Chapter 16: Introduction to Nonlinear Programming

A nonlinear program (NLP) is similar to a linear program in that it is composed of an objective function, general constraints, and variable bounds. The difference is that a nonlinear program includes at least one nonlinear function, which could be the objective function, or some or all of the constraints. Many real systems are inherently nonlinear, e.g. modelling the drop in signal power with distance from a transmitting antenna, so it is important that optimization algorithms be able to handle them.

The problem is that nonlinear models are inherently *much* more difficult to optimize. There are twelve main reasons for this, as described below.

Reason 1: It's hard to distinguish a local optimum from a global optimum. Numerical methods for solving nonlinear programs have limited information about the problem, typically only information about the current point (and stored information about past points that have been visited). The usual available information is (i) the point ***x*** itself, (ii) the value of the objective function at ***x***, (iii) the values of the constraint functions at ***x***, (iv) the gradient at ***x*** (the derivative in a 1-variable problem), and (v) the Hessian matrix (i.e. the second derivative in a 1-variable problem). This is enough information to recognize when you are at a local maximum or minimum, but there is no way of knowing whether there exists a different and better local maximum, or even how to proceed towards it. This also means that there is no way to easily determine where the global optimum is.

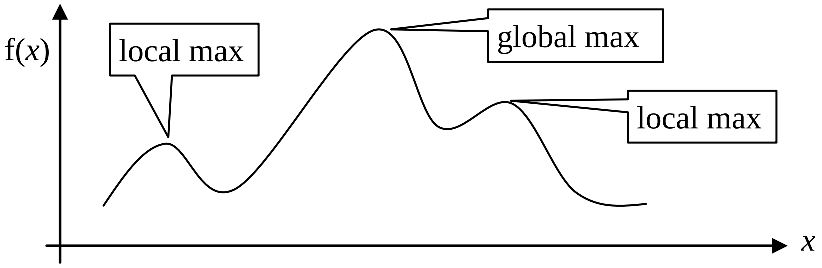

Figure 16.1: Local vs. global maxima for a 1-variable problem.

These ideas are illustrated for a 1-variable unconstrained problem in Fig. 16.1. Formally, a *local optimum* ***x***** is a feasible point that has a better value than any other feasible point in a small neighbourhood around it, say within some small ball with radius ε. The *global optimum*, on the other hand, is the point with the best value of the objective function anywhere in the feasible region. Note that the global optimum will also be a local optimum.

Reason 2: Optima are not restricted to extreme points. There are a limited number of places to look for the optimum in a linear program: we need only check the extreme point, or corner points, of the feasible region polytope. This isn't so for nonlinear programs: an optimum (local or global) could be anywhere: at an extreme point, along an edge of the feasible region, or in the interior of the feasible region! This is illustrated in Fig. 16.2, which requires a little explanation.

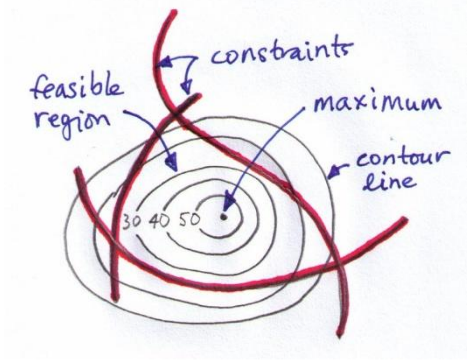

Figure 16.2: Optimum is at an interior point.

We actually need n+1 dimensions to illustrate an n-dimensional function. The added dimension is for the function value itself, i.e. f($\boldsymbol{x}$). You can see this in Fig. 16.1, which requires two dimensions to show a 1-dimension function: the horizontal axis is for the variable x and the vertical axis is for the function value f(x). However we will frequently want to show 2-dimension functions to illustrate ideas in nonlinear programming. So how do we show the 3rd dimension for the function value? One easy way is to use *contour lines* (also known as *level sets*). These are lines that connect a set of points that all have the same value of the function. You may be familiar with this concept from hiking maps which use them to show the topography.

Fig. 16.2 shows the contour lines for an objective function whose maximum point is in the interior of the feasible region, far from the extreme points or the edges. Three of the contour lines are labelled with the value of the objective function for all of the points on that particular line. In this case the optimum is a local maximum, as shown by the fact that the labels on the contour lines increase as we near the "centre" of the concentric ellipses.

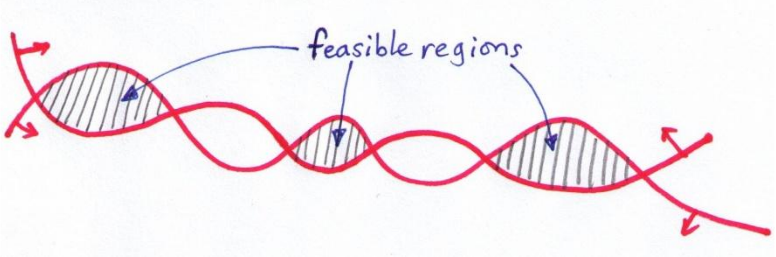

Figure 16.3: Multiple disconnected feasible regions.

Reason 3: There may be multiple disconnected feasible regions. Because of the way in which nonlinear constraints can twist and curve, there may be multiple different feasible regions. So even if you are able to find the optimum within a particular feasible region, how do you know that there isn't some other disconnected (also called *discontiguous*) feasible region that you haven't found and explored? This is illustrated in Fig 16.3 in which the small arrows indicate the feasible sides of the nonlinear inequalities.

Reason 4: Different starting points may lead to different final solutions. This is related to the way in which nonlinear solvers work. One very common algorithm chooses a direction for search (e.g. the direction of steepest descent for minimization), and then finds the best value of the objective function in that direction; the process then repeats until there is no improvement in the value of the objective function. Methods like this will typically descend to the bottom of whatever valley the starting point happens to be in. Because there may be multiple different valleys, starting at some other point may return a different final solution point and objective function value. The difficulty is compounded worse when the problem is constrained: not only are there multiple local optima, but these may be in discontiguous feasible regions! One option is to simply try restarting the solver from many different initial points, but this can be quite time-consuming.

Reason 5: It may be difficult to find a feasible starting point. The phase 1 routine in linear programming will either find a point that satisfies all of the constraints (and hence is feasible), or it will accurately determine that no feasible points exist anywhere. You have no such guarantees in NLP. Most NLP phase 1 procedures try to minimize some measure of the total infeasibility, e.g. the sum of squared constraint violations. For example, if the constraint is $c(\boldsymbol{x}) \leq 12$ and at the current point $c(\boldsymbol{x}) = 15$, then the constraint violation is 3 and the squared violation is 9. If you can find a point at which the sum of the squared constraint violations is zero, then you have reached the global optimum and found a feasible point. However this is in itself a nonlinear programming problem to solve, and also faces the difficulty that solvers are not able to distinguish local and global optima. Solvers get stuck in local optima in which the sum of the constraint violations is not zero. Again, you can try restarting the phase 1 procedure from multiple different starting points, but if you never find a feasible point you are still left unsure as to whether the model is actually infeasible, or whether it is feasible but you simply didn't start your solver in the right place.

Reason 6: It is difficult to satisfy equality constraints (and to keep them satisfied). In linear programming, the phase 1 routine finds a solution that satisfies all of the constraints (including the equalities), and thereafter these are never violated. In NLP, finding a solution that satisfies curving and twisting equations is difficult in and of itself, but even if a solution is found at some point, the equality may again be violated when the algorithm tries to move to another point that has a better value of the objective function.

Reason 7: There is no definite determination of the outcome. In linear programming, the solver (generally) has only a couple of outcomes: (i) the model is feasible and there is a globally optimum solution point, (ii) the model is feasible but unbounded, or (iii) the model is infeasible. These are very definite statements about the status of the outcome (neglecting various error-related outcome possibilities). Things are much less certain for NLP. The solver may report an optimum, but at best it can check certain conditions to guarantee that the point is a local optimum: it is not able to say whether the point is a global optimum. It may also continue improving the value of the objective function for a long time, but will not be able to say whether the model is unbounded. And as discussed in Reason 5, you are never sure whether the model is actually infeasible if no feasible solution is found.

Reason 8: There is a huge body of very complex mathematical theory and numerous solution algorithms. This is not surprising, given the fact that nonlinear functions have a much wider range of behaviours and characteristics than linear functions. But it does make it hard to know how to solve the problem at hand. How do you decide which algorithm to apply, and can you follow the complex steps correctly?

The reasons why NLP is so much harder than LP given above are mainly based on the theory of nonlinear functions. But there are even more reasons why NLP is hard! These are based on more practical considerations.

Reason 9: It is difficult to determine whether the conditions to apply a particular solver are met. Many solution algorithms require that the functions in the model have particular characteristic, which could relate to their algebraic structure (e.g. must be quadratic, or polynomial, etc.), or their shape (e.g. convexity and concavity, i.e. whether they curve up or down everywhere). The problem is that it is often difficult in practice to determine whether these conditions are satisfied or not. How would you know whether a constraint is concave in the region of interest? Or whether the constraints taken together form a convex set (more on this later)? There are very few tools for assessing models prior to selecting the solution algorithm and associated NLP solver. This is a ripe area for research.

Reason 10: Different algorithms and solvers arrive at different solutions (and outcomes) for the same formulation. Almost all NLP algorithms proceed in a step-wise fashion, iteratively improving the initial point until certain stopping conditions are met. But each algorithm will likely have a completely different trajectory of points from the initial point to the final point that is output as the solution. This means that different algorithms may terminate at different local optima, and hence will return different solutions, even for the exact same formulation and initial point. Further, some algorithms may terminate successfully at local optima while others move off in different directions and are unable to even find a feasible point, so the outcomes are completely different. Even worse, different implementations of the *same* algorithm (i.e. different solvers) may also give different results due to different choices of parameters and internal heuristics!

Reason 11: Different but equivalent formulations of the model given to the same solver may produce different solutions and outcomes. Here are two mathematically equivalent formulations of a constraint function: (i) $x^2 + y^2 \leq 25$, or (ii) $a = x^2$, $b= y^2$, $a + b \leq 25$. Solvers may treat these formulations differently however, perhaps finding the second version easier to handle by first dealing with a simple linear inequality in a and b, and then later solving for the values of x and y. Thus the same solver may have a different trajectory of points for different formulations of the same model, and hence arrive at different final solutions and outcomes. This is a marked contract to LP, where equivalent but different formulations of the same model will normally have the same outcome (though the sequence of intermediate corner point solutions will be different).

Reason 12: Using the available nonlinear solvers can be complex. Most solvers have a large number of user-settable parameters which control details of how the solver will operate. It's hard to know in advance how these parameter settings will affect the solution. What type of line search should be used? What tolerances should be used? What is the largest acceptable basis inverse condition number that should be accepted? Etc. There may be 50 such parameters. Fortunately these are normally set at good default values, but adjusting them takes expert knowledge.

But there is some good news! Some things that were difficult in the past can now be done relatively easily:

- **Derivatives.** It used to be difficult to get function derivatives, which are often required by solvers. There were two approaches: numerical approximation by finite differences, or requiring the modeller to provide code for the derivative in addition to the code for the original function. This latter approach was often problematic when the derivative code and the function code did not match up (due to typos etc.). In that case the solver would not operate correctly. Fortunately the more recent technique of automatic differentiation is largely available. This analyzes the function expression to determine the correct expression for the derivative.

- **Input formats.** At one point, each solver had its own particular input formats to describe the model, or else required that the user attach a subroutine (written in a computer language like Fortran or C) that described the model. Thus if you found that one solver was not effective on the problem at hand it was a tedious and error-prone job to recode the model into a different format so you could try a different solver. Nowadays, however, most solvers can be linked to some standard optimization modelling languages, such as AMPL, GAMS, etc. The means that the model can be written just once, and then multiple different solvers can be tried.

And there is other good news as well:

Relatively few commercially implemented algorithms. Though the academic literature describes a vast range of solution methods, the number of algorithms that have actually been implemented in readily available commercial (or even free open-source) solvers is much smaller. Hence for practical purposes, there are relatively few algorithms that you need to understand in depth. Typical commercially implemented algorithms include the generalized reduced gradient method, quadratic programming and successive quadratic approximation, and barrier methods. As time permits we will spend some time looking at these algorithms (after laying out some groundwork).

Modelling languages and systems. In addition to eliminating the input format problem as described above, modelling languages have evolved into larger systems that include additional features like the ability to perform a pre-solve step that simplifies the model and tightens the bounds, perhaps identifying errors, automatic differentiation, reporting and programming features, etc. The language systems also allow the modeller to link to databases so that very large models can be constructed in a convenient way.

Model analysis tools. There are now a few tools that analyze nonlinear models, and that can be applied either before solution to help decide what type of algorithm might be best, or after solution, e.g. to determine the reasons for an unexpected outcome. MProbe (http://www.sce.carleton.ca/faculty/chinneck/mprobe.html) is one such tool: it empirically tests functions to estimate their shapes: Everywhere convex? Everywhere concave? Both? Almost linear? Etc. It also looks at the effectiveness of the constraints in terms of eliminating feasible solutions and has a number of other useful tools and analyses.

Solving nonlinear programs is difficult, but not impossible: the available solvers are improving rapidly for one thing. But there are a few things that you can do to increase the probability of a successful solution:

- **Always use a modelling language.** This will allow you to easily switch between solvers which may use different algorithms. This will give you a better chance of finding a solution if your first choice solver fails. It also gives you access to the other tools in the associated modelling system such as the nonlinear presolver, automatic differentiation, etc.

- **Look for a simpler formulation.** Sometimes a nonlinear function can even be reduced to a linear form and solved by an LP by simple reformulation. Look at Reason 11 above, for example.

- **Know the characteristics of your model before choosing a solution algorithm.** For example, is it composed entirely of quadratic functions? If so, then there are specialized and highly effective algorithms for solution. Can it be well-approximated by piece-wise linear functions? Then it can be reduced to a linear program. Perhaps you can apply software like MProbe to determine the shapes of the constraint and objective functions; this information will help you decide which solution algorithm to apply.

- **Provide a good starting point.** The closer your initial starting point is to the eventual optimum point, the better chance that the solver will successfully find it. But how do you know where the optimum might be? You may have solved a very similar model before, so you can use the previous solution as your new starting point. This is actually pretty common, e.g. where a company solves the same problem day after day, with just minor changes to the inputs (e.g. changes to prices, or to availability of material). You may be able to guesstimate a good initial point based on a heuristic analysis using a simpler model that is easy to solve (even an LP). Or you may just do some simple analysis. There is recent work on effective methods for improving any selected starting point before passing it to the solver (search on "Constraint consensus": yes I wrote those papers). As the old joke goes, "the best way to solve a NLP is to start at the optimum". At this point you may not see the humour in that statement, but at the end of this course you will find it hilarious!

- **Put reasonable bounds on all variables.** This narrows the search space so the solver does not spend a lot of time exploring useless regions of the variable space.

- **Make the best use of the solver parameter settings.** Many of the default settings for the parameters in a given solver have been carefully tuned to give the best performance over a broad range of model types, so it's usually best to leave them alone. But sometimes there are parameters that can greatly improve the solver performance for your particular model, especially if it has special characteristics, however it takes some expertise to make effective parameter adjustments.

Local Optimization of Nonlinear Programs

Most readily available software for nonlinear programming deals only with local optimization of nonlinear programs. In fact, when most people say "nonlinear programming" they usually mean local optimization; they will normally say "global optimization" when they have that meaning in mind. In any case, most global optimization techniques use local optimization routines in their operation, so we will start by looking at local optimization. And as usual we'll start with the simplest possible cases and work our way upward from there.

One-Dimensional Unconstrained Local Nonlinear Optimization

The simplest possible case of nonlinear optimization has only a nonlinear objective function and no constraints (hence "unconstrained") and involves just one variable. For example, you need to find the maximum value over the one-dimensional unconstrained function sketched in Fig. 16.1. This looks easy in Fig. 16.1, but recall that a computer method does not have a nice sketch to look at: it only has individual points and the associated objective function values that it calculates. What it sees after the first point it tests looks something like Fig. 16.4. The question is this: *where* should we evaluate the objective function? An algorithm does this in an organized manner that leads to a local optimum. It uses information revealed by the various test points to determine where to test next.

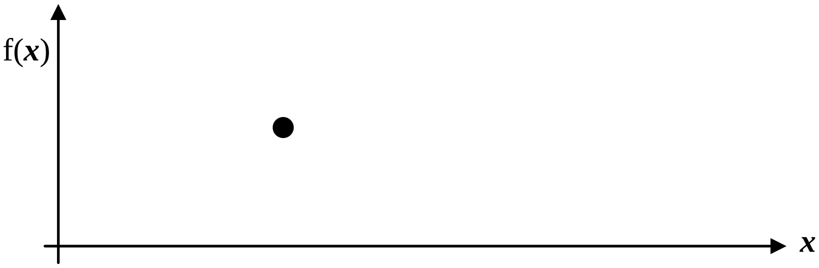

Figure 16.4: The function as seen by a computer after a single test point.

For a differentiable function, the information available at test point x is (i) the value of the function $f(x)$, (ii) the derivative of the function $\frac{df(x)}{dx}$, and (iii) the second derivative of the function $\frac{d^2f(x)}{dx^2}$.

From elementary calculus, recall that a local maximum point x^* of an unconstrained function has two properties: (i) the derivative is zero, i.e. $\frac{df(x^*)}{dx} = 0$, meaning that we are at a level spot on the function, and (ii) the second derivative is negative, i.e. $\frac{d^2f(x^*)}{dx^2} < 0$, meaning that the slope tends downwards in both directions from this spot. For minimization, the second condition changes to $\frac{d^2f(x^*)}{dx^2} > 0$.

So the fundamental question is how to search for a local optimum point x^* that meets these two conditions in an efficient manner. In computer implementations this is usually done via a *numerical* search method (it's numerical because it relies on generating and working with numbers, e.g. function values, as opposed to an *algebraic* method that works with mathematical symbols). There are numerous numerical methods for one-dimensional local optimization: bisection search, golden section search, Fibonacci search, etc. We'll take a quick look at bisection search as an example of the genre.

Bisection Search for a Local Maximum of a One-Dimension Nonlinear Objective Function

The bisection method applies when there is a single continuous variable (dimension) in a differentiable nonlinear function (it must be differentiable because we need to use the derivative). It starts with some initial range of the single variable that is known to encompass a local maximum, and then iteratively halves this range, always guaranteeing that the remaining range includes a local maximum. How does it know that the range encompasses a local maximum? It uses the derivatives at the endpoints of the range: there is at least one local maximum in the range (and there may be several) if the left endpoint has a positive derivative (the tangent line is "going uphill"), and the right endpoint has a negative derivative (the tangent line is "going downhill"): see Fig. 16.5. This is the condition that we make sure to maintain at all times.

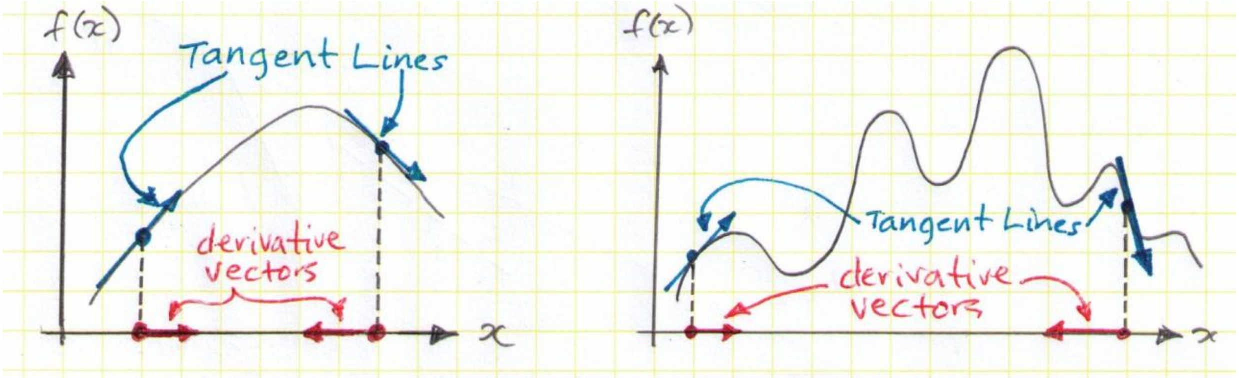

Figure 16.5: The endpoints of the range bracket at least one local maximum

But how do we find the initial range? There are many ways to do an initial coarse search to find two endpoints that satisfy the conditions. For example you could start with a random point: if its derivative is positive, then you have your initial left endpoint, and if its derivative is negative then you have your initial right endpoint (and in the unlikely case that its derivative is zero then you are at a *critical point* which could be a local maximum, a local minimum or a *saddle point*, more about this later). Once you have one endpoint then you could take a step in the specified direction (if you have a left endpoint then the step will increase the variable value; if you have a right endpoint then the step will decrease the variable value). If you don't find a suitable endpoint of the opposite type after one step, then perhaps double the stepsize and move again. Eventually you should end up with a pair of endpoints that satisfy the derivative conditions, so you know there is a local maximum point between them. Now the bisection algorithm can begin. The steps are summarized in Algorithm 16.1. Note that the output solution x^* is guaranteed to be within ε of the true optimum.

Note that the selection of an error tolerance, typically represented by the epsilon symbol ε, is a feature of most numerical algorithms, including both linear and nonlinear programming. It's a side effect of using binary representations of real numbers: they can't be represented exactly with a limited number of bits, so instead of checking whether two numbers are exactly equal, we have to check whether they are approximately equal, i.e. within ε of each other.

I've omitted some details in Algorithm 16.1 such as the case when $f(x')$ has a derivative that is equal to zero (or within ε of zero). We will also return later to the question of how to calculate the derivatives that are needed in the bisection algorithm. For now it's enough to know that they can be estimated by numerical techniques – or calculated exactly using more modern methods.

The main lesson is that there are techniques for finding an unconstrained local optimum in one dimension. This is also known as a *line search* since we searching along a single line.

Start-up:

- Choose an acceptable error tolerance ε
- Find an initial left endpoint x_L such that $\frac{df}{dx}(x_L) > 0$
- Find an initial right endpoint x_R such that $\frac{df}{dx}(x_R) < 0$ and $x_R > x_L$

Iterate:

- Bisect the range: $x' = (x_L + x_R)/2$
- If $\frac{df}{dx}(x') > 0$ then set $x_L = x'$, else if $\frac{df}{dx}(x') < 0$ then set $x_R = x'$.
- If $(x_R - x_L) \leq 2\varepsilon$ then exit with solution $x^* = (x_L + x_R)/2$, else repeat iteration.

Algorithm 16.1: Bisection Search

Multi-dimension Unconstrained Local Nonlinear Optimization

As you will see throughout these notes, many methods for nonlinear optimization build upon simpler methods. We will look at a very common method for multi-dimension unconstrained local nonlinear optimization that builds directly upon the line search method for the one-dimension case. This is known as the *method of steepest ascent* for maximization and the *method of steepest descent* for minimization, and is also sometimes called *hill-climbing*.

We will look at the case of maximizing $f(\boldsymbol{x})$ over all possible values of $\boldsymbol{x}$, where $\boldsymbol{x}$ is a vector of multiple variables (and hence shown in lowercase boldface). As for the one-dimension case, there may be multiple optima, but we will concentrate on finding a single local maximum. The main idea for maximization is simple: at the current point, find the steepest uphill direction, then conduct a line search in that direction to find a local maximum; update the point to the local maximum; repeat this process until the stopping conditions are met. This reduces the multi-dimension problem to a series of single-dimension problems that we already know how to solve using a technique like bisection search.

The first question is: how do you find the steepest uphill direction? In a one-variable problem this is simple: it is the direction given by the sign of the first derivative of the function. In Fig. 16.5 the tangent arrows on the functions show an upward slope at x_L and a downward slope at x_R. But note that the <u>direction</u> specified by the derivative occurs in the variable space: this is a one-dimensional problem, so we can only increase or decrease the single variable. The search direction is either right (increase the variable) or left (decrease the variable). Recall that sketches of the function value include an extra dimension which shows the value of the function. In Fig. 16.5 the search direction is simply along the x axis.

For multi-dimension problems, the direction of steepest ascent is given by the *gradient vector* of the function, which is the multi-dimension analog of the single-dimension derivative. The gradient vector is defined as follows:

$$\nabla f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \frac{\partial f(\boldsymbol{x})}{\partial x_3}, \dots \frac{\partial f(\boldsymbol{x})}{\partial x_n}\right]$$

i.e. the vector of first partial derivatives at the point ***x*** where there are *n* variables. The length of the gradient vector is related to the rate of increase of the function in the gradient direction.
The second question is: how do we identify a local maximum point? In the single dimension case this is done by finding a point at which the first derivative is zero (which identifies a critical point that is either a local maximum, a local minimum, or a saddle point), and at which the second derivative is negative (indicating that the function tends downwards in both the forward and backward directions. These two conditions guarantee a local maximum. For a local minimum, the second condition reverses: the second derivative must be positive.

With the gradient as the multi-dimension analog of the derivative, the first condition in the multi-dimension case is simply that $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$, i.e. that all partial derivatives are zero. This again identifies a critical point. But the multi-dimensional analog of the second derivative is more complicated. It is called the *Hessian* matrix, and is defined as follows. Where:

$$h_{ij} = \frac{\partial^2 f(\boldsymbol{x})}{\partial_i \partial_j}$$

Then the Hessian matrix is defined as the square matrix of second partial derivatives:

$$\boldsymbol{H} = \begin{bmatrix} h_{11} h_{12} \dots h_{1n} \\ h_{21} h_{22} \dots h_{2n} \\ \vdots \quad \vdots \qquad \vdots \\ h_{n1} h_{n2} \cdots h_{nn} \end{bmatrix}$$

Using the Hessian is also complicated. First you construct subsets of ***H*** where ***H***$_i$ indicates the subset created by taking the first *i* rows and columns of ***H***. Then you calculate the determinant of each of the *n* subsets created this way (where the enclosing vertical lines indicate the determinant operator):

$$H_1 = |h_{11}|,\ H_2 = \begin{vmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{vmatrix}, \cdots$$

Note that the H_i are not matrices, just scalar numbers given by the determinants. Now you determine the category of the Hessian matrix from the sign patterns of the determinants of the H_i, as follows:

- ***H*** is *positive definite* at ***x*** if H_i > 0 for i=1…n
- ***H*** is *negative definite* at ***x*** if H_1 < 0 and the remaining H_i alternate in sign.
- ***H*** can be *positive semi-definite* or *negative semi-definite* if some of the values that are supposed to be nonzero turn out to be zero.

So if the gradient vector is entirely zero at some point ***x*** (i.e. $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$), then the category of the Hessian matrix at ***x*** determines whether ***x*** is a maximum, minimum or saddle point:

- ***x*** is a local maximum if $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$ and H is negative definite.
- ***x*** is a local minimum if $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$ and H is positive definite.

- $\boldsymbol{x}$ is a saddle point if $\nabla f(\boldsymbol{x}) = \boldsymbol{0}$ and H is neither positive semi-definite nor negative semi-definite.

While we are on the topic, what is a saddle point anyway? In two dimensions it's a critical point ($\nabla f(\boldsymbol{x}) = \boldsymbol{0}$) which is simultaneously a local maximum in some hyperplane and a local minimum in another; this makes it look like a saddle. See Fig. 16.6: the red point is a local minimum in the left-to-right direction and simultaneously a local maximum in the front-to-back direction.

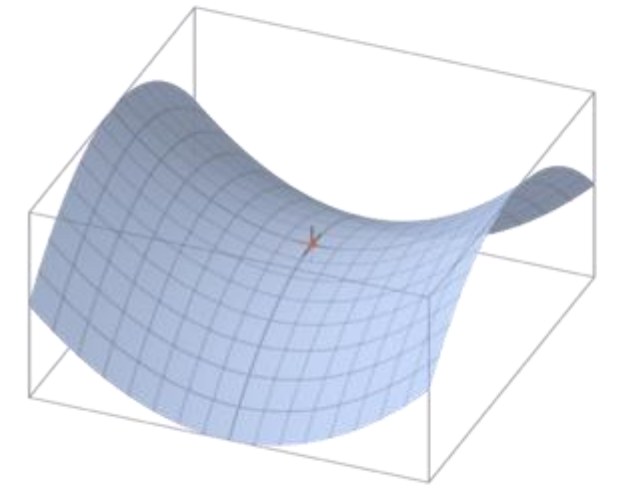

Figure 16.6: Saddle point for the function *f*(*x*)=x^2–y^2 (image from Wikipedia).

OK, now we can identify a local optimum point if we happen to be at one, but how do we go about actually finding a local optimum point? As described earlier the steepest ascent concept is simple: at the current point, find the steepest uphill direction, then conduct a line search in that direction to find a local maximum; reset your point to the local maximum; repeat this process until the stopping conditions are met. But there are a few important wrinkles in the details. Algorithm 16.2 summarizes the method.

Strictly speaking, Algorithm 16.2 only finds a critical point since it stops in Step 2 when the elements of the gradient vector are all small enough. However the Hessian of the output point can be checked to see whether it really is a local maximum. It usually is because the method is always searching in an uphill direction.

```
Start-up:
  • Choose an acceptable error tolerance ε.
  • Choose a starting point x'.
Iterate:
1. Calculate ∇f(x').
2. If |∇f(x')| ≤ ε then exit with x' as the solution.
3. Set x" = x'+t ∇f(x'), where t ≥ 0.
4. Use a line search to find t* such that f(x") is a local maximum Note: x' and ∇f(x') are fixed.
   Only t varies during the line search. Use t* to calculate x" using the expression in Step 3.
5. Set x' = x".
6. Go to Step 1.
```

Algorithm 16.2: Steepest ascent algorithm to find an unconstrained local maximum point.

Steps 3 and 4 set up the one-dimensional line search in the gradient direction. $\boldsymbol{x}'$ and $\nabla f(\boldsymbol{x}')$ are fixed during the line search and only t is varied. As Step 3 shows, varying t just gives different values of $\boldsymbol{x}''$ in the gradient direction. t expresses how far to move in the gradient direction and

$\boldsymbol{x}''$ is the actual point in n-space at that distance along the gradient vector from $\boldsymbol{x}'$. The output point corresponds to a local maximum in the gradient direction.

Consider this example: maximize $f(\boldsymbol{x}) = -x_1^2 - 4x_2^2 + 8x_1 + 16x_2$. The gradient for this function at some point $\boldsymbol{x}'$ is given by $\nabla f(\boldsymbol{x}')=[-2x_1'+8, -8x_2'+16]$. So points along the gradient direction from $\boldsymbol{x}'$ are given by $\boldsymbol{x}'' = \boldsymbol{x}'+t\nabla f(\boldsymbol{x}')$, which amounts to $\boldsymbol{x}'' = (x_1', x_2') + t[-2x_1'+8, -8x_2'+16]$ in which x_1' and x_2' are fixed and t is the only variable. The setup and first iteration in solving this problem might look like this:

Start-up: choose ε=0.000001 and $\boldsymbol{x}'$=(0,0). Two notes here. First, tolerances in the range of 10^{-6} are fairly standard in real solvers. Second, while the origin is often used as the initial point, it can be a poor choice, e.g. due to divide by zero errors. There are much better heuristics for choosing the initial point: see the note on page 6 about providing a good starting point for the solver.

The objective function value at (0,0) is 0. The first iteration proceeds as follows:

1. $\nabla f(0,0) = [-2(0)+8, -8(0)+16] = [8,16]$.

2. $[|8|,|16|] > [10^{-6}, 10^{-6}]$ so continue.

3. $\boldsymbol{x}'' = (0,0) + t[8,16] = (8t,16t)$.

4. The one-dimensional line search operates by varying t in the function $f(\boldsymbol{x}'')$, or equivalently, on $f(t) = -(8t)^2 - 4(16t)^2 + 8(8t) + 16(16t)$, obtained by substituting the elements of $\boldsymbol{x}''$ in Step 3 into the original function to create a one-dimensional function of t. Let's assume that we use a bisection search and that it returns t^*=0.147. Using the expression in Step 3, this means that $\boldsymbol{x}''$ = (0,0) + 0.147[8,16] = (1.18, 2.35). The objective function has improved: f(1.18, 2.35) = 23.529.

5. Now we would go to the next iteration, starting at the point (1.18, 2.35). The gradient at this point is [5.6471, -2.8235] so we perform a one-dimensional search along $\boldsymbol{x}''$ = (1.18, 2.35) + t[5.6471, -2.8235], yielding an optimum solution of t^*=0.312. Hence the new point is $\boldsymbol{x}''$ = (1.18, 2.35) + 0.312[5.6471, -2.8235] = (2.942, 1.469) where the objective function has again improved: f(2.942, 1.469) = 29.751.

The solution gradually moves towards the optimum point at f(4,2) = 32. It actually stops a little short of (4,2) because of the solution tolerances, but the final point is very close. As is typical of steepest ascent/descent methods, the solution trajectory zig-zags towards the optimum point, taking smaller and smaller steps, and making smaller and smaller improvements in the objective function value as it proceeds. A typical steepest descent solution trajectory for a different problem is shown in Fig. 16.7.

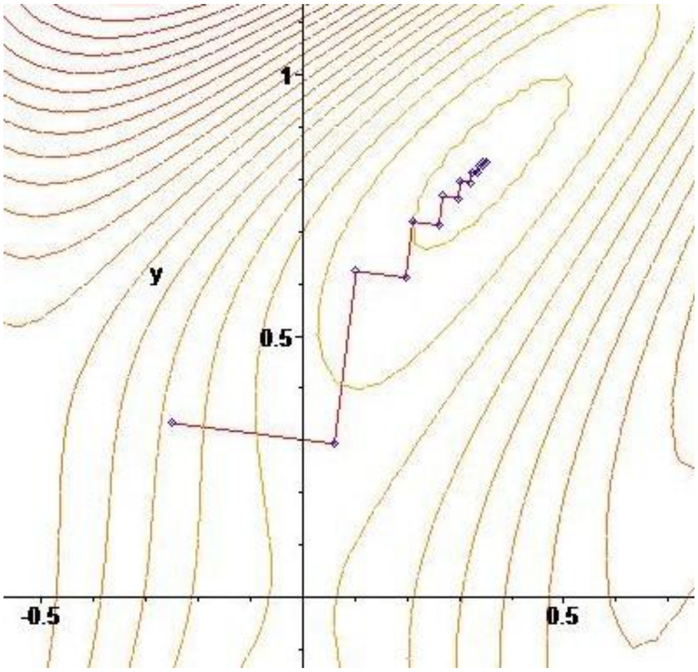

Figure 16.7: Typical zig-zagging steepest ascent/descent solution trajectory (image from Wikipedia).

Obtaining Derivative and Gradient Information

There are several ways to obtain the derivative and gradient information that the steepest/ascent (and many other solution methods) rely on. As described above, years ago the only way to get any information about either the objective function or its derivatives was to supply a subroutine which provided code for these functions. So the user had to analytically derive the derivative function and code it correctly in a programming language such as Fortran or C. When mistakes were made in this process, the derivative function was not correctly matched with the objective function and the entire solution process would go awry.

To avoid this problem, users could also usually ask for the derivatives and gradients to be estimated by *finite difference* methods. Methods such as this evaluate the objective function at the point in question and again at a different nearby point. The derivative is then estimated by dividing the differences in the function values by the distance between the points. In the single variable case there are three variations: *forward differencing* or using a point that is slightly larger than the current point, *backward differencing* or using a point that is slightly smaller than the current point, and *central differencing* or using one point slightly smaller than the current point and one point slightly larger than the current point. Solvers could use the estimated derivatives and gradients directly, or could use them as a check on the values returned by the derivative subroutines, raising an error if the two results differed by a significant amount.

Nowadays there are better alternatives. Most of the well-known modelling languages provide automatic differentiation. If the model is expressed in one of these languages, then the associated system can provide the derivative or gradient for a function at any point. It does this by parsing the expression of the function so that the derivatives and gradient expressions are discovered and hence calculated accurately. Alternatively, symbolic mathematics systems such as Mathematica or Maple can provide the analytic expression for the derivative, which then can be coded into the program.