in $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + x_3^2} \leq R$. An inequality constraint is *relaxed* if the bound in the inequality is changed in a way that enlarges the feasible set $\mathbb{F}$. For example, if the size constraint is changed to $\|x\|_2 \leq R + 1$. An inequality constraint is *binding* at an optimum x_* if the constraint inequality is an equality at x_*. It may be useful to distinguish between *weakly binding* (the definition just given) and *strongly binding*, which would mean that the optimum (1) gets strictly smaller if the constraint is relaxed. For most problems, a binding constraint is strongly binding in this sense.

Some ideas from mathematical analysis or topology are useful for discussing constrained optimization. One is the distinction between a *minimum*, which must be *attained*, and an *infimum*, which only needs to be *approached*. The value f_* is *attained* if there is an x_* with $f(x_*) = f_*$. A value f_* is *approached* if there is a sequence x_n so that

$$\lim_{n\to\infty} f(x_n) = f_* \ .$$

If f_* is approached, and $f_* \leq f(x)$ for all $x \in \mathcal{F}$, then f_* is the *infimum* of f, which is written

$$f_* = \inf_{x\in\mathcal{F}} f(x) \ .$$

There are several ways for an infimum not to be a minimum. One is that $x_n \to \infty$, as would happen for

$$f(x) = \frac{1}{1+x^2} \ , \ \ \inf_{x\in\mathbb{R}} f(x) = 0 \ , \ \ f(x_n) \to 0 \text{ as } x_n \to \infty \ .$$

Another way is that the feasible set $\mathcal{F}$ could not contain its *limit points*. An example would be looking for the minimum of $f(x) = |x|$ over the set $\mathcal{F} = \{x \mid x > 0\}$. A *minimizing sequence* $x_n \to 0$ has $n \to \infty$ (such as $x_n = \frac{1}{n}$) but the point x_* with

$$f(x_*) = \inf_{x \in \mathcal{F}} |x| = \inf_{x > 0} |x| = 0$$

is not a feasible point. A third way is for the objective/loss function not to be continuous, as in

```
def f(x):
    """A discontinuous function"""
    if x > 0:
        return x
    else:
        return 1.
```

All of these things happen in real optimization projects.

Derivatives

Most optimization algorithms work by *searching* through $\mathcal{F}$ looking for parameter combinations that have smaller f. If ∇f (the gradient of f) is the vector of first partial derivatives, then $-\nabla f$ is a direction in which f decreases. Optimization algorithms would "like" to use derivatives of the objective/loss function and functions defining the constraints, either explicitly or implicitly.

The problem is that functions that make up the overall objective/loss function may not all be differentiable. A simple non-differentiable function is $r(t) = |t|$, which has

$$r'(t) = \text{sign}(t) = \begin{cases} +1 & \text{if } x > 0 \\ -1 & \text{if } x < 0\ . \end{cases}$$

This function is not differentiable at $t = 0$, because the left and right values $(= \pm 1)$ disagree. The definition of derivative is

$$r'(0) = \lim_{h \to 0} \frac{r(h) - r(0)}{h}\ .$$

The ratio on the right has no limit, because it is $+1$ if $h > 0$ and -1 if $h < 0$. This is not just a picky mathematical point. Many optimization methods fail for non-differentiable functions, even this $r(t)$.

It happens, more often than you might think at first, that the loss function or constraint functions are not differentiable at x_*. One example is l^1 regularization ("ell one"), which is closely related to *lasso* regression. Recall that Tikhonov regularization adds a regularizing term $\lambda \|x\|_2^2$ to a linear least squares loss function $\|Ax - b\|_2^2$, resulting in the unconstrained problem of minimizing

$$f_T(x) = \|Ax - b\|_2^2 + \lambda \|x\|_2^2\ .$$

Lasso regression[1] uses $1-$norm (also called the l^1 or L^1 norm) regularization

[1] A *lasso* is a simple rope trick cowboys use to catch cows around the neck with a rope. There is a tradition in statistics of naming techniques after simple cowboy things. Other examples (statistical techniques) are the "jackknife" (nothing to do with cutting) and "bootstrap" (nothing to do with walking). The name is supposed to imply that the technique is simple and easy to use.

instead of 2−norm Tikhonov regularization. The result is the loss function

$$f_L(x) = \|Ax-b\|_2^2 + \lambda \|x\|_1 = \|Ax-b\|_2^2 + \lambda \sum_{k=1}^{n} |x_k| \ .$$

There is (as we saw) an explicit formula for the minimizer of the Tikhonov regularized least squares problem, which uses the SVD of A. There is no comparable simple algorithm for the Lasso regression problem. An advantage of Lasso regression is that components x_k with little (but non-zero) influence on the quality of fit $\|Ax-b\|_2^2$ are set to zero exactly by the Lasso regularization term. The value $x_k = 0$ is where f_L is not differentiable. Gradient descent for Lasso regression is problematic (not impossible, but harder).

Landscape, local and global minimizers, convexity

The *landscape* for an optimization problem is the "shape" of the graph of $f(x)$ over $\mathcal{F}$. The word "landscape" refers to the shape of a region of land – is it flat or hilly, smooth or rocky, etc. For example, places where f is not differentiable may be visualized as "folds" or "corners" in the graph/landscape. You should be careful in thinking of "shape" because it implies 3D intuition where f is a function of just two parameters. These intuitions sometimes go wrong in "high dimensions" (large n, many parameters).

One feature of a function landscape, important for optimization, is a *local minimum.* We say x_* is a *strict* local minimum if any nearby feasible point has a strictly larger value. To say this in mathematical notation, there is an $r > 0$ so that

$$f(x) > f(x_*) \ , \ \text{if } x \in \mathcal{F} \ , \ \ \|x - x_*\| \leq r \text{ and } x \neq x_* \ .$$

We say x_* is a *non-strict* local minimum if there is no nearby feasible point with a lower f. A non-strict local minimum can arise by symmetry. For example, here is an objective function that tries to make a point in the plane a unit distance from the origin: $f(x_1, x_2) = \left(x_1^2 + x_2^2 - 1\right)^2$. Any (x_1, x_2) with $x_1^2 + x_2^2 = 1$ is a non-strict local minimizer.

A strict *global* minimizer is x_* with $f(x_*) < f(x)$ if $x \neq x_*$ and $x \in \mathcal{F}$. The point x_* is a non-strict global minimizer if $f(x) \geq f(x_*)$ for all $x \in \mathcal{F}$. The goal of optimization, usually, is to find global minimizers. Local minimizers that are not global "get in the way". The local minimum problem is one of the greatest unsolved, probably unsolvable, challenges in optimization.

Figure 1 has an example that is typical in having more than local minimum. It is not typical because the loss function has just one argument. As we already said, it is hard to visualize functions of many arguments. The function arises from fitting a frequency parameter ω to data. There are observations times t_k, which are uniformly spaced in some observation interval. The observed values (fake data) are $y_k = A_0 \sin(\omega_0 t_k) + \epsilon_j$. Here, A_0 and ω_0 are the true values (amplitude and angular frequency) of the signal and ϵ_j is independent mean zero variance σ^2 Gaussian noise. The fit is to be found by minimizing a sum of

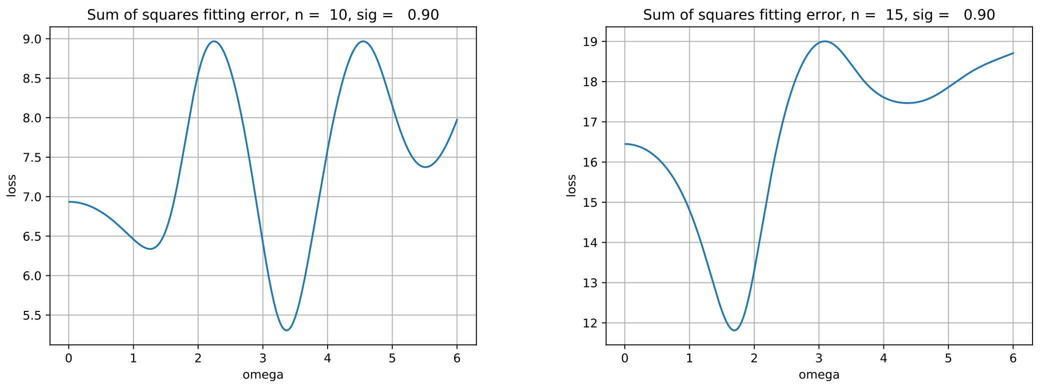

Figure 1: Loss function landscapes with local minima.

squares loss function

$$F(\omega, A) = \sum_{k=1}^{n} \left[y_k - A\sin(\omega t_k) \right]^2 \ .$$

Minimizing over A is a linear least squares problem whose solution is

$$A_*(\omega) = \frac{\sum_k y_k \sin(\omega t_k)}{\sum_k \sin(\omega t_k)^2} \ .$$

Figure 1 has two plots a plot of

$$f(\omega) = F(\omega, A_*(\omega)) \ .$$

(Different n and noise values) The true value is $\omega_0 = 1.5$. The loss function has a local minimum near this value in both runs. On the left, the global minimum is $\omega_* \approx 3.3$, with another local min at $\omega \approx 5.5$. On the right $\omega_* \approx 1.6$ is a *deep* global min with another local min at $\omega \approx 4.4$.

A function is *unimodal* (strictly or non-strictly) if any local minimizer (strict or non-strict) is a global minimizer (strict or non-strict). Convex functions (defined below) are unimodal. Unimodal functions are good for optimization because there are no local minima to get stuck in.

A set D is *convex* if you cannot leave D moving on a straight line between two points of D. More precisely, suppose $x \in D$ and $y \in D$. The line segment between x and y is the set of all *convex combinations* $z = \lambda x + (1-\lambda)y$ with $0 \leq \lambda \leq 1$. The set D is convex if all such points z are in D. The disk $\left\{(x_1, x_2) \mid x_1^2 + x_2^2 \leq 1\right\}$ is convex. Another convex set is the *unit simplex* $S \subset \mathbb{R}^3$ defined by

$$S = \{(x_1, x_2, x_3) \mid x_1 \geq 0, \ x_2 \geq 0, \ x_3 \geq 0, \ x_1 + x_2 + x_3 \leq 1\}$$

This simplex is a two dimensional triangle sitting in a three dimensional space. The simplex has corners on the coordinate axes: $(1,0,0)$, $(0,1,0)$, $(0,0,1)$. The *unit ball* in a vector space for a vector norm $\|\cdot\|$ is

$$B = \{x \mid \|x\| \leq 1\} \ .$$

The unit ball for the $2-$norm is a round ball. The unit "balls" for the $1-$norm or the max norm in 2D are both squares. The triangle inequality for vector norms is the same as requiring the unit ball to be convex.

An objective/loss function f is *convex* if the feasible set $\mathcal{F}$ is a convex set, and if the line segment connecting two points on the graph of f lies above the graph. That means that if $x \in \mathcal{F}$ and $y \in \mathcal{F}$ and if $z = \lambda x + (1-\lambda)y$ with $0 \leq \lambda \leq 1$, then $f(z) \geq \lambda f(x) + (1-\lambda) f(y)$. For example, the function $f(x) = x^2$ is convex, while $f(x) = \sin(x)$ is not. A function is *concave* if its negative is convex.[2] A function is *strictly* convex if the line segment is strictly above the graph, which is $f(z) > \lambda f(x) + (1-\lambda) f(y)$ when $0 < \lambda < 1$. The function $f(x) = x^2$ is strictly convex, while $f(x) = |x|$ is non-strictly convex. A local minimizer of a convex function is a global minimizer. A local minimizer of a strictly convex function is a strict global minimizer.

A function of one variable is convex if its second derivative (exists and) is positive. If $f''(x)$ is defined for all relevant $x \in \mathbb{R}$, then $f''(x) \geq 0$ for all x is equivalent to f being non-strictly convex. The strict inequality $f''(x) > 0$ implies that f is strictly convex. For functions of $n > 1$ variables, the second derivative function is replaced by the $n \times n$ symmetric *hessian* matrix of second partial derivatives

$$H_{jk}(x) = \frac{\partial^2 f}{\partial x_j \partial x_k} \ .$$

The hessian matrix is symmetric because partial derivatives "commute" in the sense that

$$H_{jk}(x) = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_k}\right) = \frac{\partial}{\partial x_k}\left(\frac{\partial f}{\partial x_j}\right) = H_{kj}(x) \ .$$

The function f is strictly convex if the hessian is positive definite. The function is non-strictly convex if the hessian is positive semi-definite.

Optimization software

Most optimization algorithms *search* the feasible region, looking for parameter combinations that lower the value of the objective/loss function. The algorithm "learns" about the loss/objective function by "queries". A query is a request to learn the value $f(x)$ or the gradient $\nabla f(x)$, or higher derivatives. It may learn about the feasible set $\mathcal{F}$ using queries of the functions defining $\mathcal{F}$. Some problems have a feasible set defined implicitly rather than by explicit functions. For example, the loss function may involve solving an auxiliary system of equations, which may or may not have a solution for a given x. In such cases, the

[2] The calculus book terms "concave up" and "concave down" are not used professionally.

code evaluating f may return the value `inf` (the IEEE floating point standard for "infinity") if x is not feasible. The "user" uses the optimization software by writing code to evaluate these. Typical optimization algorithms are simple enough that most of the work needed to optimize f is spent in the user code evaluating f and its derivatives. It is sensible, therefore, to measure the efficiency of an optimization algorithm by asking how much progress it makes per function/gradient evaluation.

There are exceptions to this. Optimization software for some special classes of problems, including *linear programming*, ask the user to provide the data defining f and $\mathcal{F}$. There are loss functions that are expensive to evaluate exactly but cheaper to evaluate approximately. One example is *stochastic gradient descent*, in which f is approximated using a small random sample of the large data set that defines f. There are many names for deterministic cheap approximations, including *prox functions*, *approximate response surfaces*, *model reduction*, *multi-fidelity models*, etc.

2 Gradient descent iteration, unconstrained

"Entry level" optimization software uses simple *iterative* algorithms. After n iterations, there is a *current iterate*, x_n. Iteration $n+1$ finds a new iterate by applying some operations involving x_n. Mathematically, this means that there is an iteration function $\Phi(x)$ so that $x_{n+1} = \Phi(x_n)$. A more sophisticated algorithm might use, for example, x_{n-1} in addition to x_n to get x_{n+1}.

Gradient descent with *learning rate* s, also called *step size*, is the simple iterative algorithm

$$\Phi(x) = x - s\nabla f(x) \ . \tag{3}$$

"Descent" means "going down" – reducing f in this context. *Gradient descent* means reducing f by going down in the (negative) gradient direction. The iterative algorithm corresponding to this is

$$x_{n+1} = x_n - s\nabla f(x_n) \ .$$

This may be described as moving a "distance" s in the *search direction* p, where the search direction is the negative gradient

$$p = -\nabla f \ . \tag{4}$$

This is written

$$\Phi(x) = x - sp(x) \ , \ \ p(x) = -\nabla f(x) \ . \tag{5}$$

This is "constant learning rate gradient descent". The step size s is not really the distance, which is the norm of the step. You get actual step size by multiplying the "step size" (learning rate) by the norm of the "search direction". That is, $\|x_{n+1} - x_n\| = |s| \, \|p(x_n)\|$. This probably also means that $p(x)$ is not a "direction", because a direction is a vector of length one, while the norm of p could be any non-negative number.

There is no guarantee that a gradient descent step (3) stays inside $\mathcal{F}$. It might be that $x \in \mathcal{F}$ but $\Phi(x) \notin \mathcal{F}$. This constraint violation is almost guaranteed if the constraints include one or more nonlinear equality constraints. But even if $\mathcal{F}$ is defined by simple inequalities such as positivity constraints, these can be violated by a step that is too large. *Unconstrained optimization* is supposed to mean optimization when every x is feasible, $\mathcal{F} = \mathbb{R}^n$. What *unconstrained* means in practice, more often, is that we are far enough from being bound by constraints that algorithms and analysis for problems without constraints make sense. This would be the case, for example, if x is close to x_*, and constraints are not binding at x_*.

Step size, line search, descent direction

You have to choose the learning rate/step size parameter s to do gradient descent (3). If s is too large the iterates x_n may diverge. If s is too small, the iterates will move to x_* too slowly. Step size control is an important part of any good gradient descent software.

Let x_n be the iterate and p_n the search direction. *Line search* means finding a good step size s_n. It is called "line" search because adjusting s searches only over the line $x_n + sp_n$ rather than the whole feasible set $\mathcal{F}$. One criterion for a good s is *decrease* of the objective function

$$f(x_{n+1}) < f(x_n) \ .$$

This may seem obvious, but the simple gradient descent algorithm requires ∇f not f itself. Values of f are used only in the line search phase of gradient descent.

Before implementing line search, you might ask whether there is any $s > 0$ that leads to decrease. More specifically, do you achieve decrease by making s small enough? If this is answered by simple calculus, with an affirmative answer, then p is a *descent direction.* The function value for small s has a first derivative approximation about $s = 0$ given by the chain rule:

$$\frac{d}{ds} f(x + sp)\Big|_{s=0} = \left(\nabla f(x)\right)^T p \ .$$

The search direction p is a descent direction if

$$\left(\nabla f(x)\right)^T p < 0 \ . \qquad (6)$$

The negative gradient is a search direction if x is not a stationary point. A *stationary point* is a point where $\nabla f(x) = 0$. If x is not a stationary point, then the gradient descent search direction makes the inner product in (6) equal to

$$-\left(\nabla f(x)\right)^T \nabla f(x) = -\left\|\nabla f(x)\right\|_2^2 < 0 \ .$$

The Taylor approximation is

$$f(x_{n+1}) = f(x_n) - s_n \left\|\nabla f(x_n)\right\|_2^2 + O(s_n^2) \ .$$

This shows that f decreases if s_n is small enough.

Practical optimization software often looks for a suitable s_n using *bisection search*, also called *binary search* or something like that. Suppose x is the current iterate, p the search direction, and $s > 0$ is an initial guess for the step size. Binary search means replacing s with $\frac{s}{2}$ if s is too large. Here is some "pseudo-code" describing the process:

```
while $f(x + sp) \geq f(x)$ :                (7)
    $s \leftarrow \frac{1}{2}s$
```

This will produce descent. The value of s will be different at each iteration.

Sufficient decrease, convergence to stationary points

The sequence of iterates *converges* if the limit x_* exists:

$$x_* = \lim_{n\to\infty} x_n$$

The sequence converges to a stationary point if $\nabla f(x_*) = 0$. If each step achieves descent, which is $f(x_{n+1}) < f(x_n$, then it almost always true that x_* is a local minimizer if x_* is a stationary point. Stationary points that are not local minimizers should be unstable for the iteration.

Unfortunately, decrease at every iteration, by itself, does not guarantee convergence. Convergence does not guarantee that x_* is a stationary point. An example of that is $f(x) = \frac{1}{2}x^2$ in one dimension, initial guess $x_0 = 1$ and step size $s_n = a^n$ with $0 < a < 1$. In this example, the gradient descent iteration becomes

$$x_{n+1} = x_n - a^n x_n = (1 - a^n)\, x_n \ .$$

The step size is converging to zero exponentially fast, which slows the iteration so much that it reaches a limit that is not 0. The limit is an infinite product

$$\begin{aligned}
x_1 &= (1-a)\, x_0 = 1 - a \\
x_2 &= \left(1 - a^2\right) x_1 = \left(1 - a^2\right)(1 - a) \\
&\vdots \\
x_n &= (1 - a^n)\left(1 - a^{n-1}\right)\cdots(1 - a) \\
x_* &= \prod_{n=1}^{\infty} (1 - a^n) > 0
\end{aligned}$$

You can see that $x_* > 0$ using the fact that $\log(x_*)$ is a convergent sum

$$\log(x_*) = \sum_{n=1}^{\infty} \log(1 - a^n) = L \ .$$

Therefore, $x_* = e^L > 0$ whatever L turns out to be. You can see that the sum converges using the first derivative approximation

$$\log(1 - a^n) \approx -a^n \; .$$

This shows that the sum defining L converges like a geometric series. The conclusion from this is that you want x_n to be small enough to insure decrease but not too much smaller than that.

Sufficient decrease conditions are conditions that guarantee that either $x_n \to \infty$ or x_n converges and x_* is a stationary point. The example above shows one way that violating sufficient decrease conditions can lead to convergence to a non-stationary point. Other violations can lead to non-convergence without $x_n \to \infty$. For example, you can see that $s_n = 2 - a^n$ does this with $f(x) = \frac{1}{2}x^2$. In this example, $s_n = 2$ makes $x_1 = -x_0$, then $x_2 = -x_1 = x_0$, etc., which you can call "bouncing". Making s_n slightly less than 2 has decrease of f but has bouncing that does not go to zero as $n \to \infty$.

The details of sufficient decrease conditions can be complicated, but one simple strategy involves the *Armijo* condition (approximate American/Spanish pronunciation: "are meehoe"), which involves a sufficient decrease parameter α with $0 \leq \alpha \leq 1$ ($\alpha = \frac{1}{2}$ and $\alpha = .1$ are reasonable values)

$$f(x + sp) \leq f(x) + \alpha s \, \nabla f(x) \, p \; . \tag{8}$$

The quantity $s\nabla f(x)p$ on the right is the *predicted decrease* of f, predicted using the first order Taylor approximation of f that was used to derive gradient descent. The Armijo condition is that the actual decrease of f should be within a factor of α of the predicted decrease. This prevents the "bouncing" behavior of the above example. A code based on the Armijo condition would replace (7) with (8).

Unfortunately, the Armijo condition still allows step sizes that decrease to zero too fast. One way to prevent that is the reverse Armijo condition

$$f(x + 2sp) \geq f(x) + 2\alpha s \, \nabla f(x) \, p \; . \tag{9}$$

The code can achieve this using a binary expansion search algorithm. If a step satisfies the Armijo condition (8) but not the reverse condition (9), then replace s with $2s$. This still satisfies the Armijo condition, which is the only way not to satisfy the reverse condition. If you like concrete mathematical analysis, you might want to construct a proof of the convergence theorem for this: Suppose the first and second derivatives of f are all bounded, and if s_n satisfies the Armijo and reverse Armijo conditions then either the limit x_* exists and is a stationary point, or $x_n \to \infty$.

Conditioning

Gradient descent converges slowly for problems that are poorly conditioned. We illustrate this using a quadratic *model problem*. A model problem is a problem that we may not be interested in solving in practice but want to use to understand the behavior of the algorithm. In this case, we take the model loss function to be the pure quadratic function defined by a hessian matrix H:

$$f(x) = \frac{1}{2}x^T Hx \; . \tag{10}$$

We assume that H is symmetric and positive definite. This f has gradient

$$\nabla f(x) = Hx \; .$$

With a fixed learning rate parameter, the iteration is

$$x_{n+1} = x_n - sHx_n = (I - sH)\, x_n = Mx_n \; , \;\; M = I - sH \; . \tag{11}$$

The eigenvalues λ_k are positive because H is positive definite. We want to analyze this linear iteration. The global min is $x_* = 0$, so the question is whether $x_n \to 0$ as $n \to \infty$, and, if so, how fast.

The matrix M is symmetric so it has orthonormal eigenvectors v_k and eigenvalues m_k. These satisfy $Mv_k = \lambda_k v_k$, with the normalizations $\|v_k\|_2 = 1$ and orthogonality conditions $v_j^T v_k = 0$ if $j \neq k$. The eigenvectors of M are also a good system of eigenvectors of H. The eigenvalues eigenvalues of M are related to the eigenvalues of H by

$$m_k = 1 - s\lambda_k \; , \;\; Hv_k = \lambda_k v_k \; . \tag{12}$$

The iterate x_n has an representation in terms of eigenvectors v_k and weights y_{kn} as

$$x_n = \sum_k y_{kn} v_k \; .$$

the iteration (11) can be represented in the v_k expansion of x_n as

$$y_{k,n+1} = m_k y_{nk} = (1 - s\lambda_k) y_{nk} \; .$$

Iterating this gives

$$y_{k,n} = m_k^n \, y_{0k} = (1 - s\lambda_k)^n \, y_{0k} \; . \tag{13}$$

These formulas answer our first analysis question. $x_n \to 0$ as $n \to \infty$ if and only if the coefficients converge to zero, $y_{nk} \to 0$ as $n \to \infty$ for all k. From (13), we see that this happens if and only if $|m_k| < 0$ for all k. This is equivalent to

$$-1 < 1 - s\lambda_k < 1 \; , \text{ for all } k \; . \tag{14}$$

This condition is satisfied if s is small enough. Gradient descent with fixed step size converges, for quadratic problems at least, if the step size is small enough.

We look at the formulas (13) more closely to find the rate of convergence. Assuming the basic convergence criterion (14) is satisfied, all the coefficients y_{nk} converge to zero exponentially as $n \to \infty$. The convergence rate is determined by the slowest rate, which is determined by the eigenvalue m_k with largest absolute value. The *spectral gap*[3] is determined by the "worst" eigenvalue, which is the one closest to 1 in absolute value:

$$\rho = 1 - \max_k |m_k| = 1 - \max_k |1 - s\lambda_k| \ . \tag{15}$$

This definition implies that $\| m_k| \leq 1 - \rho$ for all k, which implies that

$$|y_{nk}| \leq (1-\rho)^n \, |y_{0k}| \ , \text{ for all } n \text{ and } k \ .$$

This implies the inequality

$$\|x_n\|_2 \leq (1-\rho)^n \, \|x_0\|_2 \ . \tag{16}$$

This inequality is *sharp* which means that no better inequality of this form is possible. That's because you can take $x_0 - v_k$ with $|m_k| = 1 - \rho$, in which case convergence rate inequality (16) is an equality.

The best parameter s is the one that maximizes the spectral gap. Because of the minus sign in (15), this is found by minimizing the quantity on the right

$$\min_s \ \max_k |1 - s\lambda_k| \ . \tag{17}$$

Once we have the s_* that solves this minimization problem, we use it in (15) to find the best spectral gap, ρ_*. For the minimization problem (17), since $s > 0$ and $\lambda_k > 0$ (for all k), the numbers $1 - s\lambda_k$ are less than one. The number closest to 1 is the one with the smallest λ, which we call λ_{min}. You make $1 - s\lambda_{min}$ farther from 1 by making s larger. On the other end, if s is too large then we will have $1 - s\lambda_{max} < -1$, which makes the corresponding $|m_{max}| > 1$, which we cannot allow. We need to choose s so that $s\lambda_{max} > -1$, and the larger the gap between $s\lambda_{max}$ and -1, the better. This gap at the λ_{max} end gets bigger when s is smaller, while the gap at the λ_{min} end gets bigger when s is larger. The optimal s is the one where these gaps are the same size. Equating the these gaps gives an equation for the best s:

$$1 - (1 - s_*\lambda_{min}) = 1 - s_*\lambda_{max} - (-1)$$

The solution is

$$s_* = \frac{2}{\lambda_{max} + \lambda_{min}} \ . \tag{18}$$

The corresponding best spectral gap is found by using this s_* in either the λ_{min} or λ_{max} gap. The gap at λ_{min} becomes

$$\rho_* = s_*\lambda_{min} = \frac{2\lambda_{min}}{\lambda_{max} + \lambda_{min}} \ . \tag{19}$$

The spectral gap formula (19) is understood more clearly in a form that involves the condition number of H. This is

$$\rho_* = \frac{2}{\dfrac{\lambda_{max}}{\lambda_{min}} + 1} \ .$$

[3] The *spectrum* of a matrix is the set of eigenvalues of that matrix. The spectral "gap" is the gap (distance) between the spectrum of M and the endpoints of the interval $[-1, 1]$.

The ratio $\frac{\lambda_{max}}{\lambda_{min}}$ is the condition number of H in the 2−norm

$$\frac{\lambda_{max}}{\lambda_{min}} = \lambda_{max}\,\lambda_{min}^{-1} = \|H\|_2 \left\|H^{-1}\right\|_2 = \kappa(H) \ .$$

This,

$$\rho_* = \frac{2}{\kappa(H)+1} \ . \tag{20}$$

A typical application involving the Laplace equation might have $\kappa(H) \sim 10{,}000$. For this κ, which is common, you can ignore the difference between κ and $\kappa + 1$ in the denominator. The spectral gap is the inverse of the condition number (well, twice the inverse of the condition number), which can be tiny.

What does this imply for optimization? You can put ρ_* into the convergence rate formula (16) to see how many iterations it takes to get a specified reduction in $\|x_n\|$. The important case (for applications) has ρ_* close to 1. In that case, the approximate formula $1 - \epsilon = e^{-\epsilon}$ applies, and we get

$$\|x_n\| \sim e^{-\rho_* n} \|x_0\| \sim e^{-\frac{2n}{\kappa}} \|x_0\| \ .$$

This predicts that if you take $n = \kappa$, then the error is reduced by a factor of $e^{-2} \approx .14$. For real applications, that could mean $n = 10{,}000$ iterations gives a little less than one digit of accuracy – not very encouraging. Gradient descent is not an efficient way to find accurate solutions for ill conditioned problems.

We will see below (sort of) that this analysis predicts the local convergence of gradient descent once x_n is close enough to x_*. For this, we take H to be the hessian of the loss/objective function evaluated at x_*. We often say that a general optimization problem is well or ill conditioned[4] if $H(x_*)$ is well or ill conditioned as a matrix. This use of "conditioning" refers to the difficulty of solving a problem, not the accuracy that you can get. You might call it a misuse of the term "conditioning", but we have to accept that "conditioning" is used in both senses.

Geometry of conditioning

You can get a sense for the geometry of conditioning using $d = 2$. Then H has two eigenvalues and eigenvectors

$$Hv_1 = \lambda_1 v_1 \ , \ \ Hv_2 = \lambda_2 v_2 \ , \ \ \|v_1\|_2 = \|v_2\|_2 = 1 \ , \ \ \lambda_1 \geq \lambda_2 \ .$$

The function $f(x) = x^T H x$ has level curves $f(x) = \mathit{const}$ that are elliptical. The *major axis* of an ellipse (centered at the origin) is the line through the origin that goes to the farthest point on the ellipse. The *minor axis* is the line that goes to the nearest point. These ellipses have the same shape, so we consider just the "unit" ellipse corresponding to the $f = 1$ contour

$$E = \left\{ \, x \mid f(x) = x^T H x = 1 \, \right\} \ .$$

[4] "Ill" means "sick" or "unhealthy" in American English, but in British English "ill" also means "bad", as in "ill will" (bad feelings) or "ill wind". The terminology "well" and "ill" conditioning comes from Britain.

The major and minor axes correspond to λ_2 and λ_2 respectively. one closest point to the origin on E is

$$x_{min} = \frac{1}{\sqrt{\lambda_1}} v_1 \ , \quad \|x_{min}\|_2 = \frac{1}{\sqrt{\lambda_1}} \ .$$

This is on E because

$$f(x_{min}) = \frac{1}{\lambda_1} v_1^T H v_1 = \frac{1}{\lambda} v_1^T \lambda v_1 = 1 \ .$$

The minor axis length of E is $\frac{1}{\sqrt{\lambda_1}}$. Similarly, the major axis length is $\frac{1}{\sqrt{\lambda_1}}$, which is longer because $\lambda_2 \leq \lambda_1$. The *aspect ratio* of E is the ratio of the longer to the shorter length

$$\text{aspect ratio } = \frac{\frac{1}{\sqrt{\lambda_2}}}{\frac{1}{\sqrt{\lambda_1}}} = \sqrt{\frac{\lambda_1}{\lambda_2}} = \sqrt{\kappa(H)} \ .$$

The best possible conditioning is $\kappa = 1$, which corresponds to E being a round circle. When κ is large, the ellipse is much longer in one direction than the other. It is long and thin. A well conditioned problem (for gradient descent minimization) has round level lines. An ill conditioned problem has long thin contour lines.

The geometry in higher dimensions ($d > 2$) is similar. The level line becomes the level *surface*. The ellipse becomes an *ellipsoid*. The ratio of the farthest to nearest point of E to the origin is the square root of the condition number. The best possible condition number, $\kappa = 1$ corresponds to a round "spherical" surface. An ill conditioned problem has "thin" contour surfaces. But for $d > 2$, there are different shapes of ellipsoid corresponding to the same κ, because κ is determined by λ_{max} and λ_{min} while the shape depends on the eigenvalues between. Round level surfaces make for quick convergence of gradient descent while thin level surfaces imply slow convergence.

Preconditioning

Most large scale practical gradient descent optimization calculations use some form of *preconditioning.* Preconditioning is like finding a good initial guess. It is problem specific. It can take a lot of your time and computer experimentation to get right. It relies on heuristic or intuitive reasoning as much as theoretical analysis. There is a trade-off between what's theoretically best and what's practical or practical to know about the functions you're trying to optimize. The more of your time and problem specific information you use then better you can make it work. There is a trade-off between effectiveness and generality.

Preconditioning in an optimization problem means changing variables to make gradient descent converge faster. We discuss the case when the change of variables is linear, but it does not have to be linear. A linear change of variables takes the form of a linear substitution $x = Ay$, where A is an invertible $d \times d$

matrix. We use $g(y)$ to denote the given loss/objective function expressed in terms of the y variables

$$g(y) = f(Ay)\ .$$

The minimum of g is the same as the minimum of f and the (local or global) minimizers

$$x_* = Ay_*\ ,\ \ y_* = A^{-1}x_*\ .$$

The energy landscapes of f and g (convex or not, local minima or not) are qualitatively the same.

A good preconditioner for the local convergence rate is one that lowers the condition number of the hessian at x_* or y_*. The hessian of g is calculated from the hessian of f using the vector calculus chain rule. We use direct calculation with indices rather than matrix/vector calculations in order to be sure we understand the mechanics:

$$\frac{\partial g(y)}{\partial y_j} = \frac{\partial f(Ay)}{\partial y_j} = \sum_{l=1}^{d} \frac{\partial f}{\partial x_l}\frac{\partial x_l}{\partial y_j}\ .$$

The change of variables $x = A^1 y$ may be written out as

$$x_l = \sum_{k=1}^{d} \left(A^{-1}\right)_{lk} y_k\ .$$

This implies that

$$\frac{\partial x_l}{\partial y_j} = \left(A^{-1}\right)_{lj}\ .$$

Therefore,

$$\frac{\partial g(y)}{\partial y_j} = \sum_{l=1}^{d} \frac{\partial f}{\partial x_l}\left(A^{-1}\right)_{lj}$$

The same reasoning[5] may be applied to the functions $\frac{\partial g(y)}{\partial y_j}$. You get

$$\frac{\partial^2 g}{\partial y_j \partial y_k} = \sum_{l=1}^{d}\sum_{m=1}^{d} \frac{\partial^2 f}{\partial x_l \partial x_m}\left(A^{-1}\right)_{lj}\left(A^{-1}\right)_{mk}\ .$$

On the left are the elements of the hessian of g with respect to y variables. On the right are elements of the hessian of f with respect to x variables.

We want to express the right side in terms of the matrices H and A. The sum over m is recognized as belonging to the matrix product

$$\sum_m \frac{\partial^2 f}{\partial x_l \partial x_m}\left(A^{-1}\right)_{mk} = \left(HA^{-1}\right)_{lk}\ .$$

[5] This is a little subtle and relies on the fact that the matrix A is independent of x.

The sum over l uses the first index of A^{-1} and the first index of H, which is not part of the product matrices HA^{-1} or $A^{-1}H$. This is fixed by using the transpose of A^{-1}, which is denoted[6] A^{-T}. Therefore,

$$\frac{\partial^2 g}{\partial y_j \partial y_k} = \sum_l \left(A^{-T}\right)_{jl} \left(HA^{-1}\right)_{lk} = \left(A^{-T}HA^{-1}\right)_{jk} \ .$$

This may be written in matrix form as

$$D_y^2 g = A^{-T} D_x^2 f \, A^{-1} \ .$$

We used D^2 to denote the hessian matrix of second partial derivatives and the subscripts y or x to say which variables we're differentiating with respect to.

3 Iterations

Gradient descent methods are examples of *iterations*. Other examples are *Newton's method* and the *Gauss Newton* method. There is a general way to tell whether an iterative method has *local convergence*. Local convergence means that if x_0 is close enough to x_*, then $x_n \to x_*$ as $n \to \infty$. *Global convergence* means that $x_n \to x_*$ for any initial guess x_0. Clearly global convergence is better than local convergence. Unfortunately, there are many methods with local but not global convergence. Such a method produces x_* (as a limit) if x_0 is well chosen, but may produce nothing useful otherwise. The *basin of attraction* of x_* is the set of initial points x_0 so that $x_n \to x_*$. If x_* is locally stable, then the basin of attraction of x_* contains at least a small neighborhood of x_*. In practice, this basin of attraction can be very small and hard to find.

Local convergence and local convergence rate may be understood using local *linearization*. This is a general method that may apply to any iteration. *Iteration* (in this context) means doing same thing to a collection of parameters many times. More precisely, suppose there are d parameters arranged into a vector $x = (x_1, \cdots, x_d)$. "Doing something" to x means "doing something" to each component. That means finding $y_k = F_k(x)$. The components $F_k(x)$ form the components of a *mapping*, $F(x) = (F_1(x), \cdots, F_d(x)$. *Iteration* with F means using F to create a sequence of *iterates*

$$x_{n+1} = F(x_n) \ . \qquad (21)$$

For example, simple gradient descent with a fixed learning rate has the form (21) with $F(x) = x - s\nabla f(x)$.

We say that x_* is a *fixed point* of the iteration if $F(x_*) = x_*$. If you start an iteration at a fixed point, then all the following iterates will be x_*. We are interested in the *local* behavior of the iteration near a fixed point. A fixed point is *locally stable* if it has a basin of attraction that includes all close enough starting points. This means that there is an $r > 0$ so that if $\|x_0 - x_*\| < r$, then

[6] You can check that $\left(A^{-1}\right)^T = \left(A^T\right)^{-1}$, so it makes sense to write A^{-T} for this.

$x_n \to x_*$ as $n \to \infty$. *Warning.* Mathematicians call this property *asymptotic stability.* Stability, for them is roughly the property that if x_0 is close to x_* then the iterates stay close to x_*. The precise definition of that is not relevant here.

The *local linearization* of F is the first order Taylor approximation to F about x_*. The $d \times d$ matrix of first partial derivatives is the *jacobian* of F and is denoted in several ways, all of which we use,

$$J(x) = DF(x) = F'(x); \, , \quad J_{jk}(x) = \frac{\partial F_j(x)}{\partial x_k} \; . \tag{22}$$

In the language of perturbation theory, you approximate the change in F_j corresponding to changes Δx_k in the parameters using

$$\Delta F_j \approx \frac{\partial F_j(x)}{\partial x_1} \Delta x_1 + \cdots + \frac{\partial F_j(x)}{\partial x_d} \Delta x_d \; .$$

In vector/matrix form, this approximation is

$$\Delta F \approx DF \, \Delta x \; . \tag{23}$$

Row j of the jacobian corresponds to component j of F and column k corresponds to component k of x, which is a way to remember the roles of j and k in the elements of J in (22). The informal linearization formula (23) is a good way to think about local convergence. But at some point we will want a more precise version that uses an error bound from calculus. If all the second derivatives

$$\frac{\partial^2 F_j}{\partial x_k \partial x_l}$$

are bounded, then

$$F(x + \Delta x) = F(x) + J(x)\Delta x + O\left(\|\Delta x\|^2 \right) \; . \tag{24}$$

If Δx is small, then $J\Delta x$ is probably much larger than $O\left(\|\Delta x\|^2 \right)$. This is the justification of the linear approximation (23).

The linear approximation (23) determines (in most cases) whether an a fixed point x_* is stable or unstable for an iteration. The non-linear iteration is locally stable or unstable at x_* if the linear iteration is stable or unstable:

$$\Delta x_{n+1} = J(x_*) \, \Delta x_n \; . \tag{25}$$

For a linear iteration, stability or instability means that $\Delta x_n \to 0$ as $n \to \infty$ for all starting points $Deltax_0$, or not. To see this, take

$$\Delta x_n = x_n - x_* \; .$$

The linear iteration (25) is a consequence of the linear approximation to the nonlinear iteration (24), if we take $x = x_*$ and ignore the error term $O\left(\|\Delta x\|^2 \right)$.

The stability or instability of the linear iteration (25) is determined by the eigenvalues of $J(x_*)$. These facts are explained in any good book on linear algebra (Strang, Lax, ...). If λ_i are the eigenvalues of $J(x_*)$, then the linear iteration is stable if $|\lambda_i| < 1$ for all eigenvalues λ_i. If any eigenvalue has $|\lambda_i| > 1$ then the linear iteration is unstable. If $|\lambda_i| \leq 1$ for all λ_i but there is some λ_i with $|\lambda_i| = 1$, then the linear iteration is *neutrally stable*, meaning that there is Δx_0 so that $\Delta x_n \not\to 0$ (Δx_n does not converge to zero), and $\|\Delta x_n\| \not\to \infty$ (Δx does not "blow up"). Neutral stability of the linearized iteration usually (but not always) implies that x_* is not locally stable for the nonlinear iteration. There are examples that have a neutrally stable linearization and also local stability for the nonlinear iteration, but you will need to look at a book on dynamical systems to figure out how this can happen.

The linearization of a nonlinear iteration, if it is stable, has a *spectral gap*

$$\rho = 1 - \max |\lambda_i| \ .$$

This might look different from the spectral gap definition when talking about gradient descent. The reason is that the linearization for gradient descent has a jacobian that is the hessian of the objective function and therefore symmetric. The eigenvalues λ_i of a symmetric matrix are real, and the eigenvectors may be chosen to be an orthonormal basis. For a general J, the eigenvalues may not be real, the eigenvectors probably are not orthogonal to each other, and there may be no basis of eigenvectors, which is called *Jordan structure*. Look up "Jordan block" in a good linear algebra book. If there is a basis of eigenvectors and a positive spectral gap, then there is a $C > 0$ so that

$$\|\Delta x_n\| \leq C(1-\rho)^n \, \|\Delta x_0\| \ . \tag{26}$$

For gradient descent (with its symmetric jacobian matrix), this was true with $C = 1$. If J is not symmetric, the C can be quite large. The spectral gap is supposed to determine the convergence rate for the linear iteration and the local convergence rate for the nonlinear iteration. But there are exceptions – which numerical analysis experts know a lot about. Spectral gap analysis is usually helpful, but not always.

4 Newton's method

Newton's method is an optimization method that uses second derivatives (the hessian matrix) as well as the gradient of the objective/loss function. Newton's method is better than gradient descent in that it has much faster local convergence and that it is *affine invariant*. It has disadvantages related to needing the hessian matrix, H. It may be impractical to calculate H if the problem is too big. For example, if $d =$ one million, then H is a million by million matrix with $\frac{1}{2}d^2 =$ five hundred billion entries. It may or may not be practical to compute and store a matrix of this size, depending on the problem and the patience of the programmer. If the loss/objective function is defined in a complicated way, it may be harder to calculate H than ∇f.

One of the ways to explain Newton's method for optimization uses the *local quadratic model* approximation.

Suppose x is the current iterate is x and we want to find a better point y. In the language of Section 3, the Newton iteration is $y = F(x)$. The Newton iteration will be $x_{n+1} = F(x_n)$. We want to build F, using first and second derivatives of f, so that $x_n \to x_*$ as quickly as possible. The local quadratic model is

$$f(y) \approx Q_x(y) = f(x) + (\nabla f(x))^T (y - x) + \frac{1}{2}(y-x)^T H(x)(y-x) \; . \qquad (27)$$

We would like to choose y to minimize $f(y)$, but this is impossible. Instead, we choose y to minimize the quadratic approximation

$$y = \arg\ \min_z Q_x(z) \; . \qquad (28)$$

For this, we calculate the gradient of the quadratic approximation, which is

$$\nabla_z Q_x(z) = \nabla_x f(x) + H(x)(z - x) \; .$$

The optimal y for (28) sets this gradient to zero and solves for y:

$$y = x - H(x)^{-1} \nabla f(x) \; . \qquad (29)$$

This gives the basic Newton iteration

$$x_{n+1} = x_n - H(x_n)^{-1} \nabla f(x_n) \; . \qquad (30)$$

Local quadratic convergence

The local quadratic convergence of Newton's method is rightfully famous. If the iterations ever get close to x_*, then they converge to x_* astonishingly fast. You can understand this using the local linear approximation of the Newton iteration (30). This has the form of a general iteration (21) with

$$F(x) = x - H(x)^{-1} \nabla f(x) \; . \qquad (31)$$

The local linearization is the linear approximation to the nonlinear function (31) about a point x_* with $\nabla f(x_*) = 0$, which is a stationary point for f. We use the rules of differentiation to find

$$J(x) = DF = D\,(\,x\,) + D\left(H^{-1}(x)\,\nabla f(x)\right) \; .$$

The first term on the right is the jacobian matrix of the identity function, which is the identity matrix. In calculations, the (j,k) entry of Dx is ,

$$\frac{\partial_{x_j}}{\partial x_k} = \delta_{jk} = \text{ the } (j,k) \text{ entry of the identity matrix.}$$

The jacobian of the other term, $H^{-1}(x)\,\nabla f(x)$ seems more complicated. For one thing, it requires the derivative of the inverse of a matrix. In principle, this can be found using matrix perturbation theory, but the answer might be complicated. Another thing, H involves second derivatives of f, so its jacobian should involve third derivatives. Finally, $H(x)$, being a matrix, has two indices. The jacobian asks for derivatives of each of these entries with respect to each of the variables x_k. That is an object with three indices.

This possibly fearsome calculation goes like this. The entries of $p = H(x)^{-1}\nabla f(x)$ are

$$p_j(x) = \sum_{l=1}^{d} \left(H(x)^{-1}\right)_{jl} \frac{\partial f(x)}{\partial x_l} \ .$$

The derivative calculation is easier using the "operator" notation for ∂_{x_k} for partial derivatives. If $R(x)$ is any function of x, the partial derivatives may be denoted by

$$\frac{\partial R(x)}{\partial x_k} = \partial_{x_k} R(x) \ .$$

The (j,k) entry of the jacobian of this is (using the product rule of differentiation)

$$\begin{aligned}\frac{\partial p_j(x)}{\partial x_k} &= \partial_{x_k}\, p_j(x) \\ &= \sum_{l=1}^{d} \partial_{x_k} \left[\left(H(x)^{-1}\right)_{jl} \partial_{x_l} f(x)\right] \\ &= \sum_{l=1}^{d} \partial_{x_k} \left[\left(H(x)^{-1}\right)_{jl}\right] \partial_{x_l} f(x) + \sum_{l=1}^{d} \left(H(x)^{-1}\right)_{jl} \partial_{x_k} \left[\partial_{x_l} f(x)\right] \ . \end{aligned} \tag{32}$$

The bad three index factor (indices j,k,l) is

$$\partial_{x_k} \left[\left(H(x)^{-1}\right)_{jl}\right] \ .$$

A stationary point is, by definition, an x_* with $\nabla f(x_*) = 0$. Therefore, at x_*, the these three index factors are multiplied by zero. Evaluated at x_*, the first term of (32) is zero. The second term also simplifies in a simple yet possibly surprising way. This is because

$$\partial_{x_k}\partial_{x_l} f = \partial_{x_j} \frac{\partial f}{\partial x_l} = \frac{\partial^2 f}{\partial x_k \partial x_k} = H_{kl} \ .$$

The second sum in (32) computes the elements of the matrix product $H^{-1}H = I$. Writing this in components may be confusing or not, depending on how you like to understand things, but here it is:

$$\sum_{l=1}^{d} \left(H(x)^{-1}\right)_{jl} \partial_{x_k} \left[\partial_{x_l} f(x)\right] = \sum_{l=1}^{d} \left(H(x)^{-1}\right)_{jl} H(x)_{kl} = \delta_{jl} \ .$$

This is true for any x, not just at x_*. Altogether, we have

$$DF(x_*) = I + 0 - I = 0 \ . \tag{33}$$

This calculation shows the local quadratic convergence of Newton's method. The definition (31) of the iteration function shows that any stationary point of the objective/loss function f is a fixed point of the iteration map F. That is, if $\nabla f(x) = 0$ then $F(x) = x$. The linearization at such a fixed point is "trivial" ($DF(x_*) = 0$), so only the error term remains:

$$\begin{aligned} F(x) &= F(x_*) + DF(x_*)(x - x_*) + O\Big(\|x - x_*\|^2 \Big) \\ F(x) &= x_* + O\Big(\|x - x_*\|^2 \Big) \ . \end{aligned}$$

The Newton iteration is $x_{n+1} = F(x_n)$. The definition of "big Oh" is that there is a distance r and a constant C so that

$$\text{If } \ \|x_n - x_*\| \le r \ , \ \ \text{then } \ \ \|x_{n+1} - x_*\| \le C \, \|x_n - x_*\|^2 \ . \tag{34}$$

This is local *quadratic* convergence because the error at the next iterate is quadratic in (i.e., the square of) the error at the current iterate. It is *local* quadratic convergence because of the "locality" hypothesis $\|x_n x_*\| \le r$. In practice, local quadratic convergence implies that if the algorithm manages to get into the region where quadratic convergence applies, then the answer will be as accurate as possible in floating point arithmetic within just a few more iterations, five at most.

People often say that local quadratic convergence means that the number of "correct digits" doubles at each iteration. This comes from taking $C = 1$ and neglecting the distinction between relative and absolute accuracy. Then k digits of accuracy is $\|x_n - x_*\| \sim 10^{-k}$. The quadratic convergence inequality (34) then gives $\|x_{n+1} - x_*\| \le \|x_n - x_*\|^2 \sim 10^{-2k}$, which is $2k$ digits of accuracy. If you get an iterate with 2 digits of accuracy, then the next iterates have (because $\big(10^{-2}\big)^2 = 10^{-4}$, etc.)

$$\begin{aligned} &\|x_n - x_*\| \le 10^{-2} \\ \Longrightarrow \ &\|x_{n+1} - x_*\| \le 10^{-4} \\ \Longrightarrow \ &\|x_{n+2} - x_*\| \le 10^{-8} \\ \Longrightarrow \ &\|x_{n+3} - x_*\| \le 10^{-16} = \text{ machine precision} \ . \end{aligned}$$

Three iterations past x_n and you're done. More realistically, if $C = 100 = 10^2$ (larger than $C = 1$ but not really large), and if $\|x_n\| \le 10^{-3}$, then the error bounds become

$$\begin{aligned} \|x_n - x_*\| \le 10^{-3} &\Longrightarrow \|x_{n+1} - x_*\| \le 10^2 \, 10^{-6} \ = 10^{-4} \\ &\Longrightarrow \|x_{n+2} - x_*\| \le 10^2 \, 10^{-8} \ = 10^{-6} \\ &\Longrightarrow \|x_{n+3} - x_*\| \le 10^2 \, 10^{-12} = 10^{-10} \\ &\Longrightarrow \|x_{n+4} - x_*\| \le 10^2 \, 10^{-20} = 10^{-18} = \text{ machine precision} \ . \end{aligned}$$

With $C = 100$, quadratic convergence is ten times harder to find (10^{-3} instead of 10^{-2}), but almost as effective once you find it (four iterations instead of three).

Affine invariance

We saw that the local convergence rate for gradient descent depends on the condition number of $H(x_*)$ but the local convergence rate for Newton's method does not involve the Hessian at all. Newton