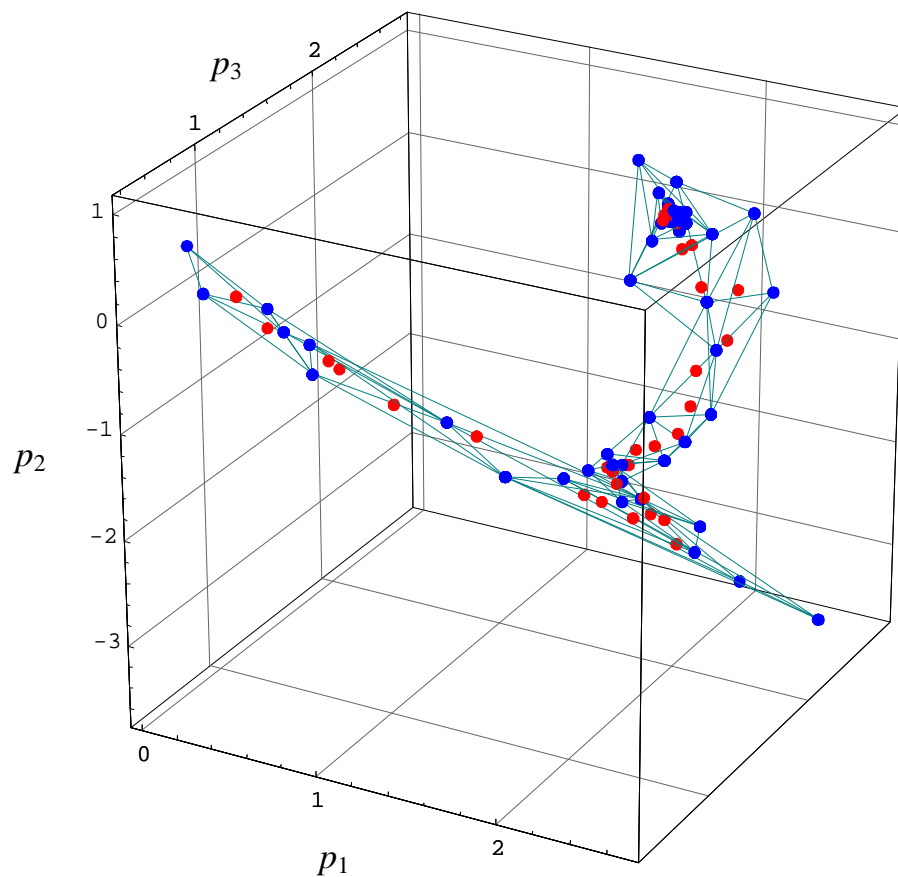


# *Simplex algorithms for nonlinear constraint optimization problems*




Technical report  
Revision 0

*By Igor Grešovnik  
Ljubljana, 2009*



Implementation remarks in this document refer to *IoptLib* (*Investigative Optimization Library*).

**Contents:**

<b>1</b>	<b>Introduction</b> .....	<b>1</b>
<b>2</b>	<b>Nelder-Mead Simplex Method for Unconstrained Minimization</b> .....	<b>1</b>
<b>3</b>	<b>Simplex Methods for Inequality Constraints</b> .....	<b>6</b>
<b>3.1</b>	<b>Adaptation by comparison relations</b> .....	<b>7</b>
3.1.1	Different definitions of comparison functions .....	8
<b>3.2</b>	<b>Addition of penalty terms to the objective function</b> .....	<b>11</b>
3.2.1	Adding discontinuous jump to the objective function on the level set.....	11
3.2.2	Exact penalty functions.....	12
3.2.3	Adaptive penalty algorithm.....	23
<b>3.3</b>	<b>Strict consideration of bound constraints</b> .....	<b>24</b>
<b>3.4</b>	<b>Implementation remarks on penalty terms and bound constraints</b>  .....	<b>26</b>
3.4.1	Basic tools for handling bound constraints .....	28
3.4.2	Penalty generating functions.....	28
3.4.3	Conversion of bound constraints to ordinary constraints .....	28
<b>4</b>	<b>Simplex methods for equality constraints</b>  .....	<b>28</b>
<b>5</b>	<b>Accelerating convergence with response approximation</b>  .....	<b>28</b>
<b>6</b>	<b>Appendix</b> .....	<b>29</b>
6.1.1	Relations .....	30
<b>7</b>	<b>Sandbox (this is not part of this report)</b> .....	<b>1</b>

## 1 INTRODUCTION

This document describes variants of the modified Nelder-Mead simplex method adapted for solution of constrained non-linear optimization problems.

We will consider the nonlinear optimization problems of the form

$$\begin{aligned} \text{minimise} & & f(\mathbf{x}), & \quad \mathbf{x} \in \mathbb{R}^n \\ \text{subject to} & & c_i(\mathbf{x}) \leq 0, & \quad i \in I \\ \text{and} & & c_j(\mathbf{x}) = 0, & \quad j \in E, \\ \text{where} & & l_k \leq x_k \leq u_k, & \quad k = 1, 2, \dots, n. \end{aligned} \tag{1}$$

Function  $f$  is called the *objective* function,  $c_i$  and  $c_j$  are called constraint functions and  $l_k$  and  $u_k$  are called upper and lower bounds. The second and third line of the equation are referred to as inequality and equality constraints, respectively (with  $I$  and  $E$  being the corresponding inequality and equality index sets). We will collectively refer to  $f$ ,  $c_i, i \in I$  and  $c_j, j \in E$  as constraint functions.  $l_k$  and  $u_k$  are the lower and upper bounds for the optimization variable.

## 2 NELDER-MEAD SIMPLEX METHOD FOR UNCONSTRAINED MINIMIZATION

One minimization method that does not belong within the context of the subsequent text is the simplex method<sup>[2], [3],[6]</sup>. It has been known since the early sixties and could be classed as another heuristic method since it is not based on a substantial theoretical background.

The simplex method neither uses line searches nor is based on minimization of some simplified model of the objective function, and therefore belongs to the class of direct search methods. Because of this the method does not compare well with other described methods with respect to local convergence properties. On the other hand, for the same reason it has some other strong features. The method is relatively insensitive to numerical noise and does not depend on some other properties of the objective function (e.g. convexity) since no specific continuity or other assumptions are incorporated in its design. It merely requires the evaluation of function values. Its performance in practice can be as satisfactory as any other non-derivative method, especially when

---

high accuracy of the solution is not required and the local convergence properties of more sophisticated methods do not play so important role. In many cases it does not make sense to require highly accurate solutions of optimization problems, because the obtained results are inevitably inaccurate with respect to real system behavior due to numerical modeling of the system (e.g. discretization and round-off errors or inaccurate physical models). These are definitely good arguments for considering practical use of the method in spite of the lack of good local convergence results with respect to some other methods.

The simplex method is based on construction of an evolving pattern of  $n+1$  points in  $\mathbb{R}^n$  (vertices of a simplex). The points are systematically moved according to some strategy such that they tend towards the function minimum. Different strategies give rise to different variants of the algorithm. The most commonly used is the Nelder-Mead algorithm described below. The algorithm begins by choice of  $n+1$  vertices of the initial simplex  $(\mathbf{x}_1^{(1)}, \dots, \mathbf{x}_{n+1}^{(1)})$  so that it has non-zero volume. This means that all vectors connecting a chosen vertex to the reminding vertices must be linearly independent, e.g.

$$\exists \lambda_i \neq 0 \Rightarrow \sum_{i=1}^n \lambda_i (\mathbf{x}_{i+1}^{(1)} - \mathbf{x}_1^{(1)}) \neq 0.$$

If we have chosen  $\mathbf{x}_1^{(1)}$ , we can for example obtain other vertices by moving, for some distance, along all coordinate directions. If it is possible to predict several points that should be good according to experience, it might be better to set vertices to these points, but the condition regarding independence must then be checked.

Once the initial simplex is constructed, the function is evaluated at its vertices. Then one or more points of the simplex are moved in each iteration, so that each subsequent simplex consists of a better set of points:

**Algorithm 2.1:** The Nelder-Mead simplex method.

After the initial simplex is chosen, function values in its vertices are evaluated:

$$f_i^{(1)} = f(\mathbf{x}_i^{(1)}), i = 1, \dots, n + 1.$$

Iteration  $k$  is then as follows:

1. **Ordering step:** Simplex vertices are first reordered so that  $f_1^{(k)} \leq f_2^{(k)} \leq \dots \leq f_{n+1}^{(k)}$ , where  $f_i^{(k)} = f(\mathbf{x}_i^{(k)})$ .

2. **Reflection step:** The worst vertex is reflected over the centre point of the best  $n$  vertices

$$(\bar{\mathbf{x}}^{(k)} = \frac{1}{n} \sum_{i=1}^n \mathbf{x}_i^{(k)}), \text{ so that the reflected point } \mathbf{x}_r^{(k)} \text{ is}$$

$$\mathbf{x}_r^{(k)} = \bar{\mathbf{x}}^{(k)} + (\bar{\mathbf{x}}^{(k)} - \mathbf{x}_{n+1}^{(k)})$$

Evaluate  $f_r^{(k)} = f(\mathbf{x}_r^{(k)})$ . If  $f_1^{(k)} \leq f_r^{(k)} < f_n^{(k)}$ , accept the reflected point and go to 6.

3. **Expansion step:** If  $f_r^{(k)} < f_1^{(k)}$ , calculate the expansion

$$\mathbf{x}_e^{(k)} = \bar{\mathbf{x}}^{(k)} + 2(\mathbf{x}_r^{(k)} - \bar{\mathbf{x}}^{(k)})$$

and evaluate  $f_e^{(k)} = f(\mathbf{x}_e^{(k)})$ . If  $f_e^{(k)} < f_r^{(k)}$ , accept  $\mathbf{x}_e^{(k)}$  and go to 6. Otherwise accept  $\mathbf{x}_r^{(k)}$  and go to 6.

4. **Contraction step:** If  $f_r^{(k)} \geq f_n^{(k)}$ , perform contraction between  $\bar{\mathbf{x}}^{(k)}$  and the better of  $\mathbf{x}_{n+1}^{(k)}$  and  $\mathbf{x}_r^{(k)}$ . If  $f_r^{(k)} < f_{n+1}^{(k)}$ , set

$$\mathbf{x}_c^{(k)} = \bar{\mathbf{x}}^{(k)} + \frac{1}{2}(\mathbf{x}_r^{(k)} - \bar{\mathbf{x}}^{(k)})$$

(this is called the outside contraction) and evaluate  $f_c^{(k)} = f(\mathbf{x}_c^{(k)})$ . If  $f_c^{(k)} \leq f_r^{(k)}$ , accept  $\mathbf{x}_c^{(k)}$  and go to 6.

If in contrary  $f_r^{(k)} \geq f_{n+1}^{(k)}$ , set

$$\mathbf{x}_c^{(k)} = \bar{\mathbf{x}}^{(k)} - \frac{1}{2}(\bar{\mathbf{x}}^{(k)} - \mathbf{x}_{n+1}^{(k)})$$

(inside contraction) and evaluate  $f_c^{(k)}$ . If  $f_c^{(k)} < f_{n+1}^{(k)}$ , accept  $\mathbf{x}_c^{(k)}$  and go to 6.

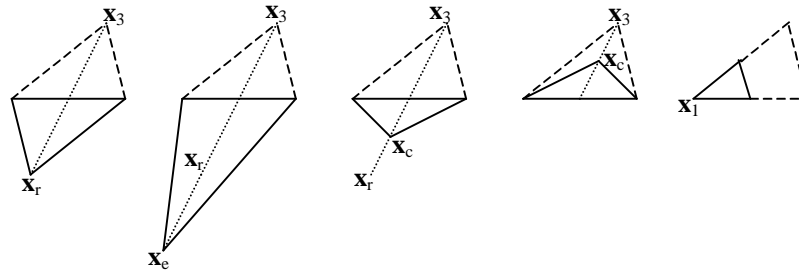
5. **Shrink step:** Move all vertices except the best towards the best vertex, i.e.

$$\mathbf{v}_i^{(k)} = \mathbf{x}_1^{(k)} + \frac{1}{2}(\mathbf{x}_i^{(k)} - \mathbf{x}_1^{(k)}), i = 2, \dots, n + 1,$$

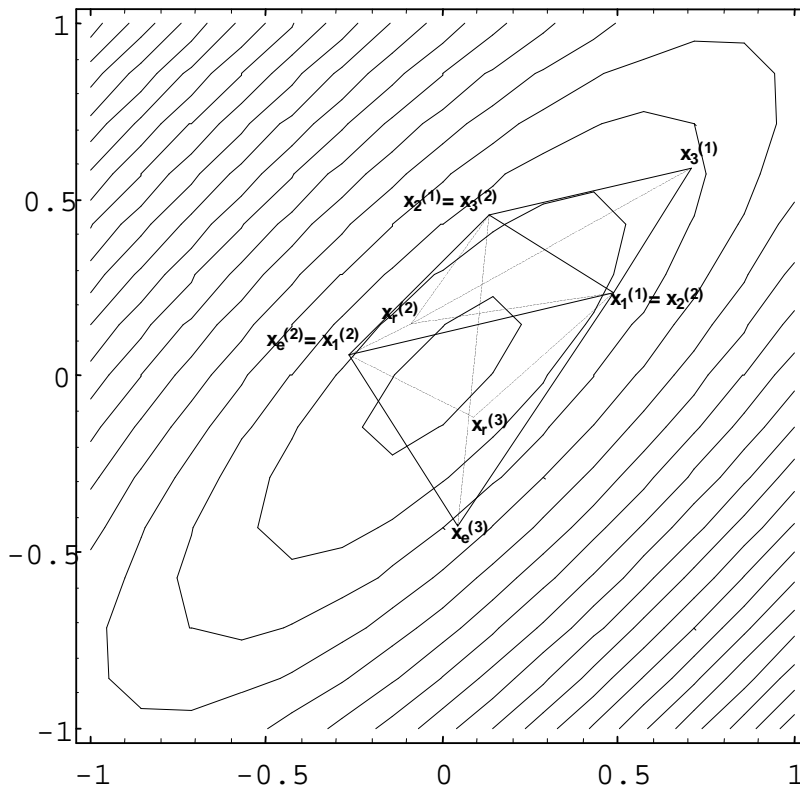
and evaluate  $f_i^{(k)} = f(\mathbf{v}_i^{(k)}), i = 2, \dots, n + 1$ . Accept  $\mathbf{v}_i^{(k)}$  as new vertices.

6. **Convergence check:** Check if the convergence criterion is satisfied. If so, terminate the algorithm, otherwise start the next iteration.

Figure 1 illustrates possible steps of the algorithm. A possible situation of two iterations when the algorithm is applied is shown in Figure 2. The steps allow the shape of the simplex to be changed in every iteration, so the simplex can adapt to the surface of  $f$ . Far from the minimum the expansion step allows the simplex to move rapidly in the descent direction. When the minimum is inside the simplex, contraction and shrink steps allow vertices to be moved closer to it.



**Figure 1:** Possible steps of the simplex algorithm in two dimensions (from left to right): reflection, expansion, outside and inside contraction, and shrink.



**Figure 2:** Example of evolution of the simplex.

There are basically two possibilities for the convergence criterion. Either that function values at vertices must become close enough or the simplex must become small enough. It is usually best to impose both criteria, because either of them alone can be misleading.

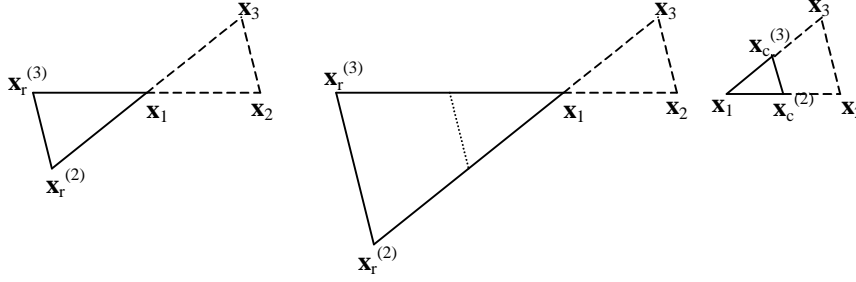
It must be mentioned that convergence to a local minimum has not been proved for the Nelder-Mead algorithm. Examples have been constructed for which the method does not converge for certain choices of the initial guess<sup>[2]</sup>. However, the situations for which this was shown are quite special and unlikely to occur in practice. Another theoretical argument against the algorithm is that it can fail because the simplex collapses into a subspace, so that vectors connecting its vertices become nearly linearly dependent. Investigation of this phenomenon indicates that such behavior is related to cases when the function to be minimized has highly elongated contours (i.e. ill conditioned Hessian). This is also a problematic situation for other algorithms.

The Nelder-Mead algorithm can be easily adapted for constrained optimization. One possibility is to add a special penalty term to the objective function, e.g.

$$f'(\mathbf{x}) = f(\mathbf{x}) + f_{n+1}^{(1)} - \sum_{i \in I} c_i(\mathbf{x}) + \sum_{i \in I} |c_j(\mathbf{x})|, \quad (2)$$

where  $f_{n+1}^{(1)}$  is the highest value of  $f$  in the vertices of the initial simplex. Since subsequent iterates generate simplices with lower values of the function at vertices, the presence of this term guarantees that whenever a trial point in some iteration violates any constraints, its value is greater than the currently best vertex. The last two sums give a bias towards the feasible region when all vertices are infeasible. The derivative discontinuity of the terms with absolute value should not be problematic since the method is not based on any model, but merely on comparison of function values. A practical implementation is similar to the original algorithm.  $f$  is first evaluated at the vertices of the initial simplex and the highest value is stored. Then the additional terms in (2) are added to these values, and in subsequent iterates  $f$  is replaced by  $f'$ .

Another variant of the simplex method is the multidirectional search algorithm. Its iteration consists of similar steps to the Nelder-Mead algorithm, except that all vertices but the best one are involved in all operations. There is no shrink step and the contraction step is identical to the shrink step of the Nelder-Mead algorithm. Possible steps are shown in Figure 3. The convergence proof exists for this method<sup>[2]</sup>, but in practice it performs much worse than the Nelder-Mead algorithm. This is due to the fact that more function evaluations are performed at each iteration and that the simplex can not be adapted to the local function properties as well as the former algorithm. The shape of the simplex can not change, i.e. angles between its edges remain constant (see Figure 3). The multidirectional search algorithm is better suited to parallel processing because  $n$  function evaluations can always be performed simultaneously.



**Figure 3:** possible steps in the multidirectional search algorithm: reflection, expansion, and contraction.

### 3 SIMPLEX METHODS FOR INEQUALITY CONSTRAINTS

We define the *feasible set*  $\Psi$  as the set of points that satisfy all constraints:

$$\Psi = \{ \mathbf{x}; c_i(\mathbf{x}) \leq 0 \forall i \in I \wedge c_j(\mathbf{x}) = 0 \forall j \in E \}. \quad (3)$$

We will denote by  $\mathbf{x}^*$  the solution of the problem (1).  $\mathbf{x}^*$  is a local solution (optimum) of the problem (1) if there exists a neighborhood  $\Omega$  of  $\mathbf{x}^*$  such that

$$\mathbf{x} \geq \mathbf{x}^* \quad \forall \mathbf{x} \in \Psi \cap \Omega. \quad (4)$$

We will define the constraint residual

$$r_i(\mathbf{x}) = \begin{cases} 0; & i \in I \wedge c_i(\mathbf{x}) \leq 0 \\ c_i(\mathbf{x}); & i \in I \wedge c_i(\mathbf{x}) > 0. \\ |c_i(\mathbf{x})|; & i \in E \end{cases} \quad (5)$$

We will denote the sum of residuals as

$$S(\mathbf{x}) = \sum_{i \in I \cup E} r_i(\mathbf{x}) \quad (6)$$

And the maximal residual



$$R(\mathbf{x}) = \max_{i \in I \cup E} r_i(\mathbf{x}) \quad (7)$$

Further, we will define the number of violated constraints as

$$N_r(\mathbf{x}) = \sum_{i \in I \cup E} \begin{cases} 1; & r_i(\mathbf{x}) > 0 \\ 0; & r_i(\mathbf{x}) = 0 \end{cases} \quad (8)$$

### 3.1 Adaptation by comparison relations

In the simplex method, only comparison of the objective function at different parameters is performed. In fact, no predictions are made based on actual values or gradients of the objective function, i.e. only the comparison relation “is greater than” is used.

The idea is then to adapt the method for constraint problems by introduction of the “is better than” in such a way that it takes into account constraints. We will denote the relation as

$$\mathbf{x}_1 \tilde{<} \mathbf{x}_2, \quad (9)$$

which will read  $\mathbf{x}_1$  is better than  $\mathbf{x}_2$ .

We impose the following common rules:

$$\begin{aligned} \mathbf{x}_1 \tilde{<} \mathbf{x}_2 &\Leftrightarrow \mathbf{x}_2 \tilde{>} \mathbf{x}_1 \\ \neg(\mathbf{x}_1 \tilde{<} \mathbf{x}_2) \wedge \neg(\mathbf{x}_2 \tilde{<} \mathbf{x}_1) &\Leftrightarrow (\mathbf{x}_1 \cong \mathbf{x}_2), \end{aligned} \quad (10)$$

where  $\mathbf{x}_1 \tilde{>} \mathbf{x}_2$  means  $\mathbf{x}_1$  is worse than  $\mathbf{x}_2$ . We also define the equivalence relation such that  $\mathbf{x}_1 \cong \mathbf{x}_2$  means  $\mathbf{x}_1$  is equally good than  $\mathbf{x}_2$  and  $\mathbf{x}_1 \tilde{\geq} \mathbf{x}_2$  means  $\mathbf{x}_1$  is worse or equally good than  $\mathbf{x}_2$ , etc.

In algorithm code, these relations will be reflected in implementation of the comparison function  $\text{cmp}(\mathbf{x}_1, \mathbf{x}_2)$ , which will be used for comparison of the simplex apices:

$$\text{cmp}(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} -1; & \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ 0; & \mathbf{x}_1 \tilde{\geq} \mathbf{x}_2 \\ 1; & \mathbf{x}_1 \cong \mathbf{x}_2 \end{cases}. \quad (11)$$

The first condition for the ordering relation is that

---

$$\mathbf{x}^* \tilde{\leq} \mathbf{x} \quad \forall \mathbf{x} \in \Omega, \quad (12)$$

Where  $\Omega$  is the neighborhood from (4). This will be satisfied if the following is valid:

$$\begin{aligned} \mathbf{x}_1 \in \Psi \wedge \mathbf{x}_2 \notin \Psi &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ \mathbf{x}_1 \in \Psi \wedge \mathbf{x}_2 \in \Psi \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \end{aligned} \quad (13)$$

The second line also implies

$$\mathbf{x}_1 \in \Psi \wedge \mathbf{x}_2 \in \Psi \wedge f(\mathbf{x}_1) = f(\mathbf{x}_2) \Rightarrow \mathbf{x}_1 \tilde{=} \mathbf{x}_2$$

This is the primary condition for the comparison relation. In order to apply the comparison relation in the minimization algorithm, we will have to require that some additional conditions are satisfied.

In particular, it is desired that for any two point  $\mathbf{x}_0$  there exist a local solution  $\mathbf{x}^*$  of the problem (1) such that a descent path connecting  $\mathbf{x}$  and  $\mathbf{x}^*$  exists:

$$\forall \mathbf{x} \exists \mathbf{x}^*, \mathbf{s}(t), \begin{cases} \mathbf{x}^* \text{ local optimum} \\ \mathbf{s}(0) = \mathbf{x} \\ \mathbf{s}(1) = \mathbf{x}^* \\ \forall t_1 \in [0,1], t_2 \in [0,1] \quad (t_2 > t_1 \Rightarrow \mathbf{s}(t_2) \tilde{\leq} \mathbf{s}(t_1)) \\ \forall t_1 \in [0,1], t_2 \in [0,1] \\ \left( \forall \sigma > 0 \exists \varepsilon > 0 \left( |t_1 - t_2| < \varepsilon \Rightarrow \|\mathbf{s}(t_1) - \mathbf{s}(t_2)\| < \sigma \right) \right) \end{cases} \quad (14)$$

We must specify the rules additional to (13) such that relations  $\tilde{\leq}$  and  $\tilde{=}$  will be precisely defined and (14) will be satisfied for regular cases.

### 3.1.1 Different definitions of comparison functions

#### 3.1.1.1 Comparison based on sum of residuals

$$\begin{aligned} S(\mathbf{x}_1) < S(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ S(\mathbf{x}_1) = S(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \end{aligned} \quad (15)$$

$$cmp_S(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} -1; & (S(\mathbf{x}_1) < S(\mathbf{x}_2)) \vee (S(\mathbf{x}_1) = S(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2)) \\ 0; & S(\mathbf{x}_1) = S(\mathbf{x}_2) \wedge f(\mathbf{x}_1) = f(\mathbf{x}_2) \\ 1; & otherwise \end{cases} \quad (16)$$

### 3.1.1.2 Comparison based on maximal residual

$$\begin{aligned} R(\mathbf{x}_1) < R(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ R(\mathbf{x}_1) = R(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \end{aligned} \quad (17)$$

The second line also implies

$$\mathbf{x}_1 \in \Psi \wedge \mathbf{x}_2 \in \Psi \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2) \Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2.$$

$$cmp_R(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} -1; & (R(\mathbf{x}_1) < R(\mathbf{x}_2)) \vee (R(\mathbf{x}_1) = R(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2)) \\ 0; & R(\mathbf{x}_1) = R(\mathbf{x}_2) \wedge f(\mathbf{x}_1) = f(\mathbf{x}_2) \\ 1; & otherwise \end{cases} \quad (18)$$

### 3.1.1.3 Comparison based on number of violated constraints and sum of residuals

$$\begin{aligned} N_r(\mathbf{x}_1) < N_r(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2) \wedge S(\mathbf{x}_1) < S(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2) \wedge S(\mathbf{x}_1) = S(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \end{aligned} \quad (19)$$

$$cmp_{NS}(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} -1; (N_r(\mathbf{x}_1) < N_r(\mathbf{x}_2)) \vee \\ \quad (N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2)) \wedge \\ \quad (S(\mathbf{x}_1) < S(\mathbf{x}_2)) \vee (S(\mathbf{x}_1) = S(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2)) \\ 0; N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2) \wedge S(\mathbf{x}_1) = S(\mathbf{x}_2) \wedge f(\mathbf{x}_1) = f(\mathbf{x}_2) \\ 1; \textit{otherwise} \end{cases} \quad (20)$$

#### 3.1.1.4 Comparison based on number of violated constraints and maximal residual

$$\begin{aligned} N_r(\mathbf{x}_1) < N_r(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2) \wedge R(\mathbf{x}_1) < R(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \\ N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2) \wedge R(\mathbf{x}_1) = R(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow \mathbf{x}_1 \tilde{<} \mathbf{x}_2 \end{aligned} \quad (21)$$

$$cmp_{NR}(\mathbf{x}_1, \mathbf{x}_2) = \begin{cases} -1; (N_r(\mathbf{x}_1) < N_r(\mathbf{x}_2)) \vee \\ \quad (N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2)) \wedge \\ \quad (R(\mathbf{x}_1) < R(\mathbf{x}_2)) \vee (R(\mathbf{x}_1) = R(\mathbf{x}_2) \wedge f(\mathbf{x}_1) < f(\mathbf{x}_2)) \\ 0; N_r(\mathbf{x}_1) = N_r(\mathbf{x}_2) \wedge R(\mathbf{x}_1) = R(\mathbf{x}_2) \wedge f(\mathbf{x}_1) = f(\mathbf{x}_2) \\ 1; \textit{otherwise} \end{cases} \quad (22)$$

This function can be possibly implemented as follows:

```
int cmpNR(analysisdata pt1, analysisdata pt2)
{
    int N1, N2;
    double R1, R2, f1, f2;
    N1=Nr(pt1); N2=Nr(pt2);
    if (N1<N2)
        return -1;
    if (N1>N2)
        return 1;
    R1=R(pt1); R2=R(pt2);
    if (R1<R2)
```

```

        return -1;
    if (R1>R2)
        return 1;
    f1=f(pt1); f2=f(pt2);
    if (f1<f2)
        return -1;
    if (f1>f2)
        return 1;
    return 0;
}
    
```

In the above code,  $pt1$  and  $pt2$  are data structures that contain data calculated within direct analyses at two parameter sets,  $Nr(pt)$  is the function that returns the number of violated constraints according to the analysis data that is its argument,  $R(pt)$  returns maximal constraint residuum and  $f(pt)$  returns the value of the objective function that is stored on its argument  $pt$ .

### *3.2 Addition of penalty terms to the objective function*

#### **3.2.1 Adding discontinuous jump to the objective function on the level set**

Because only comparison of the objective function at different parameters is performed by the Nelder-Mead simplex method, the method is relatively insensitive on discontinuity of the objective function. We can add jump discontinuities to the objective function, and this does not affect the efficiency of the method.

Let us denote  $f_m(\mathbf{x})$  the modified objective function with added jump discontinuities. As long as

$$\begin{aligned}
 f(\mathbf{x}_1) < f(\mathbf{x}_2) &\Rightarrow f_m(\mathbf{x}_1) < f_m(\mathbf{x}_2) \\
 f(\mathbf{x}_1) = f(\mathbf{x}_2) &\Rightarrow f_m(\mathbf{x}_1) = f_m(\mathbf{x}_2)
 \end{aligned}
 \tag{23}$$

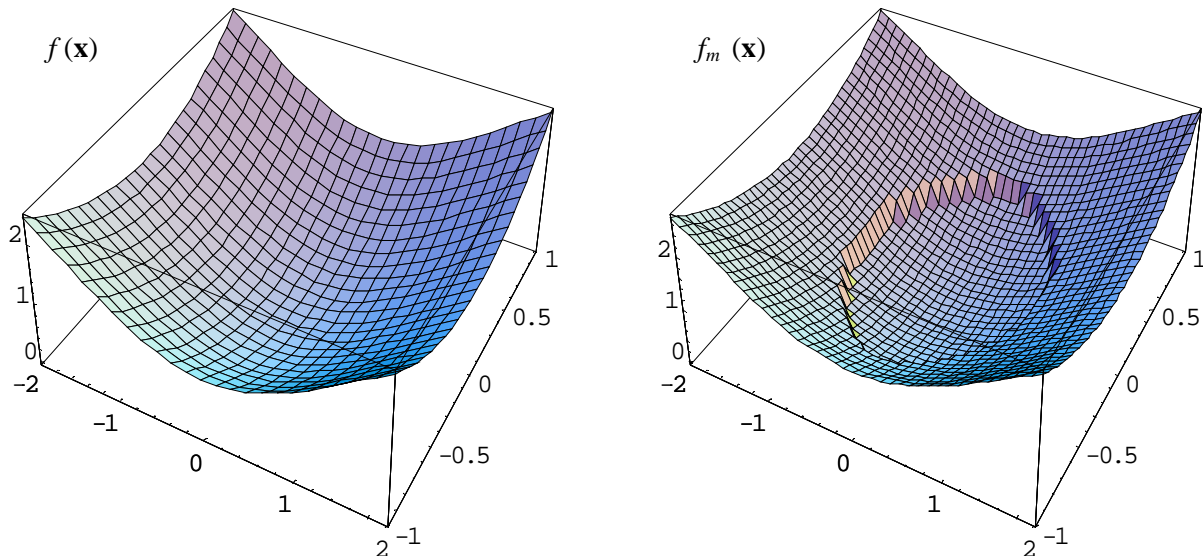
the minimum of  $f_m$  is the same as the minimum of  $f$ . The above relation is valid if we define the modified objective function in the following way (Figure 4):

$$f_m(\mathbf{x}) = \begin{cases} f(\mathbf{x}); & \mathbf{x} < c \\ f(\mathbf{x}) + h; & \mathbf{x} \geq c \end{cases} ; h > 0 .
 \tag{24}$$

This function is obtained from  $f$  by adding to it a positive constant within the following domain:

$$\Omega^+ = \{\mathbf{x} \mid f(\mathbf{x}) \geq c\}. \quad (25)$$

Edge of this domain  $\partial \Omega^+$  is the level hypersurface (isosurface in 3D, isoline in 2D) of  $f$ .



**Figure 4:** Original and modified objective function.

Modification of  $f$  according to (24) does not change performance of the Nelder-Mead algorithm. It is obvious (according to algorithm description in Section 2) that the path of the algorithm can be changed only if the result of comparisons of function values in any pair of points is changed. However, this is not the case with  $f_m(\mathbf{x})$  for which relation (23) holds.

### 3.2.2 Exact penalty functions

The fact that modification (24) that introduces a jump discontinuity does not change the performance of the algorithm indicates that the method could be efficiently modified for solution of constraint problems by forming a discontinuous exact penalty function. **Exact penalty function** is obtained by adding a penalty term such that a *minimum of the obtained penalty function corresponds to the constraint minimum of the original problem*. Solution of the original problem can then be obtained by finding a minimum of the penalty function.

When only inequality constraints are present, the penalty function can be formed by addition of penalty terms for each constraint in the following way:

$$f_p(\mathbf{x}; \mathbf{p}_p) = f(\mathbf{x}) + \sum_{i \in I} h_p(c_i(\mathbf{x}); \mathbf{p}_p), \quad (26)$$

where the penalty terms can be defined for example as

$$h_p(c_i, \{k, h\}) = \begin{cases} 0; & c_i \leq 0 \\ h + k c_i; & c_i > 0 \end{cases}; \quad h \geq 0 \wedge k \geq 0. \quad (27)$$

Non-negative penalty parameter  $k$  and  $h$  must be large enough if we want that  $f_p$  represents an exact penalty function. In the sequel, we define more precisely the conditions that the penalty function is exact penalty function.

We usually require

$$h_p(0; \mathbf{p}_p) = 0. \quad (28)$$

It is clear that in the infeasible region where  $\forall i \in I, c_i(\mathbf{x}) > 0$ , the derivative of  $h$  with respect to the violated constraint must be positive, i.e.

$$c > 0 \Rightarrow \frac{\partial h_p(c; \mathbf{p}_p)}{\partial c} > 0. \quad (29)$$

However, this is not a sufficient condition that the penalty function has a local minimum in the solution of the constrained problem. The derivative must be large enough in order to compensate for eventual falling of the objective function as the constraint function grows. What one needs to achieve is that in the infeasible region, the dot product of the gradient of the penalty function with the gradient of any constraint function belonging to a violated constraint, is positive.

The sufficient condition that the penalty function is exact (i.e. it has a local minimum in the solution of the original constrained optimization problem) is the following: There must exist a neighborhood  $\varepsilon$  of the solution  $\mathbf{x}^*$  such that in each point of the neighborhood, the gradient of the penalty function has positive dot product with gradients of all constraint functions which are greater than zero (i.e. belong to violated constraints) in that point. In this way, we can find a neighborhood of  $\mathbf{x}^*$  such that a descent path exists from any point in this neighborhood to  $\mathbf{x}^*$ . The condition can be expressed in the following way:

$$\begin{aligned} & \forall \mathbf{x} \in \varepsilon, \forall i \in I, \\ & c_i(\mathbf{x}) > 0 \Rightarrow \left\langle \nabla_{\mathbf{x}} f_p(\mathbf{x}; \mathbf{p}_p), \nabla c_i(\mathbf{x}) \right\rangle > 0. \end{aligned} \quad (30)$$

The above equation says that the directional gradient of the penalty function must be positive in the direction of the gradient of any violated constraint. From (26) we have

$$\nabla_{\mathbf{x}} f_p(\mathbf{x}; \mathbf{p}_p) = \nabla f(\mathbf{x}) + \sum_{i \in I} \left. \frac{\partial h_p(c; \mathbf{p})}{\partial c} \right|_{c=c_i(\mathbf{x})} \nabla c_i(\mathbf{x}) \quad (31)$$

Equation (30) defines the condition that the penalty function has a local minimum that corresponds to the solution of the original constraint optimization problem. From the algorithmic point of view this is not sufficient. We want to ensure that minimization algorithm applied to the penalty function will actually yield the local minimum that corresponds to a local solution of the unconstrained problem (since the penalty function can have several local minima or can even be unbounded below). In our case we will apply the unconstrained Nelder-Mead simplex algorithm, but the same reasoning applies to application of other algorithms. It is intuitively obvious that if the region  $\varepsilon$  on which (30) holds is larger, the applied minimization algorithm will converge to the solution of the original problem from a larger region. Running the algorithm from a starting point that is far from the region where (30) holds will more likely cause it to diverge (in the case that the penalty function is unbounded below) or converge to a local minimum that is not a solution of the original problem.

The best is if the condition (30) holds everywhere. Considering equations (30) and (31), in order to achieve that, the function  $h_p(c; \dots)$  must grow sufficiently fast with its  $c$ . In this way, the second term in (31) can compensate for eventual negative projection of the gradient of the objective function on the gradients of violated constraints. However, making  $h_p(c; \dots)$  grow too fast close to  $c=0$  would introduce ill-conditioning in the minimization of the penalty function. We must therefore look for a suitable compromise, which is not trivial in some cases.

While addition of discontinuous term of the form (24) does not affect the performance of the Nelder-Mead simplex method, addition of penalty terms of the form (26) can significantly reduce its efficiency. This is because the penalty terms limit the space where the simplex moves, and the simplex makes more rejected trials when hitting sharp growth of the penalty function at constraint boundaries.

A disadvantage of the penalty function generated by  $h_p$  of the form (27) is that it is difficult to fulfill the condition (30) on a large sub-domains of the infeasible domain in the cases where the objective function falls progressively or when the constraint functions grow regressively with the distance from the zero level hyper-surfaces of constraint functions. This can be alleviated by making  $h_p$  grow progressively with increasing positive argument by adding exponential or higher order monomial terms, e.g.

$$h_p(c_i, \{k, h\}) = \begin{cases} 0; & c_i \leq 0 \\ h + k \left( c_i + \left(\frac{c_i}{4}\right)^2 + \left(\frac{c_i}{8}\right)^3 + \left(\frac{c_i}{16}\right)^4 + \exp\left(\frac{c_i}{64}\right) \right); & c_i > 0 \end{cases}; h \geq 0 \wedge k \geq 0 \quad (32)$$



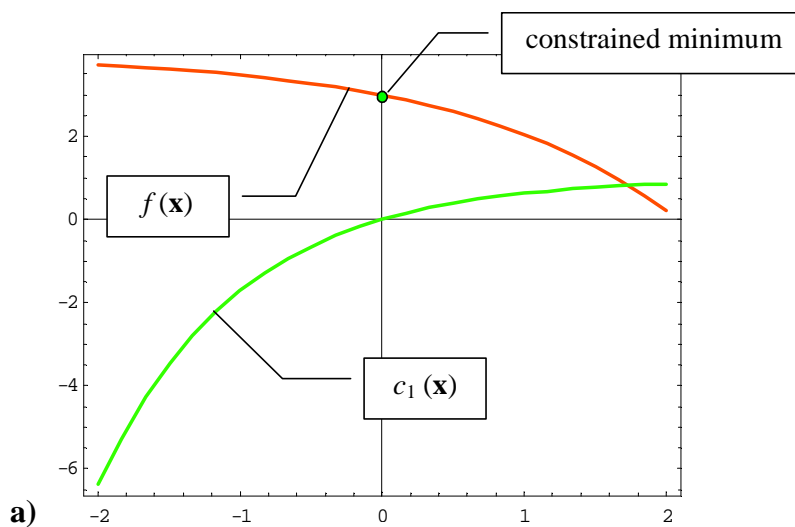
Increasing denominators take care that higher order terms contribute significantly only when the constraint functions are large enough, which makes minimization of the penalty function less ill conditioned. However, this is not so important when the Nelder-Mead simplex method is used for minimization of the penalty function, because this method only makes comparisons of function values and does not make use of higher order function information.

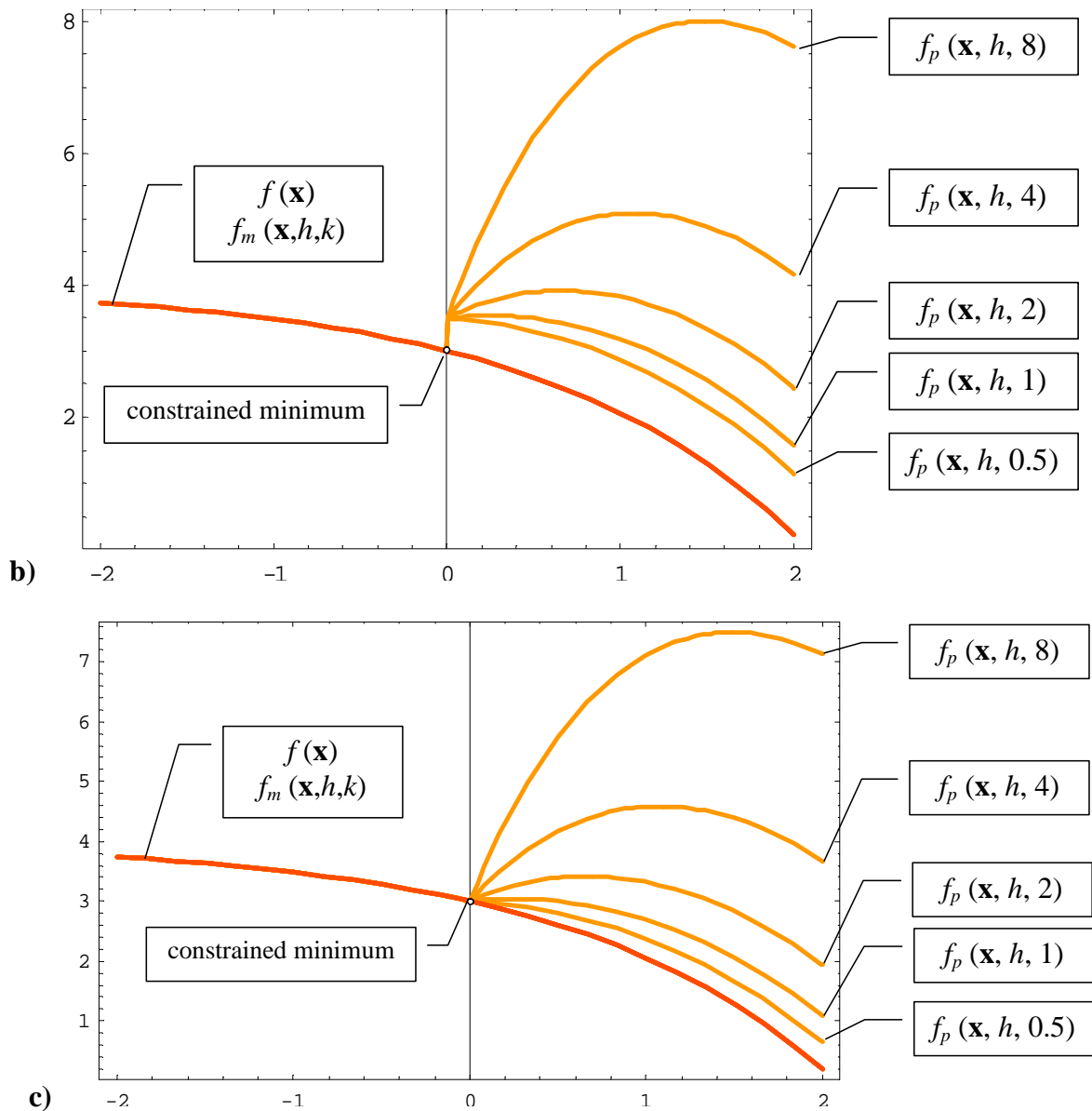
### 3.2.2.1 Examples: discontinuous exact penalty function

In order to illustrate addition of a penalty term, we consider the following one dimensional problem:

$$\begin{array}{ll} \text{minimise} & f(x) = 4 - e^{-\frac{2}{3}x} \\ \text{subject to} & c_1(x) \leq 1 - e^{-x} \leq 0 \end{array} \quad (33)$$

We form penalty functions according to (26) and (27). Results are shown in Figure 5.





**Figure 5:** Problem (33): **a)** problem objective function and constraint function, **b)** family of penalty functions of form (27) for different penalty parameters ( $h=0.5$  is constant) and **c)** family of penalty functions at the same parameters  $k$  and at  $h=0$ .

The above example has two features that are somehow problematic for penalty methods. Firstly, the objective function progressively falls in the infeasible region ( $x>0$ ) as the violation of constraints (i.e. value of the constraint function) grows. Secondly, the constraint function falls regressively, i.e. its second derivative is non-zero. This means that the function  $h_p$  that defines the penalty terms should grow progressively enough in order to compensate for progressive falling of the objective function and regressively growth of the constraint functions. Since we have chosen a

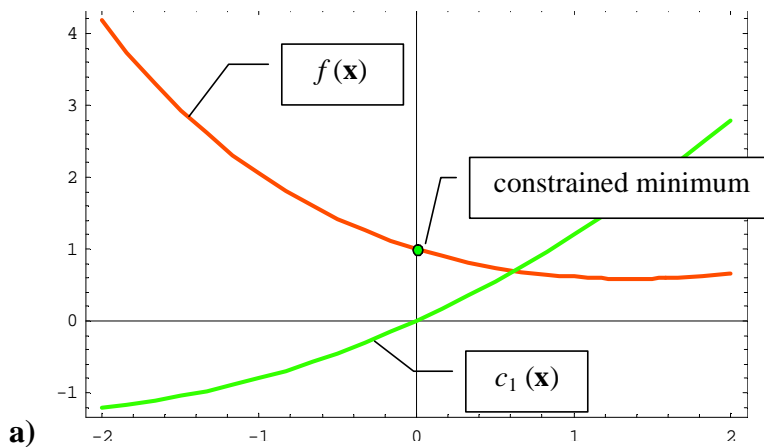
simple linear form of  $h_p$ , there will always exist a subset of infeasible region where the value of the penalty function will be smaller than its value in the solution of the constraint problem. The difficulties related to these problematic features can be alleviated by using different (progressively growing) forms of  $h_p$  such as (32).

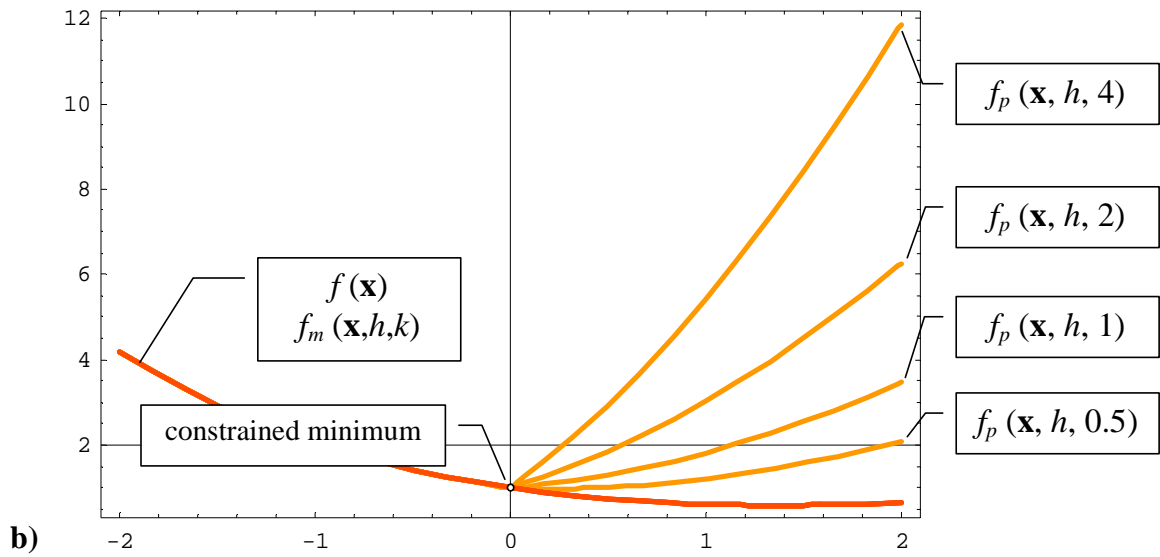
It can be seen from the figure that for small  $k$  and  $h=0$  the penalty function does not have a local minimum in the solution of the constrained problem. If  $h>0$  then the penalty function does have a local minimum in the solution of the constrained problem even for  $k=0$ . However, for minimization algorithms it will be difficult to locate this minimum because they can easily jump over it. For large  $k$ , the subset of the infeasible region for which condition (30) holds increases. However, the sharpness of the edge that the penalty function forms in the solution of the original problem also increases, and this feature usually affects the efficiency of the applied minimization algorithm. For the Nelder-Mead algorithm this feature is not problematic in one dimension, but can be in more dimensions where zero contours of constraints form a sharp cone with its tip located in the problem solution.

The next example does not contain the problematic features of the previous one:

$$\begin{array}{ll}
 \text{minimise} & f(x) = e^{-\frac{2}{3}x} + \frac{1}{10}x^2 \\
 \text{subject to} & c_1(x) = x + 0.2x^2
 \end{array} \tag{34}$$

The problem and formation of penalty functions is illustrated in Figure 6.



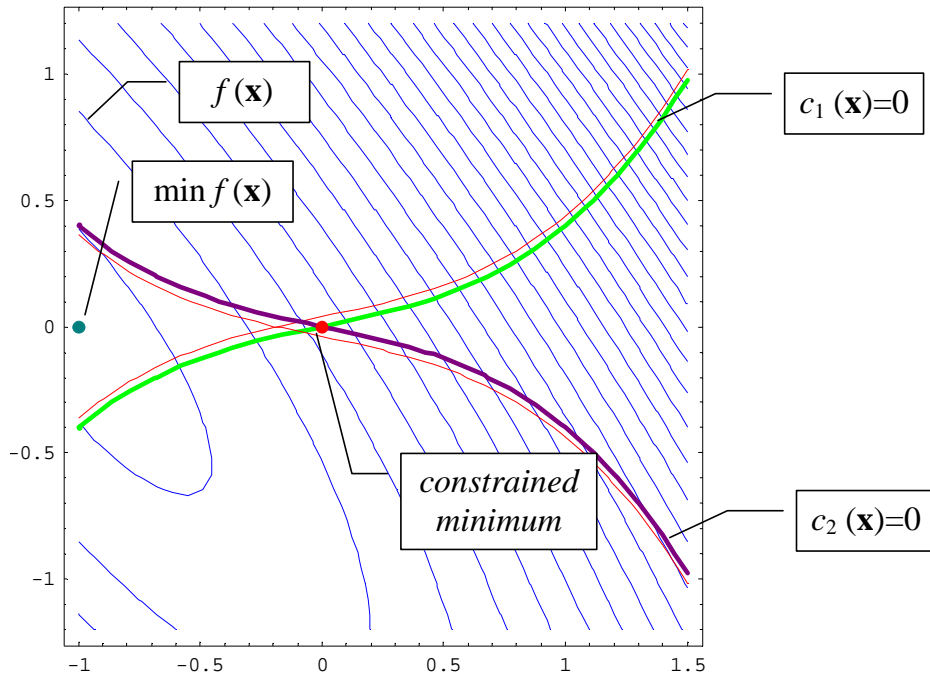


**Figure 6:** Problem (34): **a)** problem objective function and constraint function and **b)** family of penalty functions of form (27) for different penalty parameters ( $h=0$  is constant).

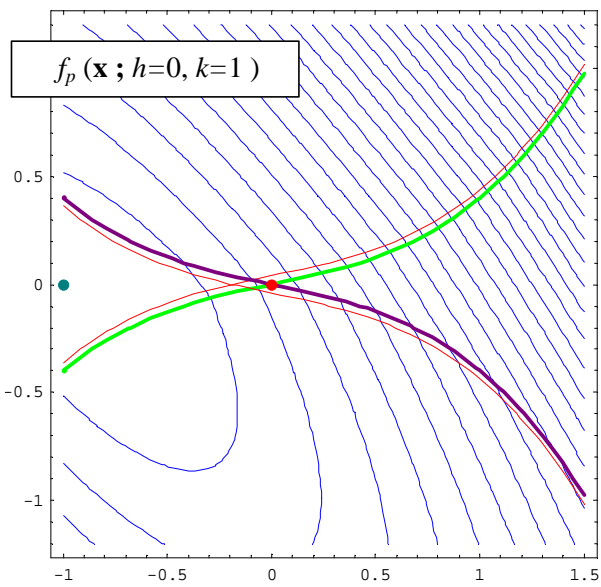
Next we consider the following two dimensional example (Figure 7):

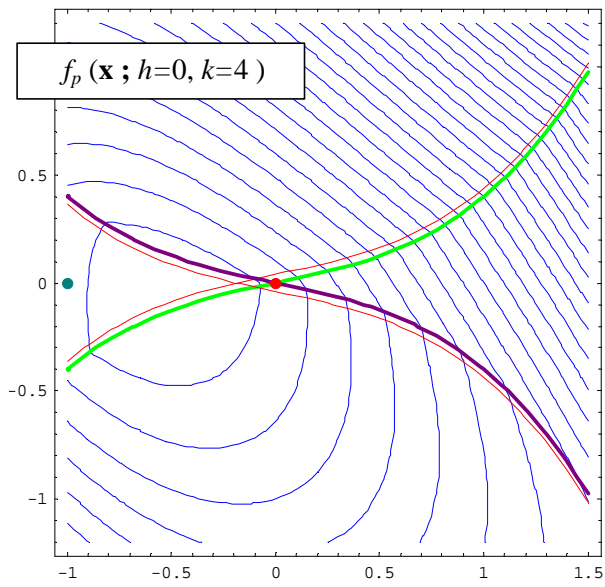
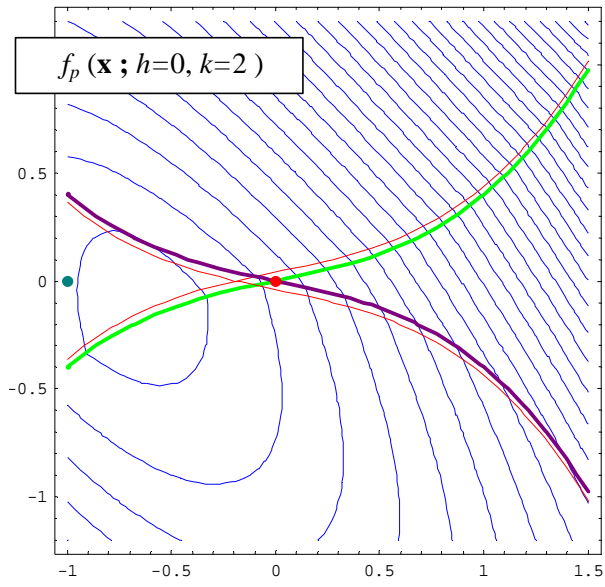
$$\begin{aligned}
 &\text{minimise} && f(x, y) = (x+1)^2 + 2(x+1+y)^2 \\
 &\text{subject to} && c_1(x, y) = -0.2x^3 - 0.2x + y \\
 &\text{and} && c_1(x, y) = -0.2x^3 - 0.2x - y
 \end{aligned} \tag{35}$$

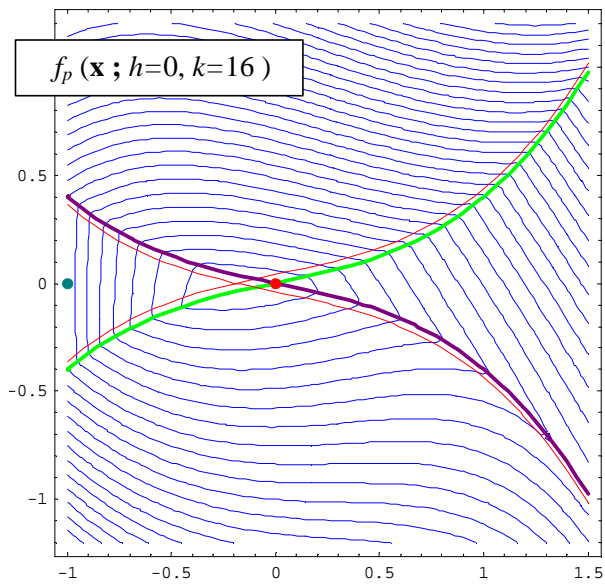
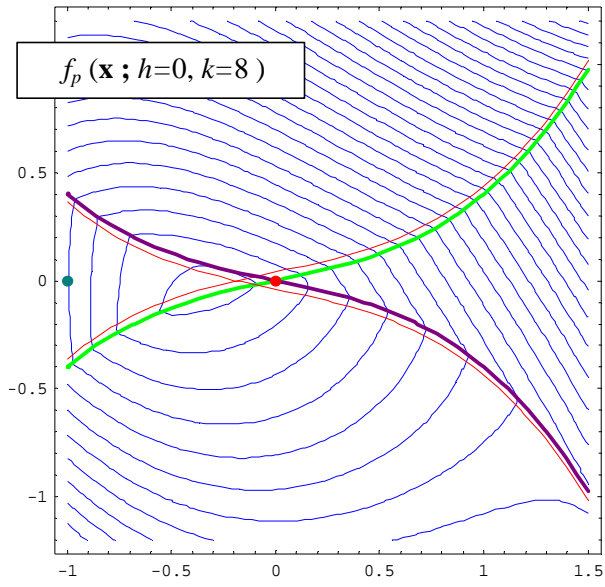
Figure 8 and Figure 9 show penalty functions of the form (27) for this problem for different values of parameter  $k$ .

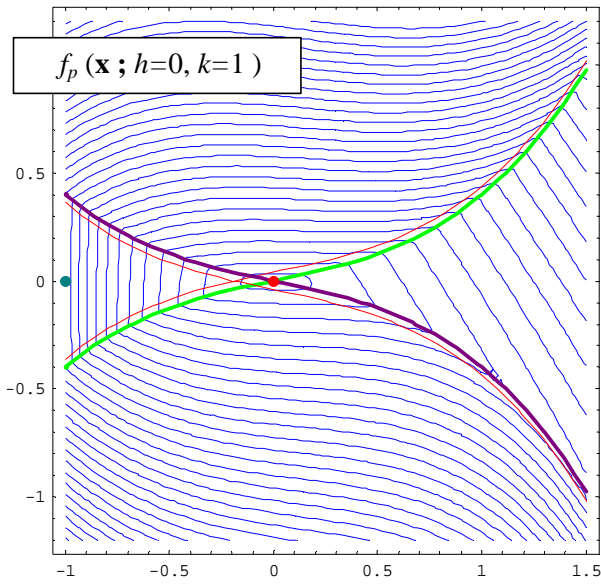


**Figure 7:** Illustration of problem (33).

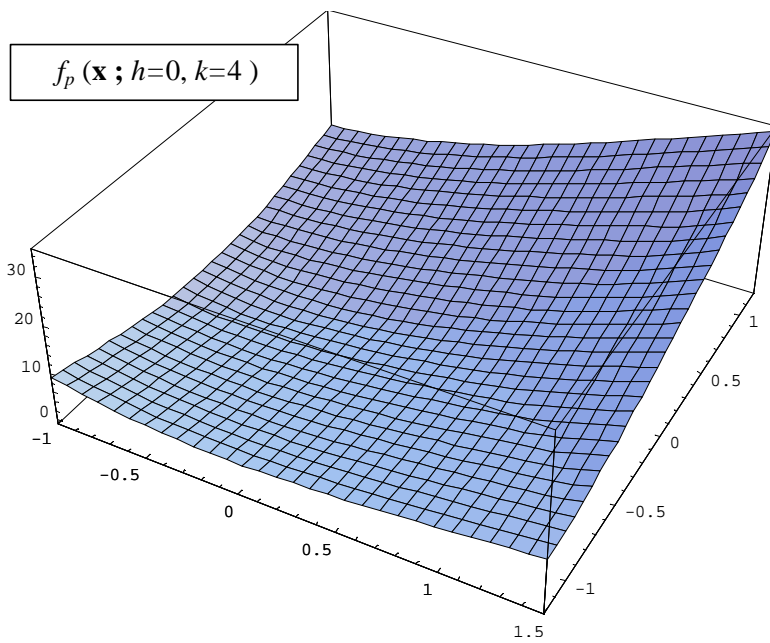




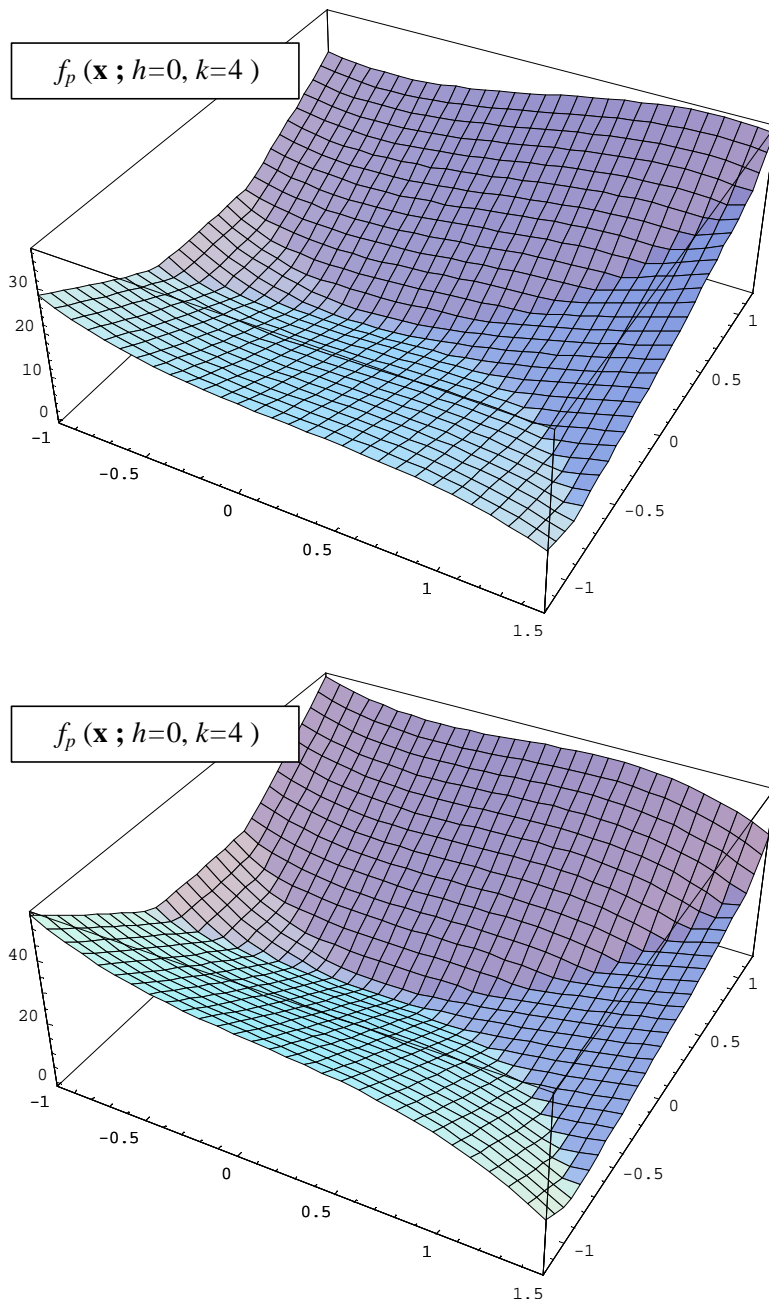




**Figure 8:** Contour plots of penalty functions for problem (33), with different penalty parameters  $k$  and  $h=0$ .







**Figure 9:** Penalty functions of form (27) for problem (33), with different penalty parameters  $k$  and  $h=0$ .

### 3.2.3 Adaptive penalty algorithm

Addition of penalty terms can badly affect performance of the Nelder-Mead simplex algorithm, especially when the zero level hyper-surfaces of inequality constraints form narrow valleys around the solution (see e.g. problem (33)). This can be overcome by adapting penalty parameters through the algorithm performance. The approach assumes some features of usual penalty methods where the penalty parameter is gradually increased, and the generated successive penalty functions are minimized by using the minimum of the previous penalty function as a starting guess. In this way the problems with ill conditioning of the minimization of penalty functions is alleviated, and successive minima of the penalty functions converge to the constraint minimum.

The idea of adaptive penalty algorithm is a bit different in that the penalty parameters are adaptively adjusted during the algorithm progress, rather than after complete minimizations are performed. Detailed description of the algorithm is beyond the scope of this report.

### 3.3 *Strict consideration of bound constraints*

This section describes how violation of bound constraints can be prevented during minimization by the simplex method. This is done by a new analysis function, which shifts parameter components that violate bound constraints on interval limits, calculates the objective and constraint functions in new points, and adds a penalty term that depends on how much the constraints were violated.

This procedure should be significantly changed for algorithm that uses function approximations to increase the speed. This is because the procedure introduces discontinuities in the derivatives at constraint bounds.

Let us say that we are solving the problem (1) with only inequality constraints and with additional *bound constraints* on the parameter vector:

$$\underline{\forall k, l_k \leq x_k \leq r_k} . \quad (36)$$

In many cases, the bound constraints are defined only for particular parameters, for some of which only minimal ( $l_k$ ) or only maximal ( $r_k$ )<sup>1</sup> value is defined. For the sake of convenience in implementation of computational procedures, we will use the formula (36) as if both bounds are defined, and will set  $l_k = -\infty$  and  $r_k = \infty$  for those cases where the bounds are not defined.

Let us say that a direct analysis is called at parameters  $\mathbf{x}=\{x_1, x_2, \dots, x_n\}$  where some of the bound constraints are violated. We actually run the analysis at modified parameters  $\tilde{\mathbf{x}}$ , which are obtained by correction of actual parameters (at which the analysis is requested) in such a way that which are defined in such a way that bound constraints are satisfied:

---

<sup>1</sup> In this notation, letter  $l$  is used as “left” and  $r$  as “right”.

$$\forall k, \tilde{x}_k = \begin{cases} x_k ; l_k \leq x_k \leq r_k \\ l_k ; x_k < l_k \\ r_k ; x_k > r_k \end{cases} \quad (37)$$


---

We then modify the value of the objective function in the following way:

$$\tilde{f}(\mathbf{x}) = f(\tilde{\mathbf{x}}) + \sum_{i=1}^n h_{k_i}(\mathbf{x}) + h_{k_r}(\mathbf{x}), \quad (38)$$


---

where

$$h_{k_l}(\mathbf{x}) = \begin{cases} h_p(l_k - x_k; \mathbf{p}_p); l_k > -\infty \\ 0; otherwise \end{cases} \quad (39)$$

$$h_{k_r}(\mathbf{x}) = \begin{cases} h_p(x_k - r_k; \mathbf{p}_p); r_k < \infty \\ 0; otherwise \end{cases}$$


---

and  $k_p$  is a function for generation of penalty terms of a convenient form such as (27) or (32). Constraint functions are not modified and are simply set to the values of constraint functions at  $\tilde{\mathbf{x}}$ :

$$\forall i \in I, \tilde{c}_i(\mathbf{x}) = c_i(\tilde{\mathbf{x}}). \quad (40)$$


---

Expression (39) is addition of penalty terms as in (26) ad (27), where the following constraint functions are assigned to bound constraints:

$$\begin{aligned} l_k > -\infty &\Rightarrow c_{k_l}(\mathbf{x}) = l_k - x_k \\ r_k < \infty &\Rightarrow c_{k_r}(\mathbf{x}) = x_k - r_k \end{aligned} \quad (41)$$

Penalty terms have the following contributions to the gradient of the objective function:

$$\begin{aligned} \nabla h_{k_l}(\mathbf{x}) &= - \frac{\partial h_p(t; \mathbf{p}_p)}{\partial t} \Bigg|_{t=(l_k - x_k)} \mathbf{e}_k \\ \nabla h_{k_r}(\mathbf{x}) &= \frac{\partial h_p(t; \mathbf{p}_p)}{\partial t} \Bigg|_{t=(x_k - r_k)} \mathbf{e}_k \end{aligned}, \quad (42)$$

where  $\mathbf{e}_k$  is the co-ordinate vector  $k$  (component  $k$  equals 1, others equal 0).

---

### ***3.4 Implementation remarks on penalty terms and bound constraints***

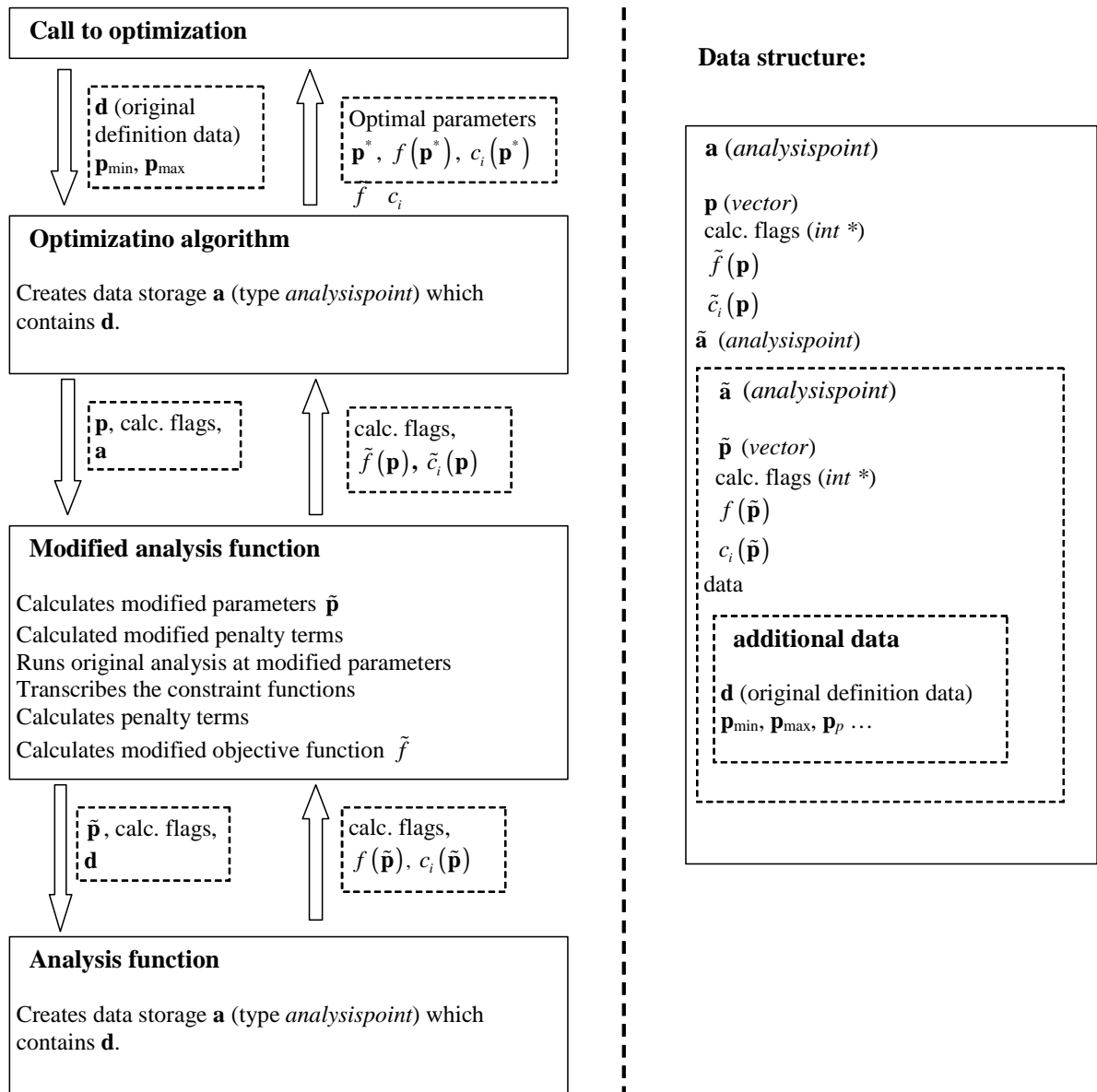
This Section discusses some details relevant for implementation of penalty terms and bound constraints in the *IoptLib* (Investigative Optimization Library). It is meant for developers and advanced users of the library because a good knowledge of the library is necessary to understand the section.

We consider modification of the original analysis function according to (38). In principle, the implementation of the modified analysis is quite simple: we form a new analysis function that takes the parameters, calculates the sum of penalty terms according to parameters and bound constraints, modifies the parameters, runs the analysis function at the modified parameters, adds the calculated objective function to the sum of penalty terms to form the modified objective function, takes the calculated constraint functions and returns the results. All the operations could be performed in place, i.e. without allocation of additional space for auxiliary variables.

The scheme is a bit more complicated if one would like to preserve information that is not returned by the modified analysis function, e.g. the modified parameters at which the original analysis function is performed, or the value of the objective function at the modified parameters. In the modified Nelder-Mead algorithm, for example, this information is sometimes desired for checking algorithm progress or for post-processing and analyzing the acquired results. In this case, additional storage is necessary to keep the additional information.

There may be different possibilities with respect to what information should be kept, and modification of analysis defined by (38) can be combined by other modifications such as adaptive penalty functions. Different ways of handling the storage of additional data (together with the appropriate data types) should be implemented in order to optimize the speed and memory usage, but this would increase the complexity of code and its maintenance costs. In *IoptLib* a compromise solution is achieved by using some standard data types and related functionality. In particular, the type `analysispoint` is utilized that is intended for storage of analysis results. Because of dynamically allocated storage for things such as optimization parameters and values of objective and constraint functions, the amount of additional memory necessary to support comfortable standard uses is not large. Manipulation of additional storage is relatively simple because standard functionality designed around `analysispoint` the type can be used. This functionality can be easily extended in line with the standards when necessary. Beside some additional storage, the cost for using standard data types and procedures is also some additional data transcriptions (e.g. the values of constraint functions are transcribed from the nested (inner) `analysispoint` structure to the outer one).

A scheme for performing the modified analysis function is shown in Figure 10. The structure of data that is passed to the modified function is also shown in the figure.



**Figure 10:** Scheme for handling bound constraints and penalty terms in algorithms.

### **3.4.1 Basic tools for handling bound constraints**

### **3.4.2 Penalty generating functions**

### **3.4.3 Conversion of bound constraints to ordinary constraints**

## **4 SIMPLEX METHODS FOR EQUALITY CONSTRAINTS**

## **5 ACCELERATING CONVERGENCE WITH RESPONSE APPROXIMATION**

## 6 SOFT SIMPLEX (ADAPTIVE PENALTY)

## 7 APPENDIX

### 7.1.1 Relations

**Equivalence relations** are those that are *reflexive*, *symmetric* and *transitive*.

**Relations of partial ordering** are those that are *reflexive*, *antisymmetric* and *transitive*.

**Relations of ordering** (complete ordering, linear ordering) are *reflexive*, *antisymmetric*, *transitive* and *linear*.

Properties of binary relations (between elements of a set  $A$ ):

*Reflexive*:  $\forall a \in A \ a R a$

*Ireflexive*:  $\forall a \in A \ \neg a R a$

*Symmetric*:  $\forall a, b \in A \ (a R b \Rightarrow b R a)$

*Antisymmetric*:  $\forall a, b \in A \ (a R b \wedge b R a \Rightarrow a = b)$

*Transitive*:  $\forall a, b, c \in A \ (a R b \wedge b R c \Rightarrow a R c)$

*Linear*:  $\forall a, b \in A \ (a R b \vee b R a)$



### References:

- [1] I. Grešovnik, *IoptLib User's Manual*, revision 0. Ljubljana, 2007.
- [2] M. H. Wright, *Direct Search Methods: Once Scorned, Now Respectable*, in D. F. Griffiths and G. A. Watson (eds.), *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis)*, p.p. 191 – 208, Addison Wesley Longman, Harlow, 1996.
- [3] W.H. Press, S.S. Teukolsky, V.T. Vetterling, B.P. Flannery, *Numerical Recipes in C – the Art of Scientific Computing*, Cambridge University Press, Cambridge, 1992.
- [4]
- [5]
- [6] R. Fletcher, *Practical Methods of Optimization (second edition)*, John Wiley & Sons, New York, 1996).
- [7] M. H. Wright, *Direct Search Methods: Once Scorned, Now Respectable*, in D. F. Griffiths and G. A. Watson (eds.), *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis)*, p.p. 191 – 208, Addison Wesley Longman, Harlow, 1996.
- [8]
- [9]
- [10]
- [11]
- [12] I. Grešovnik. "A General Purpose Computational Shell for Solving Inverse and Optimisation Problems - Applications to Metal Forming Processes", Ph.D. thesis, available at <http://www.c3m.si/inverse/doc/phd/index.html> .
- [13]
- [14]
- [15]
- [16]
- [17]

## 8 SANDBOX (THIS IS NOT PART OF THIS REPORT)

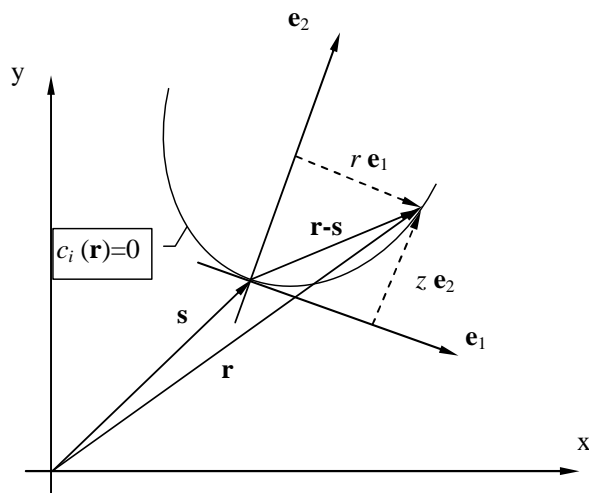


Figure 11: Basis vectors for definition of rotationally symmetric constraint functions .

