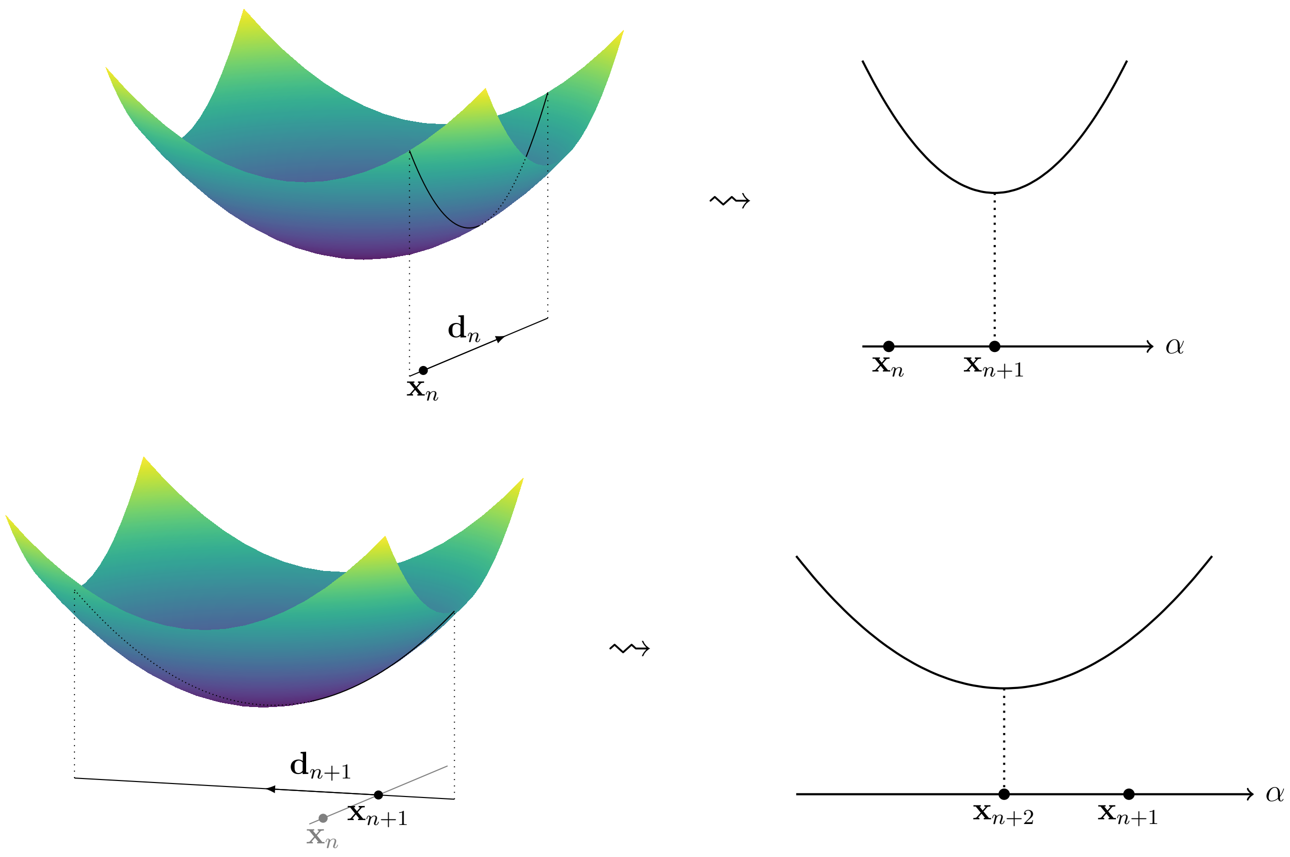
3.5 Line search methods and gradient methods

We now step things up and consider functions of more than one variable. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a (multi-dimensional) function that we wish to minimize.

The basic idea of a **line search method** is to start the search for a minimiser at a point p, and then search for the next point in a chosen direction $\mathbf{d}$. The chosen direction forms a cross section of the original function, which can be treated as a function in one variable. Thus, the one-dimensional search methods of the previous sections can be employed to find a local minimiser *in that direction*, after which a new direction is chosen and the process is repeated.



Formally, at each step n, a **search direction** $\mathbf{d}_n$ and a **step size** α_n is chosen, and the point $\mathbf{x}_{n+1}$ is defined in terms of $\mathbf{x}_n$ as

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \alpha_n \mathbf{d}_n. \tag{3.3}$$

But first we must find a sensible way of choosing the search direction $\mathbf{d}_n$ at each step. Recall that the rate of change of f in a direction $\mathbf{d}$ with $\|\mathbf{d}\| = 1$ is given by $\nabla^T f(\mathbf{x})\mathbf{d}$. Since we seek a minimiser, it makes sense to choose $\mathbf{d}$ so that the rate of change is as negative as possible, leading us in the direction of steepest descent. When $\|\mathbf{d}\| = 1$, we have $\nabla^T f(\mathbf{x})\mathbf{d} = \|\nabla f(\mathbf{x})\| \cos(\theta)$, where θ is the angle between the gradient and $\mathbf{d}$. Therefore the maximal *increase* is when $\theta = 0$, and the maximal *decrease* is when $\theta = \pi$. So, we take the search direction to be

$$\mathbf{d}_n = -\nabla f(\mathbf{x}_n),$$

giving the update procedure

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \nabla f(\mathbf{x}_n). \tag{3.4}$$

Using the affine approximation, for small enough α_n,

$$f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n) - \alpha_n \|\nabla f(\mathbf{x}_n)\|^2 + \alpha_n^2(\cdots) < f(\mathbf{x}_n), \tag{3.5}$$

which confirms the search is heading in the right direction.

A **gradient method** is a line search method where the direction $\mathbf{d}_n$ at each step is given by $\mathbf{d}_n = -\nabla f(\mathbf{x}_n)$. It remains to be seen how to choose the step size α_n. One can take small steps, which is time-consuming as at every step one has to evaluate the gradient, or take bigger steps which might be quicker but could lead to a more zig-zag path to the solution (if you get there at all). More formally, one can ask whether it is more efficient to perform more iterations of (3.3) (while assuring that $f(\mathbf{x}_{n+1}) < f(\mathbf{x}_n)$ at each step), or spend more iterations to find a good approximation to the optimal α_n.

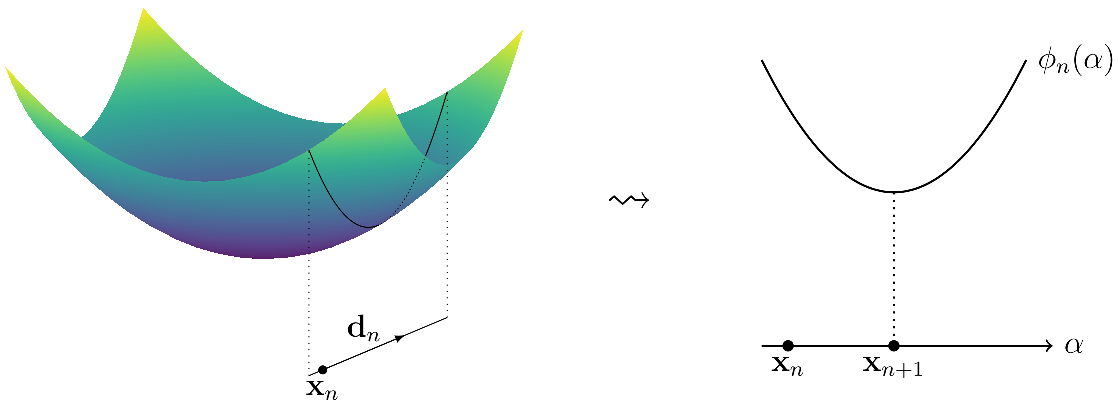
One way to choose the step size is a generalisation of **Newton's method**. Consider $f\colon \mathbb{R}^n \to \mathbb{R}$. If the Hessian matrix $D^2 f$ is invertible, we may choose the step size α_n as $[D^2 f(\mathbf{x}_n)]^{-1}$ so that the iteration step becomes

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [D^2 f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n).$$

Ideally, the step size α_n is chosen such that the objective function $\phi_n\colon \mathbb{R} \to \mathbb{R}$,

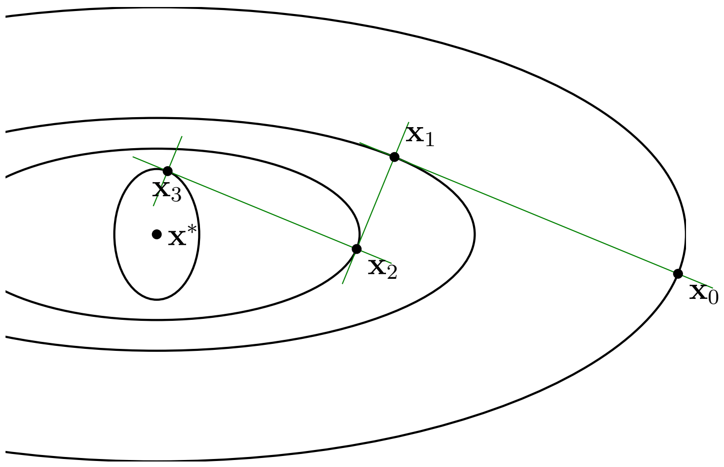
$$\phi_n(\alpha) = f(\mathbf{x}_n + \alpha \mathbf{d}_n),$$

is minimised. The function ϕ_n can be viewed as the cross section of f in the direction $\mathbf{d}_n$.



Since $\mathbf{x}_n$ and $\mathbf{d}_n$ are already established, the function ϕ_n can be minimised for α using any of the one-dimensional search methods considered earlier. The gradient method where, at each step, the step size α_n is chosen by minimising the function ϕ_n, is called the **method of steepest descent**. It is a gradient method where the step size is chosen to achieve the maximal amount of decrease at each step; i.e., it optimizes the line search.

One feature of this method is that it moves in orthogonal steps, as depicted below.



This can be proven as follows: using (3.4), we have

$$\langle \mathbf{x}_{n+1} - \mathbf{x}_n, \mathbf{x}_{n+2} - \mathbf{x}_{n+1} \rangle = \alpha_n \alpha_{n+1} \langle \nabla f(\mathbf{x}_n), \nabla f(\mathbf{x}_{n+1}) \rangle,$$

and, since α_n optimises the objective function $\phi_n(\alpha) = f(\mathbf{x}_n - \alpha\nabla f(\mathbf{x}_n))$, by the FONC

$$0 = \frac{d}{d\alpha}\phi_n(\alpha_n) = \nabla f(\mathbf{x}_n - \alpha_n\nabla f(\mathbf{x}_n))^T(-\nabla f(\mathbf{x}_n)) = -\langle \nabla f(\mathbf{x}_{n+1}), \nabla f(\mathbf{x}_n)\rangle,$$

so $\nabla f(\mathbf{x}_{n+1})$ and $\nabla f(\mathbf{x}_n)$ are orthogonal.

It is clear from (3.5) that, when $\nabla f(\mathbf{x}_n) \neq 0$, there is an $\bar{\alpha}$ such that $\phi_n(\alpha) < \phi_n(0)$ for all $\alpha \in (0, \bar{\alpha}]$. As α_n is the minimiser, the method possesses the **descent property**:

$$f(\mathbf{x}_{n+1}) = \phi_n(\alpha_n) \leqslant \phi_n(\bar{\alpha}) < \phi_n(0) = f(\mathbf{x}_n).$$

When implementing the algorithm, the stopping condition is usually chosen when $\|\nabla f(\mathbf{x}_n)\|$ reaches the desired level of tolerance; i.e., $\|\nabla f(\mathbf{x}_n)\| < \epsilon$ for some pre-specified threshold ϵ. Other possible stopping conditions are:

$$\begin{array}{ccc} |f(\mathbf{x}_{n+1}) - f(\mathbf{x}_n)| < \epsilon, & \frac{|f(\mathbf{x}_{n+1})-f(\mathbf{x}_n)|}{|f(\mathbf{x}_n)|} < \epsilon, & \frac{|f(\mathbf{x}_{n+1})-f(\mathbf{x}_n)|}{\max(1,|f(\mathbf{x}_n)|)} < \epsilon, \\ \|\mathbf{x}_{n+1} - \mathbf{x}_n\| < \epsilon, & \frac{\|\mathbf{x}_{n+1}-\mathbf{x}_n\|}{\|\mathbf{x}_n\|} < \epsilon, & \frac{\|\mathbf{x}_{n+1}-\mathbf{x}_n\|}{\max(1,\|\mathbf{x}_n\|)} < \epsilon. \end{array}$$

We say that an iterative algorithm is **globally convergent** if, for any arbitrary starting point, the algorithm is guaranteed to generate a sequence of points converging to a point that satisfies the FONC for a minimizer. When the algorithm is not globally convergent, it may still generate a sequence that converges to a point satisfying the FONC, provided the initial point is sufficiently close to the point. In this case, we say that the algorithm is **locally convergent**. How close to a solution point we need to start for the algorithm to converge depends on the local convergence properties of the algorithm. A related issue of interest pertaining to a convergent algorithm is the **rate of convergence**; that is, how fast the algorithm converges to a solution point.

Chong and Żak show that for a quadratic function of the form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} - \mathbf{b}^T\mathbf{x},$$

where $\mathbf{Q} \in \mathbb{R}^{n\times n}$ is a symmetric positive definite matrix, and $\mathbf{b}, \mathbf{x} \in \mathbb{R}^n$, the method of steepest descent takes the form

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\mathbf{g}^T\mathbf{g}}{\mathbf{g}^T\mathbf{Q}\mathbf{g}}\mathbf{g}, \qquad \mathbf{g} = \nabla f(\mathbf{x}_n),$$

and, that it is globally convergent, see [3, Theorem 8.2]. To find the unique minimizer exactly one would have to solve the FONC, $\nabla f(\mathbf{x}) = \mathbf{Q}\mathbf{x} - \mathbf{b} = 0$; i.e. invert an $\mathbb{R}^{n\times n}$ matrix. The fixed-step-size gradient algorithm (3.4), with constant $\alpha_n = \alpha$, is globally convergent if and only if

$$0 < \alpha < \frac{2}{\lambda_{\max}(\mathbf{Q})},$$

where $\lambda_{\max}$ is the greatest eigenvalue of $\mathbf{Q}$, see [3, Theorem 8.3].

3.6 The downhill simplex method

This method is often called the Nelder-Mead method after John Nelder and Roger Mead. It is simple to describe, and it is often the choice of something quick and dirty that will work. It is a method that does not require any derivatives, only function evaluations.

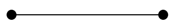
An ***n*-simplex** is a set of $n+1$ points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^n$ such that

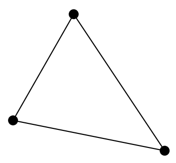
$$\det \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \ldots & \mathbf{x}_n \\ 1 & 1 & \ldots & 1 \end{pmatrix} \neq 0.$$

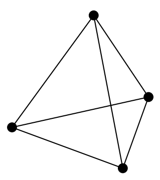
The latter property is to ensure that two points in $\mathbb{R}$ are different, two points in $\mathbb{R}^2$ are not collinear, three points in $\mathbb{R}^3$ are not coplanar, and so on. Each point in an n-simplex is called a **vertex**.

A simplex is one way to generalise a triangle to arbitrarily many dimensions:

- A 1-simplex is a set of 2 points in $\mathbb{R}$, which bound a line segment.
- A 2-simplex is a set of 3 points in $\mathbb{R}^2$, which form the vertices of a triangle.
- A 3-simplex is a set of 4 points in $\mathbb{R}^3$, which form the vertices of a tetrahedron.


a line segment,
a 1-simplex in $\mathbb{R}$


a triangle,
a 2-simplex in $\mathbb{R}^2$


a tetrahedron,
a 3-simplex in $\mathbb{R}^3$

Given an objective function $f\colon \mathbb{R}^n \to \mathbb{R}$, we take an initial n-simplex, which may be chosen randomly. Then,

1. Eliminate the worst vertex (i.e., the vertex with the largest function evaluation).
2. Apply some transformations to the worst vertex to obtain a set of candidates for a new vertex.
3. If the candidates are better than the worst vertex, then replace the worst vertex with the best one of these candidates (i.e., the vertex with the smallest function evaluation).

The candidates, and the parameters, are as follows:

1. Reflection and multiplication by a factor α.
2. Expansion by a factor β.
3. Contraction by a factor γ.
4. Reduction by a factor δ.

The method begins by sorting the vertices so that

$$f(\mathbf{x}_1) \leqslant f(\mathbf{x}_2) \leqslant \ldots \leqslant f(\mathbf{x}_{n+1}).$$

Note that $\mathbf{x}_{n+1}$ is the "worst" vertex. The transformations are applied relative to the **centre of mass** of all points except the worst:

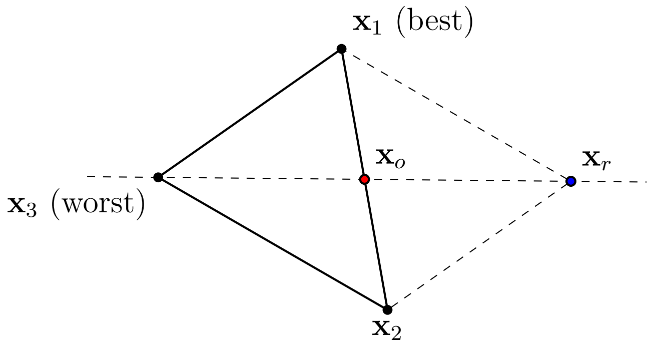
$$\mathbf{x}_o = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i.$$

We now consider each transformation in turn, demonstrating them when applied to a 2-simplex.

Since $\mathbf{x}_{n+1}$ yields the largest evaluation of f at the vertices, we might expect a lower value of f to be found opposite the point $\mathbf{x}_{n+1}$. In the **reflection** step, the point $\mathbf{x}_{n+1}$ is reflected relative to $\mathbf{x}_o$, and multiplied by a factor α, giving

$$\mathbf{x}_r = \mathbf{x}_o + \alpha(\mathbf{x}_o - \mathbf{x}_{n+1}).$$

For a 2-simplex, the reflection might look like this:



There are three possibilities here:

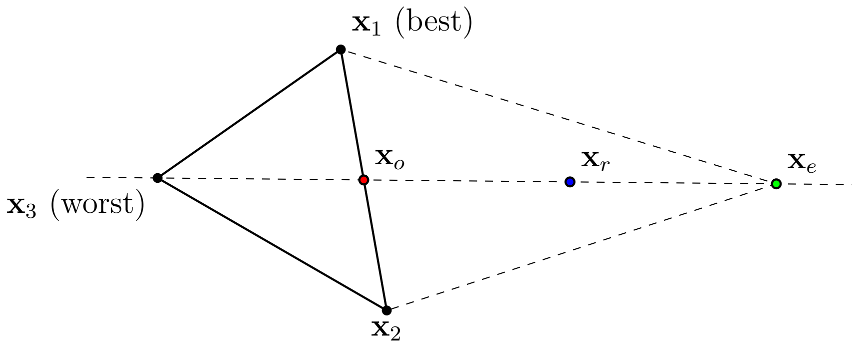
1. $f(\mathbf{x}_1) \leqslant f(\mathbf{x}_r) < f(\mathbf{x}_n)$, in which case $\mathbf{x}_r$ improves the simplex.
2. $f(\mathbf{x}_r) < f(\mathbf{x}_1)$, which would make $\mathbf{x}_r$ the best choice so far.
3. $f(\mathbf{x}_r) \geqslant f(\mathbf{x}_n)$, which would be problematic: if $\mathbf{x}_{n+1}$ were replaced with $\mathbf{x}_r$, then $\mathbf{x}_r$ would be the worst vertex in the simplex; the reflection at the next iteration will then be back in the direction of $\mathbf{x}_{n+1}$, which is already established as a poor choice.

In the first case, we replace $\mathbf{x}_{n+1}$ with $\mathbf{x}_r$ and then proceed to the next iteration.

In the second case, given that $\mathbf{x}_r$ is the best point so far, it seems reasonable that there might be a better choice to be found by moving in the same direction. The **expansion** step expands out from $\mathbf{x}_r$ by a factor of γ, giving

$$\mathbf{x}_e = \mathbf{x}_o + \gamma(\mathbf{x}_r - \mathbf{x}_o).$$

For a 2-simplex, the expansion might look like this:



The expansion will be considered successful if $f(\mathbf{x}_e)$ is smaller than $f(\mathbf{x}_r)$, in which case $\mathbf{x}_{n+1}$ is replaced with $\mathbf{x}_e$. Otherwise, $\mathbf{x}_{n+1}$ is replaced with $\mathbf{x}_r$.

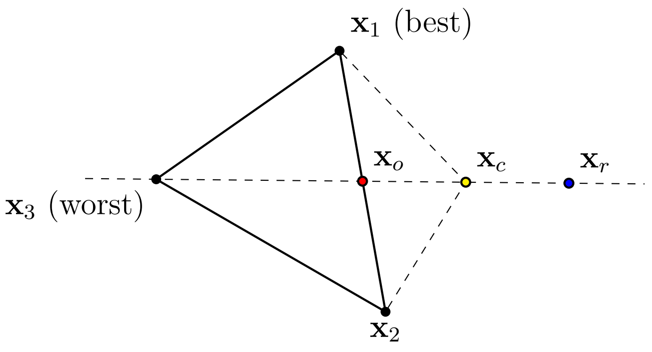
In the third case, the reflected point is a bad option, suggesting that the search should remain closer to the simplex instead. The **contraction step** is based on two possibilities:

1. $f(\mathbf{x}_n) \leqslant f(\mathbf{x}_r) < f(\mathbf{x}_{n+1})$, which means that $\mathbf{x}_r$ is better than the worst point, but still a bad choice; instead, we find a point on the line segment joining $\mathbf{x}_o$ and $\mathbf{x}_r$.
2. $f(\mathbf{x}_r) \geqslant f(\mathbf{x}_{n+1})$, which means that $\mathbf{x}_r$ is worse than the worst point; this is a poor choice of direction to move in, so instead we construct a new point inside the simplex.

In the first case, we perform an **outside contraction** by a factor of ρ, given by

$$\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_r - \mathbf{x}_o).$$

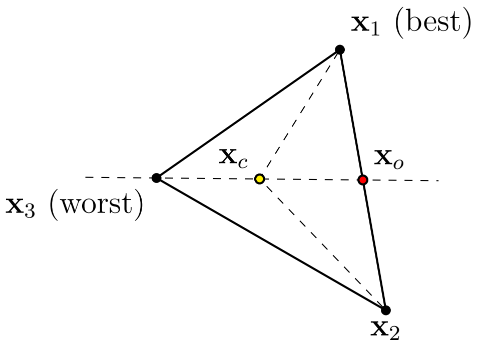
For a 2-simplex, an outside contraction might look like this:



In the second case, we perform an **inside contraction** by a factor of ρ, given by

$$\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_{n+1} - \mathbf{x}_o).$$

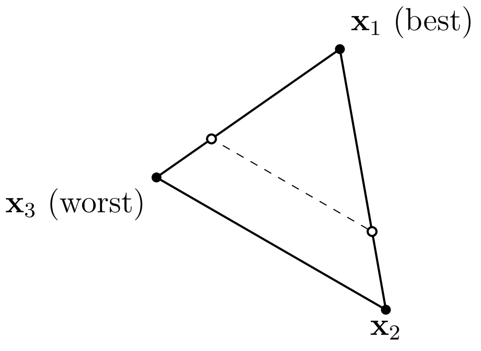
For a 2-simplex, an inside contraction might look like this:



And finally, if none of the previous candidates lead to an improvement, we **shrink** by moving all points closer to the best point:

$$\mathbf{x}_i \to \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1), \quad i \neq 1.$$

For a 2-simplex, the shrink step might look like this:



This gives us the **downhill simplex method**, as follows:

1. Order the vertices so that
$$f(\mathbf{x}_1) \leqslant f(\mathbf{x}_2) \leqslant \ldots \leqslant f(\mathbf{x}_{n+1}).$$
2. Calculate the center of mass of all points except $\mathbf{x}_{n+1}$,
$$\mathbf{x}_o = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i.$$

3. Calculate the reflection of $\mathbf{x}_{n+1}$ through the center of mass,
$$\mathbf{x}_r = \mathbf{x}_o + \alpha(\mathbf{x}_o - \mathbf{x}_{n+1}).$$
4. If $f(\mathbf{x}_1) < f(\mathbf{x}_r) < f(\mathbf{x}_n)$, then replace $\mathbf{x}_{n+1}$ with $\mathbf{x}_r$, and return to step 1.
5. If $f(\mathbf{x}_r) < f(\mathbf{x}_1)$, then calculate the expansion of $\mathbf{x}_{n+1}$,
$$\mathbf{x}_e = \mathbf{x}_o + \gamma(\mathbf{x}_r - \mathbf{x}_o),$$
and if $f(\mathbf{x}_e) < f(\mathbf{x}_r)$, then replace $\mathbf{x}_{n+1}$ with $\mathbf{x}_e$; otherwise, replace $\mathbf{x}_{n+1}$ with $\mathbf{x}_r$ and return to step 1.
6. If $f(\mathbf{x}_r) < f(\mathbf{x}_{n+1})$, calculate the outside contraction,
$$\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_r - \mathbf{x}_o).$$
7. If $f(\mathbf{x}_r) \geqslant f(\mathbf{x}_{n+1})$, calculate the inside contraction,
$$\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_{n+1} - \mathbf{x}_o).$$
8. If $f(\mathbf{x}_c) < f(\mathbf{x}_{n+1})$, then replace $\mathbf{x}_{n+1}$ with $\mathbf{x}_c$ and return to step 1.
9. Move all points except $\mathbf{x}_1$ towards $\mathbf{x}_1$,
$$\mathbf{x}_i \to \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1), \quad i \neq 1,$$
and then return to step 1.

Typical values of the parameters are $\alpha = 1$, $\gamma = 2$, $\rho = \sigma = 1/2$.

3.7 Regression revisited

We will finish this chapter with a demonstration of the downhill simplex method applied to a nonlinear regression problem. Recall that, in Section 2.4, we attempted to fit three data points, $(1, 4)$, $(2, 5)$ and $(3, 6)$, to a function $f(x) = ae^{bx}$. To do this, the objective function we wish to minimise is the sum of squared residuals,

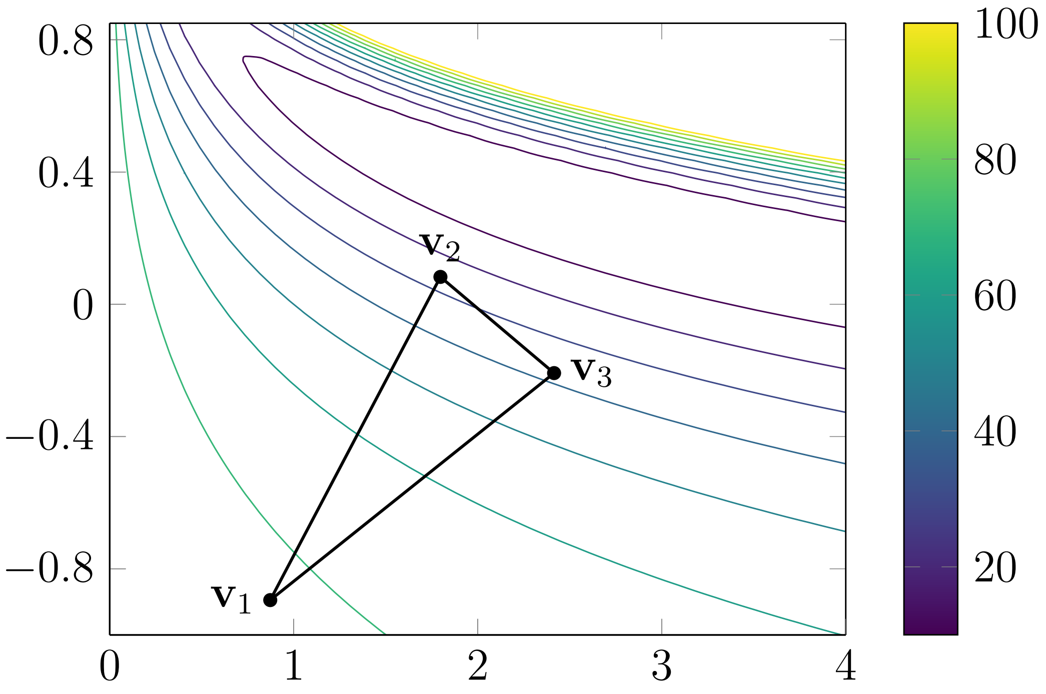
$$\begin{aligned} F(a, b) &= (4 - f(1))^2 + (5 - f(2))^2 + (6 - f(3))^2 \\ &= (4 - ae^b)^2 + (5 - ae^{2b})^2 + (6 - ae^{3b})^2. \end{aligned}$$

It was shown that a direct application of the FONC will fail, as it results in a pair of equations that may not even be possible to solve. So, an iterative method is suitable.

We choose to apply the downhill simplex method with parameters $\alpha = 1$, $\gamma = 2$, $\rho = 1/2$ and $\sigma = 1/2$. Since this is a function of 2 variables, we will need to start with a 2-simplex, which contains 3 vertices. For illustrative purposes, the following initial vertices were randomly generated:

$$\mathbf{v}_1 = (0.8725, -0.8946), \quad \mathbf{v}_2 = (1.7973, 0.0828), \quad \mathbf{v}_3 = (2.4150, -0.2081).$$

Some level sets of F are shown below, with the initial simplex overlain.



Intuitively, the vertex $\mathbf{v}_1$ is quite high up, and the vertices $\mathbf{v}_2$ and $\mathbf{v}_3$ are relatively low; we expect the simplex to move away from $\mathbf{v}_1$ and towards $\mathbf{v}_2$ and $\mathbf{v}_3$. The algorithm will start by evaluating the objective function at each of the vertices:

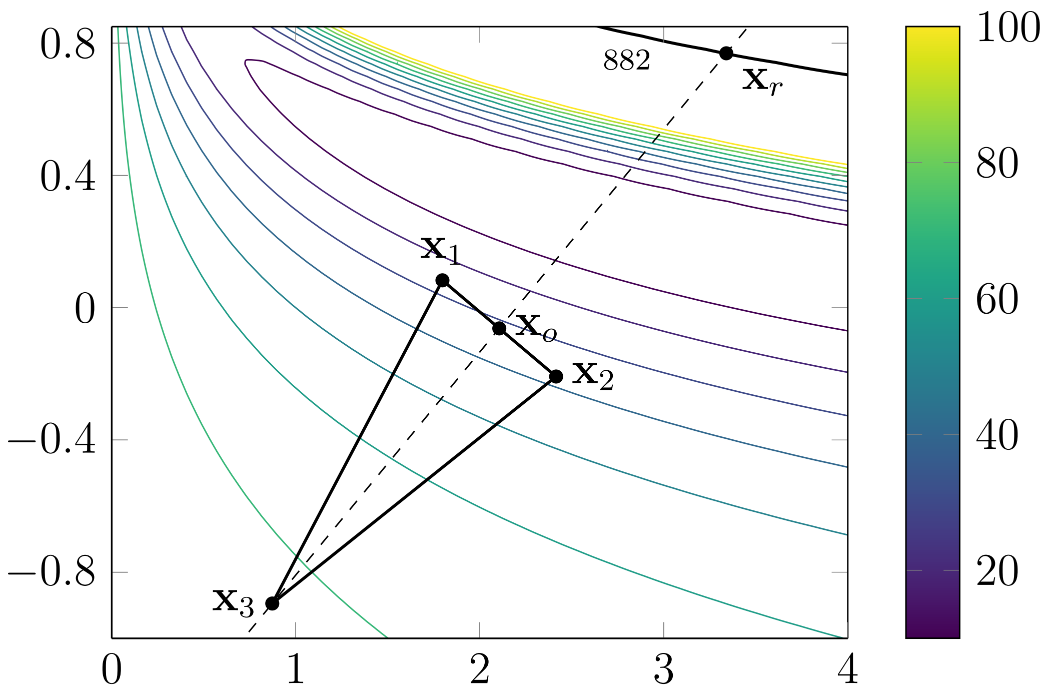
a	b	$F(a, b)$
0.8725	−0.8946	72.1258
1.7973	0.0828	26.1409
2.4150	−0.2081	37.9159

So, arranging the vertices from "best" to "worst", we have

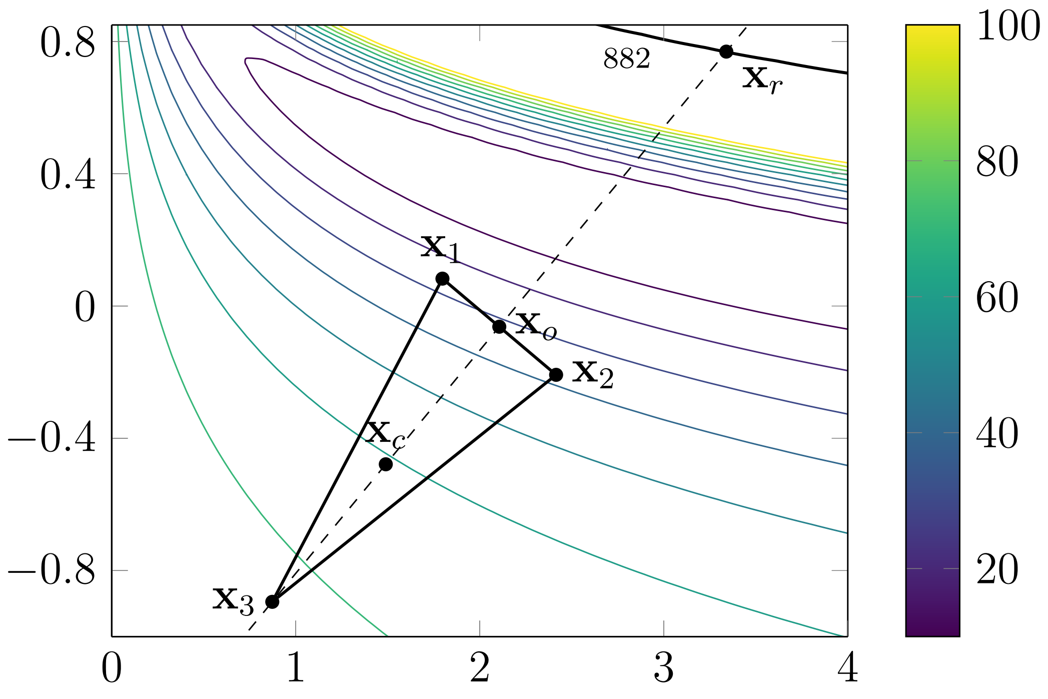
$$\begin{aligned} \mathbf{x}_1 &= (1.7973, 0.0828), & F(\mathbf{x}_1) &= 26.1409, \\ \mathbf{x}_2 &= (2.4150, -0.2081), & F(\mathbf{x}_2) &= 37.9159, \\ \mathbf{x}_3 &= (0.8725, -0.8946), & F(\mathbf{x}_3) &= 72.1258. \end{aligned}$$

The centre of mass is $\mathbf{x}_o = \frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_2) = (2.1062, -0.0626)$, giving $\mathbf{x}_r = 2\mathbf{x}_0 - \mathbf{x}_3 = (3.3399, 0.7694)$ and $f(\mathbf{x}_r) = 882.8699$.

The reflection of $\mathbf{x}_3$ at this iteration is depicted below.



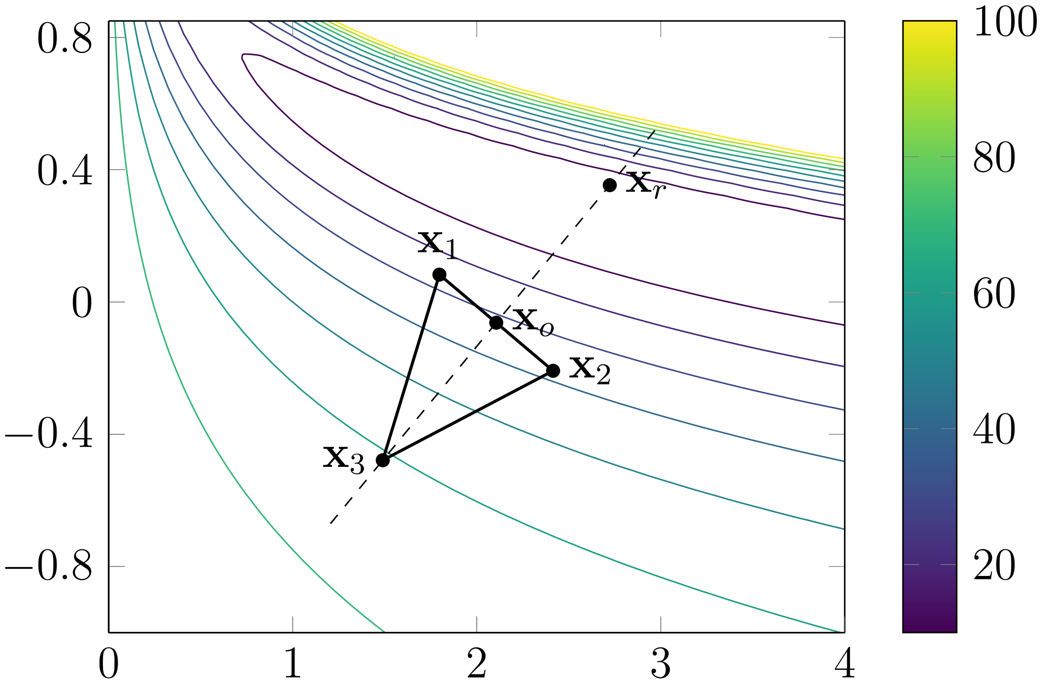
Since $F(\mathbf{x}_r) = 882.8699$ is larger than all of the previous function evaluations, the reflection fails and we proceed to the contraction step. Since $F(\mathbf{x}_r) \geqslant F(\mathbf{x}_3)$, an inside contraction is performed.



We find $\mathbf{x}_c = (1.4894, -0.4786)$ and $F(\mathbf{x}_c) = 60.9499 < F(\mathbf{x}_3)$. So the contraction step is successful, and $\mathbf{x}_3$ is replaced with $\mathbf{x}_c$. The next iteration will have

$$\begin{aligned} \mathbf{x}_1 &= (1.7973, 0.0828), & F(\mathbf{x}_1) &= 26.1409, \\ \mathbf{x}_2 &= (2.4150, -0.2081), & F(\mathbf{x}_2) &= 37.9159, \\ \mathbf{x}_3 &= (1.4894, -0.4786), & F(\mathbf{x}_3) &= 60.9499. \end{aligned}$$

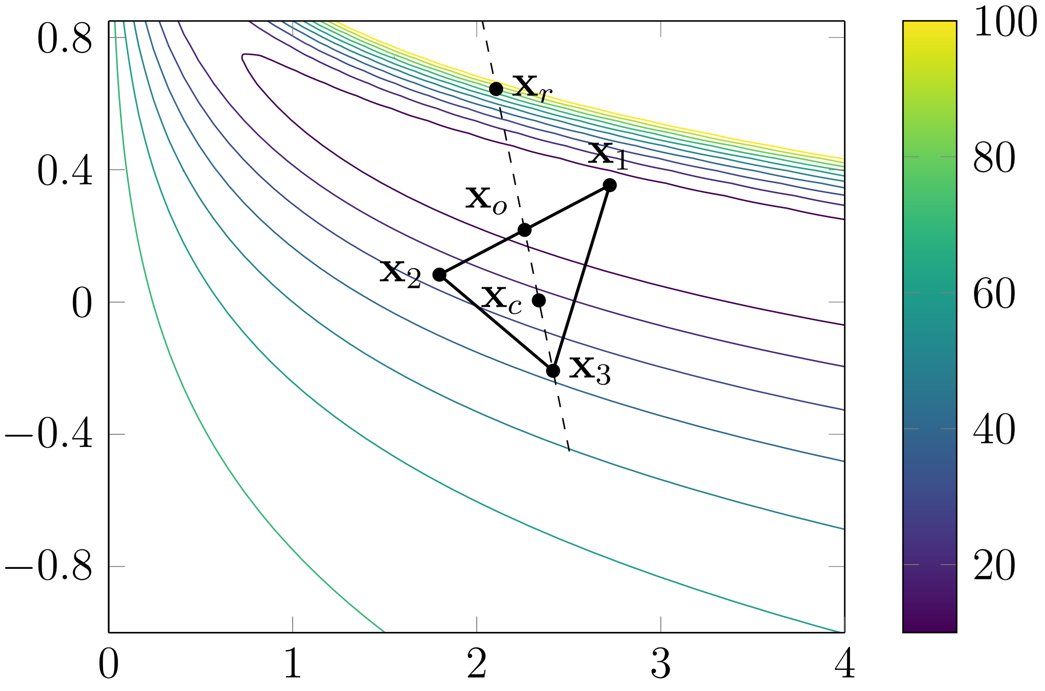
The reflection of $\mathbf{x}_3$ at this iteration is depicted below.



Here we find $\mathbf{x}_r = (3.3399, 2.723)$ and $F(\mathbf{x}_r) = 3.7503$. Since $F(\mathbf{x}_r) < F(\mathbf{x}_1)$, an expansion will be attempted. Inspecting the plot reveals that an expansion moves in the upward direction away from $\mathbf{x}_r$, so that will be unsuccessful. Hence, $\mathbf{x}_3$ is replaced with $\mathbf{x}_r$, and the new simplex has

$$\begin{aligned}
\mathbf{x}_1 &= (2.723, 0.3534), & F(\mathbf{x}_1) &= 3.7503,\\
\mathbf{x}_2 &= (1.7973, 0.0828), & F(\mathbf{x}_2) &= 26.1409,\\
\mathbf{x}_3 &= (2.4150, -0.2081), & F(\mathbf{x}_3) &= 37.9159.
\end{aligned}$$

For the next iteration, the plot below shows that a reflection of $\mathbf{x}_3$ will be unsucessful, but an inside contraction will succeed.

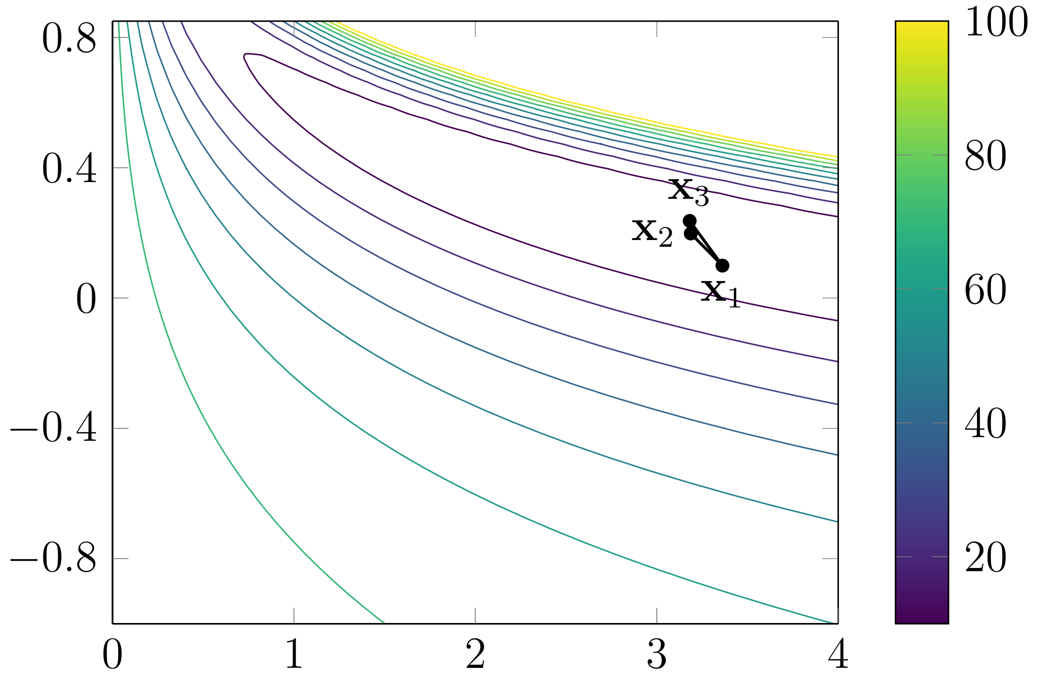


Observe that the simplex is gradually moving down the level sets—hence the name “downhill simplex method”.

After 10 total iterations, the simplex has

$$\begin{aligned}
\mathbf{x}_1 &= (3.3619, 0.0995), \quad F(\mathbf{x}_1) = 0.0995,\\
\mathbf{x}_2 &= (3.1874, 0.1981), \quad F(\mathbf{x}_2) = 0.1331,\\
\mathbf{x}_3 &= (3.1813, 0.2368), \quad F(\mathbf{x}_3) = 0.2373.
\end{aligned}$$

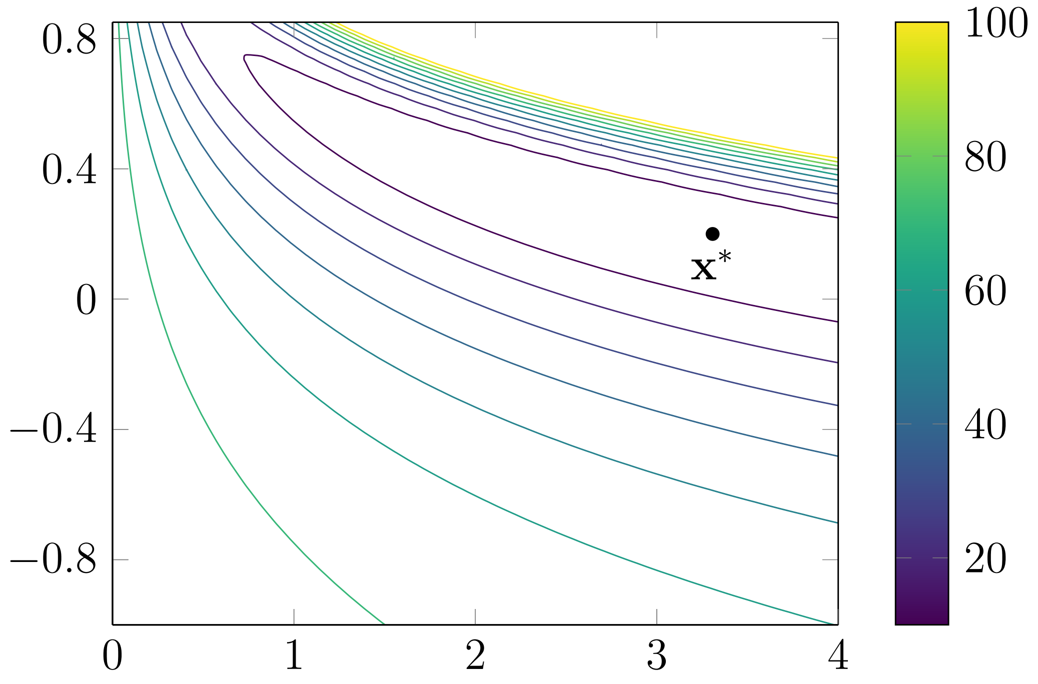
This simplex is depicted below.



After 100 total iterations, the simplex has

$$\begin{aligned}
\mathbf{x}_1 &= (3.3078, 0.2000), \quad F(\mathbf{x}_1) = 0.0066,\\
\mathbf{x}_2 &= (3.3078, 0.2000), \quad F(\mathbf{x}_2) = 0.0066,\\
\mathbf{x}_3 &= (3.3078, 0.2000), \quad F(\mathbf{x}_3) = 0.0066.
\end{aligned}$$

The three vertices are indistinguishable to four decimal places, so we can deduce that, to four decimal places, the local minimiser is $\mathbf{x}^* = (3.3087, 0.2000)$.



Rather than showing more steps in detail, this chapter ends with an implementation of the downhill simplex method in MATLAB. We represent a simplex by a matrix whose columns correspond to its vertices, so the initial simplex described above is defined as follows:

```
simplex = [[0.8725; -0.8946] [1.7973; 0.0828] [2.4150; -0.2081]];
```

You could instead start at a random simplex using, say,

```
simplex = rand(2,3);
```

The objective function is defined in `objective.m`, in a manner that permits columnwise evaluation of the simplex:

```
function z = objective(vars)
a = vars(1,:);
b = vars(2,:);
z = (4-a.*exp(b)).^2 + (5-a.*exp(2*b)).^2 + (6-a.*exp(3*b)).^2;
end
```

On the next page, a function is defined (that should be placed in a file `iterateSimplex.m`), which implements a single iteration of the downhill simplex method. Then, the following code can be used to perform 25 iterations of the downhill simplex method.

```
simplex = [[0.8725; -0.8946] [1.7973; 0.0828] [2.4150; -0.2081]]
for i = 1:25
    simplex = iterateSimplex(simplex);
end
simplex
```

This gives an output of

```
simplex =
    3.3100    3.3127    3.3062
    0.1995    0.1993    0.2001
```

To approximate the minimiser, we could take, say, the centre of mass of the vertices:

```
mean(simplex, 2);
```

In this case, we get approximately $(3.3096, 0.1997)$.

Alternatively, we might choose to implement a tolerance level. For example, iterating until the length of the vector between successive centres of mass is small:

```
simplex = [0.8725 1.7973 2.4150; -0.8946 0.0828 -0.2081];
previous = Inf;
current = mean(simplex, 2);
count = 0;
while norm(previous-current) > 10^-5
    simplex = iterateSimplex(simplex);
    previous = current;
    current = mean(simplex,2);
    count = count+1;
end
com = mean(simplex,2);
fprintf("After %d steps, the minimiser is [%f,%f].\n", count, com)
```

This gives

```
After 39 steps, the minimiser is [3.307772,0.199983].
```

The iterateSimplex function is as follows:

```
function simplex = iterateSimplex(simplex)
[alpha, gamma, rho, sigma] = deal(1,2,1/2,1/2); % parameters
[n,~] = size(simplex); % dimension

% Sort the vertices from best to worst
[evals, indices] = sort(objective(simplex), 2);
simplex = simplex(:, indices);

% Evaluate some values for later use
x1 = simplex(:,1);
fx1 = evals(1);
fxn = evals(n);
worst = simplex(:,n+1);
fworst = evals(n+1);

% Construct the centre of mass; construct and check the reflection
xo = mean(simplex(:, 1:n), 2);
xr = xo + alpha*(xo - worst);
fxr = objective(xr);
if fx1 < fxr && fxr < fxn
    simplex(:,n+1) = xr;
    return;
end
if fxr < fx1
    % Attempt an expansion
    xe = xo + gamma*(xr - xo);
    fxe = objective(xe);
    if fxe < fxr
        simplex(:,n+1) = xe;
    else
        simplex(:,n+1) = xr;
    end
    return;
end

% Calculate and check the appropriate contraction
if fxr < fworst
    xc = xo + rho*(xr - xo);
else
    xc = xo + rho*(worst - xo);
end
fxc = objective(xc);
if fxc < fworst
    simplex(:,n+1) = xc;
    return;
end

% Shrink
simplex(:,2:end) = x1 + sigma*(simplex(:,2:end) - x1);
end
```