# Mini-HOWTO on using Octave for Unconstrained Nonlinear Optimization*

Nonlinear optimization problems are very common and when a solution cannot be found analytically, one usually tries to find it numerically. This document shows how to perform unconstrained nonlinear minimization using the Octave language for numerical computation. We assume to be so lucky as to have an initial guess from which to start an iterative method, and so impatient as to avoid as much as possible going into the details of the algorithm. In the following examples, we consider multivariable problems, but the single variable case is solved in exactly the same way.

All the algorithms used below return numerical approximations of *local minima* of the optimized function. In the following examples, we minimize a function with a single minimum (Figure 1), which is relatively easily found. In practice, success of optimization algorithms greatly depend on the optimized function and on the starting point.

## A simple example

We will use a call of the type

```
[x_best, best_value, niter] = minimize (func, x_init)
```

to find the minimum of

$$f : \quad (x_1, .x_2, x_3) \in \mathbb{R}^3 \quad \longrightarrow \quad (x_1 - 1)^2 / 9 + (x_3 - 1)^2 / 9 + (x_3 - 1)^2 / 9$$
$$- \cos(x_1 - 1) - \cos(x_2 - 1) - \cos(x_3 - 1).$$

The following commands should find a local minimum of $f()$, using the Nelder-Mead (aka "downhill simplex") algorithm and starting from a randomly chosen point x0 :
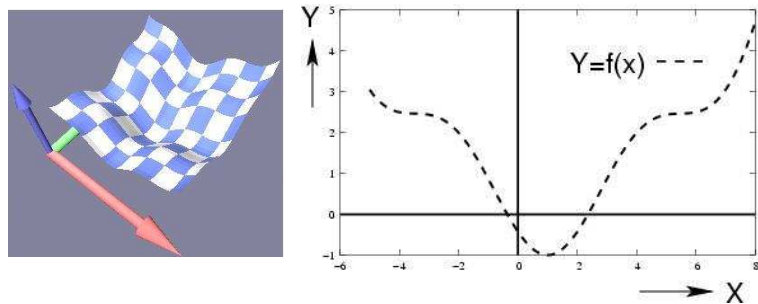
Figure 1: 2D and 1D slices of the function that is minimized throughout this tutorial. Although not obvious at first sight, it has a unique minimum.

```
function cost = foo (xx)
  xx--;
  cost = sum (-cos(xx)+xx.^2/9);
endfunction
x0 = [-1, 3, -2];
[x,v,n] = minimize ("foo", x0)
```

The output should look like :

```
x =
  1.00000 1.00000 1.00000
v = -3.0000
n = 248
```

This means that a minimum has been found in $(1, 1, 1)$ and that the value at that point is $-3$. This is correct, since all the points of the form $x_1 = 1 + 2i\pi$, $x_2 = 1 + 2j\pi$, $x_3 = 1 + 2k\pi$, for some $i, j, k \in \mathbb{N}$, minimize $f()$. The number of function evaluations, 248, is also returned. Note that this number depends on the starting point. You will most likely obtain different numbers if you change x0.

The Nelder-Mead algorithm is quite robust, but unfortunately it is not very efficient. For high-dimensional problems, its execution time may become prohibitive.

## Using the first differential

Fortunately, when a function, like $f()$ above, is differentiable, more efficient optimization algorithms can be used. If minimize() is given the differential of the optimized function, using the "df" option, it will use a conjugate gradient method.

```
## Function returning partial derivatives
function dc = diffoo (x)
    x = x(:)' - 1;
    dc = sin (x) + 2*x/9;
```

2

```
        endfunction
        [x, v, n] = minimize ("foo", x0, "df", "diffoo")
```

This produces the output :

```
    x =
       1.00000 1.00000 1.00000
    v = -3
    n =
       108 6
```

The same minimum has been found, but only 108 function evaluations were needed, together with 6 evaluations of the differential. Here, `diffoo()` takes the same argument as `foo()` and returns the partial derivatives of $f()$ with respect to the corresponding variables. It doesn't matter if it returns a row or column vector or a matrix, as long as the $i^{\text{th}}$ element of `diffoo(x)` is the partial derivative of $f()$ with respect to $x_i$ .

## Using numerical approximations of the first differential

Sometimes, the minimized function is differentiable, but actually writing down its differential is more work than one would like. Numerical differentiation offers a solution which is less efficient in terms of computation cost, but easy to implement. The "ndiff" option of `minimize()` uses numerical differentiation to execute exactly the same algorithm as in the previous example. However, because numerical approximation of the differentia is used, the outpud may differ slightly :

```
        [x, v, n] = minimize ("foo", x0, "ndiff")
```

wich yields :

```
    x =
       1.00000 1.00000 1.00000
    v = -3
    n =
       78 6
```

Note that each time the differential is numerically approximated, `foo()` is called 6 times (twice per input element), so that `foo()` is evaluated a total of (78+6*6=) 114 times in this example.

## Using the first and second differentials

When the function is twice differentiable and one knows how to compute its first and second differentials, still more efficient algorithms can be used (in our case, a

variant of Levenberg-Marquardt). The option "d2f" allows to specify a function that returns the value of the function, the first and second differentials of the minimized function. Entering the commands :

```
function [c, dc, d2c] = d2foo (x)
    c = foo(x);
    dc = diffoo(x);
    d2c = diag (cos (x(:)-1) + 2/9);
end
[x,v,n] = minimize ("foo", x0, "d2f", "d2foo")
```

produces the output :

```
x =
  1.0000 1.0000 1.0000
v = -3
n =
  34 5
```

This time, 34 function evaluations, and 5 evaluations of d2foo() were needed.

## Summary

We have just seen the most basic ways of solving nonlinear unconstrained optimization problems. The online help system of Octave (try e.g. "help minimize") will yield information on other issues, such as *passing extra arguments* to the minimized function, *controling the termination* of the optimization process, choosing the algorithm etc.