EDUCATIONAL AID

in the frame of
TEMPUS JEP 0438-92/93
contractor : Prof. Guy Guerlement
Faculte Polytechnique de Mons

MATHEMATICAL PROGRAMMING METHODS IN STRUCTURAL OPTIMIZATION

by

Assoc. Prof. Károly Jármai

University of Miskolc, Hungary
No.: Edu. aid. TEMPUS JEP 0438-UM-10-1993.

1. INTRODUCTION

The different single-objective optimization techniques make the designer able to determine the optimal sizes of structures, to get the best solution among several alternatives. The efficiencies of these mathematical programming techniques (MP) are different. A large number of algorithms has been proposed for the nonlinear programming solution [1,2]. Each technique has its own advantages and disadvantages, no one algorithm is suitable for all purposes. The choice of a particular algorithm for any situation depends on the problem formulation and the user.

We have shown the efficiency of these techniques at the optimum design of steel structures such as single bay frames, main girders of overhead travelling cranes, spindle-bearing systems of a machine tool, stiffened plates, cellular plates, compressed columns, sandwich beams etc. [3,4,5].

The general formulation of a single-criterion nonlinear programming problem is the following:

$$\begin{aligned} &\text{minimize} \quad f(\mathbf{x}), && \mathbf{x} = x_1, x_2, .., x_N, \\ &\text{subject to} \quad g_j(\mathbf{x}) \le 0, && j = 1,2,.....,P, \\ &\qquad\qquad h_i(\mathbf{x}) = 0, && i = P+1,...,P+M. \end{aligned} \tag{1}$$

$f(\mathbf{x})$ is a multivariable nonlinear function, $g_j(\mathbf{x})$ and $h_i(\mathbf{x})$ are nonlinear inequality and equality constraints.

In this paper we describe four different single objective optimization methods, the Sequential Unconstrained Minimization Technique (SUMT), the combinatorial Backtrack, the Method of Moving Asymptotes (MMA) and the Feasible Sequential Quadratic Programming (FSQP) technique.

2. SEQUENTIAL UNCONSTRAINED MINIMIZATION TECHNIQUE (SUMT)

The procedure was developed by Fiacco and McCormick [6]. The technique uses the problem constraints and the original objective function to form an unconstrained objective function which is minimized by any appropriate unconstrained, multivariable technique.

The algorithm proceeds as follows:

1. A modified objective function is formulated consisting of the original function and penalty functions with the form

$$F(x,r)=f(x)+r_k*\sum_{j=1}^{P} 1/g_j(x) + r_k^{-1/2} * \sum_{i=P+1}^{P+M} h_i^2(x) \tag{2}$$

where r_k is a positive constant. As the algorithm progresses, r_k is re-evaluated to form a monotonically decreasing sequence $r_1 > r_2 > ... > 0$. As r_k becomes small,

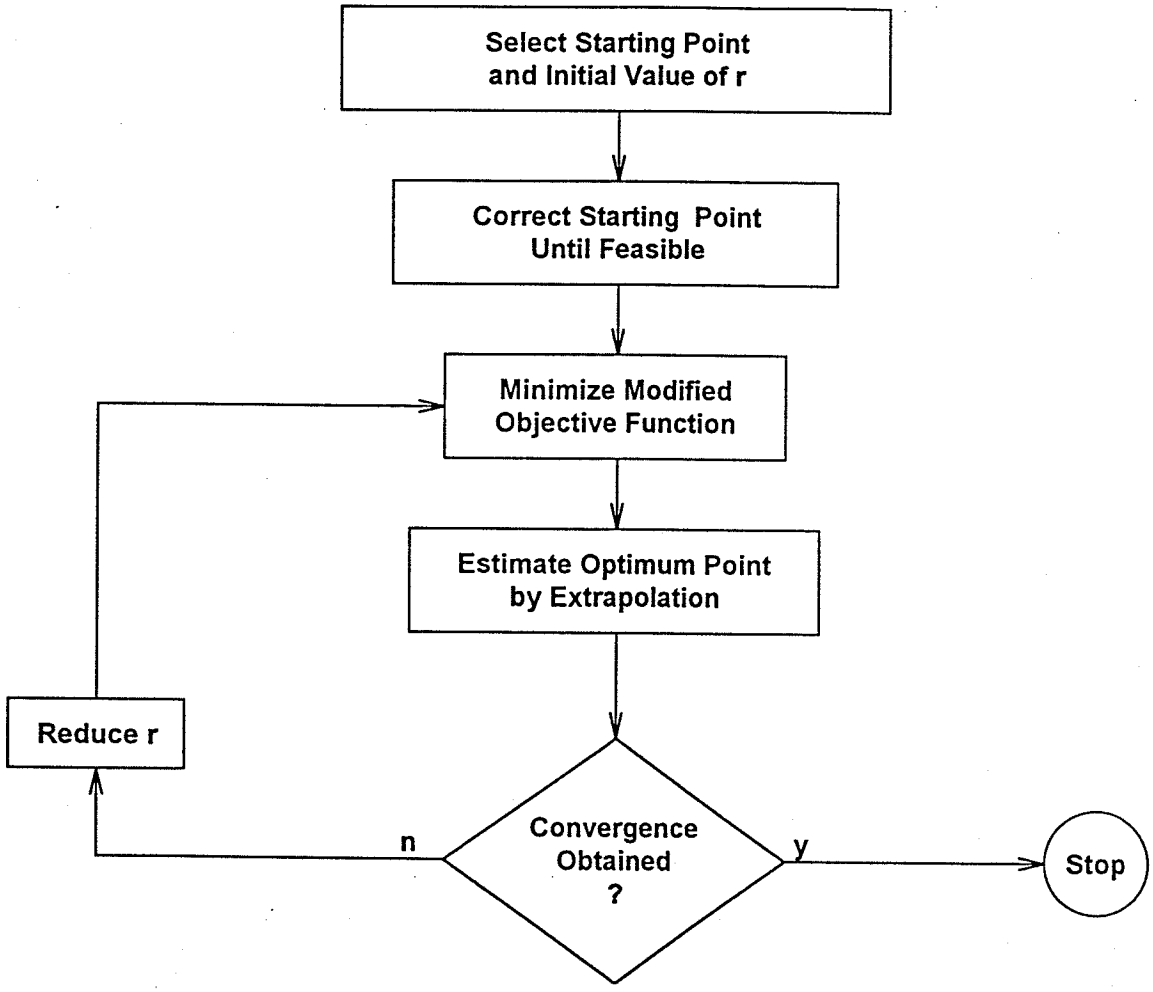


Fig.1 Flow chart of the Fiacco McCormick SUMT algorithm

under suitable conditions F approaches f and the problem is solved.

2. Select a starting point (feasible or nonfeasible) and an initial value for r_K.

3. Determine the minimum of the modified objective function for the current value of r_K using an appropriate technique (several options available).

4. Estimate the optimal solution using extrapolation formulae.

5. Select a new value for r_K and repeat the procedure until the convergence criterion is satisfied. The flow chart of this method can be seen in Fig. 1. The value of r_K is decreasing during the procedure down to 10^{-4} or so.

The penalty function of the inequality constraints can be an other function in (2), instead of reciprocal $\sum_{j=1}^{P} 1/g_j(x)$ a logarithmic function $-\sum_{j=1}^{P} \ln(g_j(x))$. The efficiency of the penalty function depends on the problems.

.We've worked out the SUMT program on PC in C language (Borland C++) and found it very quick in special cases. The disadvantage is that it was sensitive on the type of penalty function and it needs a feasible starting point.

3. BACKTRACK PROGRAMMING

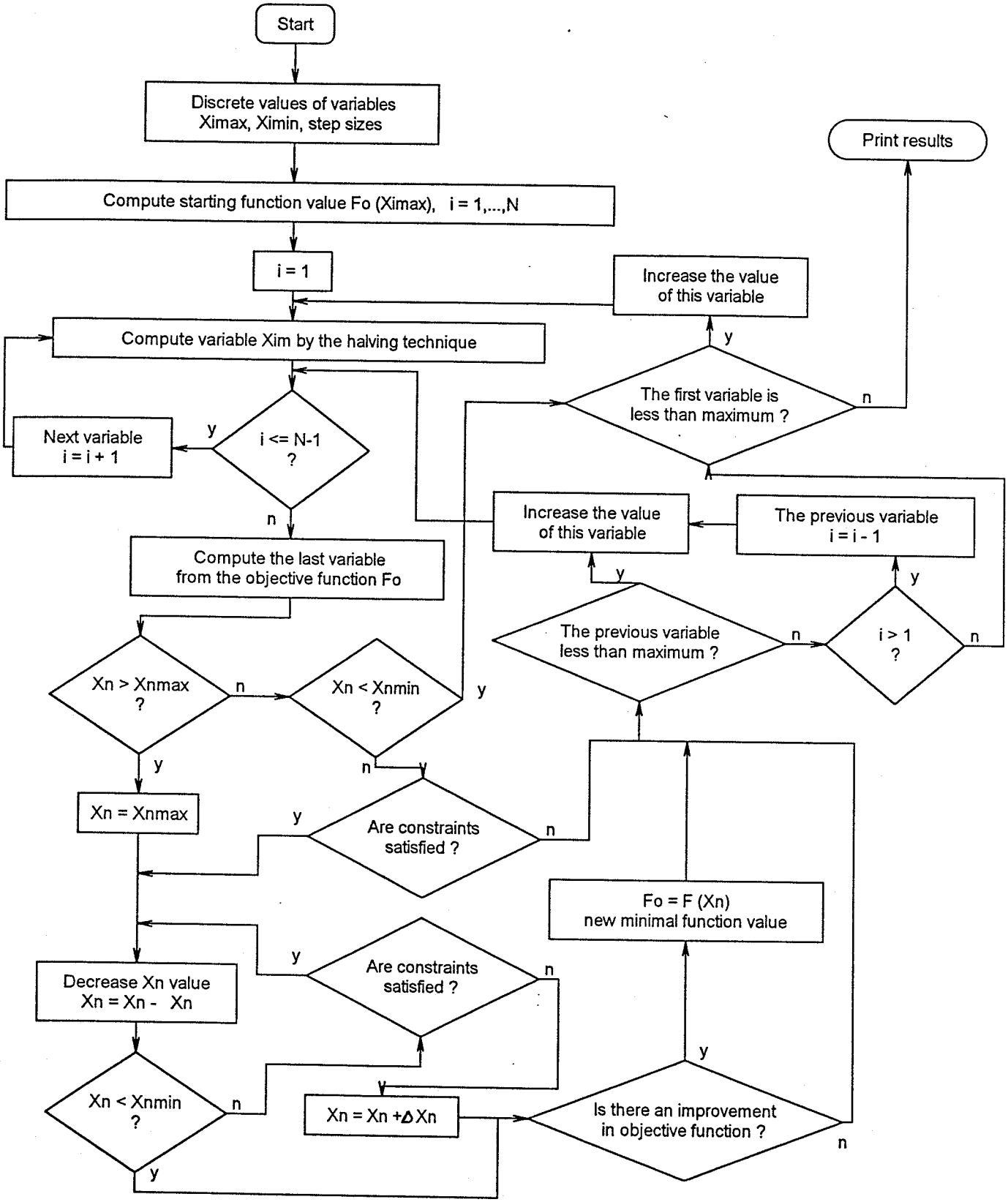
The backtrack method is a combinatorial programming technique, solves nonlinear constrained function minimization problems by a systematic search procedure. This discrete programming method can be successfully applied to optimization problems with few unknowns. The general description of backtrack can be found in Golomb and Baumert [7].

The algorithm is suitable for optimum design of structures which are characterized by monotonically increasing objective functions. Thus, the optimum solution can be found by decreasing the variables. The search may be made more efficient by using the interval halving procedure.

The variables are in a vector form $\mathbf{x} = \{x_i\}^T$ (i = 1,...,n) for which the objective function $K(\mathbf{x})$ will be a minimum and which will also satisfy the design constraints $g(\mathbf{x}) \geq 0$ (j = 1,...,P). For the variables, series of discrete values are given in an increasing order. In special cases the series may be determined by $x_{k,min}$, $x_{k,max}$ and by the constant steps Δx_k between them. The flow chart for the backtrack method is given in Fig. 2.

First a partial search is carried out for each variable and if all variations have been investigated, a backtrack is made and a new partial search is performed on the previous variable. If this variable is the first one: no variations have to be investigated (a number of backtrack has been made), then the process stops. The main phases of the calculation are as follows.

1. With a set of constant values of $x_{i,t}$ (i = 2,...,n) the minimum $x_{i,m}$ value



Flow chart of the Backtrack method

satisfying the design constraints is searched for. The search may be made more efficient by using the interval halving procedure. This method can be employed if the constraints and the objective function are monotonous from the sense of variables.

2. As in the case of the first phase, the halving process is now used with constant values, and the minimum $x_{1,m}$ value, satisfying the design constraints is then determined.

3. The least value $x_{n,m}$ is calculated from the objective function K (**x**)

where K is the value of the cost function calculated by inserting the maximum **x**-values. Regarding the $x_{n,m}$ value, three cases may occur as follows.

(3a) If we decrease x_{n-1} step-by step till it satisfies the constraints or till $x_{n,min}$, the minimal values are reached. If all variations of the x_n value have been investigated, then the program jumps to the x_{n-1} and decreases it step-by step till x satisfies the constraints or till $x_{n-1,\,min}$ are reached.

(3b) If $x_{n,m} < x_{n,1}$, we backtrack to x_{n-1}.

(3c) If $x_{n,m}$ does not satisfy the constraints, we backtrack to $x_{n-1,m}$. If the constraints are satisfied, we continue the calculation according to 3a.

The number of all possible variations is $\prod_{i=1}^{n} t_i$. However, the method investigates only a relatively small number of these. Since the efficiency of the method depends on many factors (number of unknowns, series of discrete values, position of the optimum values in the series, complexity of the cost function and/or that of the design constraints), it is difficult to predict the run time. The main disadvantage of the method is, that the runtime increases exponentially, if we increase the number of unknowns.

We've made the program in C language modifying the procedure in the sense, that originally the program depended on the number of variables. All variables were computed by the halving procedure except the last one, which was computed from the objective function. The modified version is independent from the number of variables. Advantage of the method is, that it gives discrete values, usually finds global minimum. The disadvantage is, that it is useful only for few variables.

THE METHOD OF MOVING ASYMPTOTES (MMA)

The Method of Moving Asymptotes (MMA) is a mathematical programming method which has been implemented in several large systems for structural optimization, e.g. in OPTSYS at the Aircraft division of Saab-Scania and in OASIS at ALFGAM Opt. AB.

In each iteration, a convex subproblem, which approximates the original problem, is generated and solved. An important role in the generation of these

subproblems is played by a set of parameters which influence the "curvature" of the approximations, and also act as "asymptotes" for the subproblem. By moving these asymptotes between each iteration, the convergence of the overall process can be stabilized. The original version of MMA was presented in ref [8].

Consider a structural optimization problem P1 of the following form:

P1: minimize $f_0(\mathbf{x})$

subject to: $f_i(\mathbf{x}) \le \bar{f}_i \qquad i = 1,\ldots,M$

$x_{jL} \le x_j \le x_{jU} \qquad j = 1,\ldots,N$

where:

$\mathbf{x} = \{x_1,\ldots,x_n\}$ is the vector of design variables, typically elemental sizes or shape variables,

$f_0(\mathbf{x})$ is the objective function, typically the structural weight,

$f_i(\mathbf{x}) \le \bar{f}_i$ are "behaviour constraints", typically limitations on stresses and displacements, stability, fatigue, loss factor, eigenfrequency, under different load cases,

$x_{jL} \le x_j \le x_{jU}$ are bounds, size constraints, technological constraints on the variables.

It is assumed that a discretization of the structure has been made, in accordance with the finite element method. Sometimes the considered structure is discrete already from the start. A typical example of this is a truss structure where each bar is a natural element.

P1 is often a difficult problem. One important reason for this is that the constraint functions are, in general, not explicitly given (if it is necessary to compute values by finite element technique). Instead of this, it typically holds that $f_i(\mathbf{x}) = h_j(\mathbf{x},\mathbf{u})$ where $\mathbf{h}$ is a given explicit function while the vector $\mathbf{u}$ depends (implicitly) on the vector $\mathbf{x}$ by the relation: $K(\mathbf{x})\mathbf{u} = \mathbf{p}$. Here, $K(\mathbf{x})$ is the structural stiffness matrix (which depends on $\mathbf{x}$), $\mathbf{p}$ is a vector describing the applied load, while $\mathbf{u}$ is a vector describing the displacements of the structure.

For each new $\mathbf{x}$, the stiffness matrix $K(\mathbf{x})$, which may have several thousands of rows, must be assembled before the displacement vector $\mathbf{u}$ is obtained as the solution of the (large) linear system $K(\mathbf{x})\mathbf{u} = \mathbf{p}$. This, of course, makes each new evaluation of the constraint functions expensive.

An encouraging feature of many structural optimization problems, however, is that it is possible to calculate, by a so called "semi-analytical" method, gradients of the constraint functions in an efficient way.

For a given $\mathbf{x}$, both $f_i(\mathbf{x})$ and $\nabla f_i(\mathbf{x})$ can be calculated for a cost which is not too much greater, than the cost for calculating just $f_i(\mathbf{x})$.

Because of the possibility of calculating gradients, the following iterative approach is well established for solving structural optimization problems on the form P1:

Step 0: Choose a starting point **x** and let the iteration index k be equal to 1.

Step 1: Given $\mathbf{x}^{(k)}$, calculate $f_i\ \mathbf{x}^{(k)}$ and $\nabla f_i\ (\mathbf{x}^{(k)})$ for $i=0,1,\ldots,m$

Step 2: Generate an explicit subproblem $P^{(k)}$ which approximates P1:

$P^{(k)}$: minimize $\tilde{f}_0^{(k)}(\mathbf{x})$

subject to $\tilde{f}_i^{(k)}(\mathbf{x}) \le \bar{f}_i, \qquad i=1,\ldots,M$

$x_{jL} \le x_j \le x_{jU}, \qquad j=1,\ldots,N$

The $\tilde{f}_i^{(k)}$ are explicit functions which approximate the implicit functions f_i. The choices of these approximating functions are based on the previously calculated function values and gradients. The constraints $x_{jL} \le x_j \le x_{jU}$ are often replaced by more restrictive bounds on the variables, so called "move limits", to prevent **x** from going too far away from the current iteration point **x** .

Step 3: Solve the subproblem $P^{(k)}$, with some suitable method (dependent on how the approximating functions have been chosen). Let the optimal solution of $P^{(k)}$ be the next iteration point $\mathbf{x}^{(k+1)}$ and go to step 1, with k replaced by $k+1$.

The process is interrupted when the convergence criteria are fulfilled, or simply when the user is satisfied with the current solution **x**. The central step in this approach is to choose good approximating functions. The main information available for doing this is the calculated function values and gradients (from the current iteration as well as from the previous iterations). In addition, some important properties of the considered problem may be known [9].

Each approximating function $\tilde{f}_i^{(k)}(\mathbf{x})$ is obtained by a linearization of $f_i(\mathbf{x})$ in variables of the type $1/(U_j - x_j)$ or $1/(x_j - L_j)$, where U_j L_j are the upper and lower limits for variables.

$$\tilde{f}_i^{(k)}(\mathbf{x}) = \sum_{j=1}^{n} \left(\frac{p_{ij}}{U_j - x_j} + \frac{q_{ij}}{x_j - L_j} \right) + r_i \qquad \text{for } i=1,..,m$$

If $\dfrac{\partial f_i}{\partial x_j} > 0$ at $\mathbf{x}^{(k)}$ then $p_{ij} = (U_j - x_j^{(k)})^2 . \dfrac{\partial f_i}{\partial x_j}$ and $q_{ij} = 0.$

If $\dfrac{\partial f_i}{\partial x_j} < 0$ at $\mathbf{x}^{(k)}$ then $q_{ij} = -(x^{(k)}_j - L_j)^2 . \dfrac{\partial f_i}{\partial x_j}$ and $p_{ij} = 0.$

r_i is choosen such that $\tilde{f}^{(k)}(\mathbf{x}) = f_i\ (\mathbf{x}^{(k)})$

It is also important that the subproblem $P^{(k)}$ does not become too hard to solve. It is to prefer, e.g., that the chosen approximating functions are convex.

Several methods based on the above approach (Step 0 - Step 3) have been suggested. The main difference between these methods is how the approximating functions are chosen.

The perhaps most obvious possible method is so called "Sequential Linear Programming" (SLP) where the approximating functions are chosen as the first order Taylor expansion, i.e.:

$$\tilde{f}_i^{(k)}(\mathbf{x}) = f_i(\mathbf{x}^{(k)}) + \sum_j a_{ij}(x_j - x_j^{(k)}) \quad \text{for } i = 0, 1,\ldots,M,$$

where $a_{ij} = \dfrac{\partial f_i}{\partial x_j}$, calculated at $\mathbf{x}=\mathbf{x}^{(k)}$

With these, the subproblem $P^{(k)}$ becomes a Linear Programming (LP) problem, which may be efficiently solved by the Simplex method.

In general, SLP works well if the number of active constraints at the optimal solution **x** of P1 (i.e. the number of constraints of type (2) or (3) in P1 that are satisfied as equalities at **x**) is equal to the number of design variables. Otherwise, the convergence to **x** might be very low, obtained only through the use of decreasing move limits. It is, of course, not possible in general to know in advance how many (or which) constraints are active at the optimal solution of P1.

For element sizing problems, Schmit suggested that the approximating functions for the constraints should be chosen as the first order Taylor expansion in the reciprocal element sizes ($1/x_j$):

$$\tilde{f}_i^{(k)}(\mathbf{x}) = f_i(\mathbf{x}^{(k)}) + \sum_j b_{ij}\left(\frac{1}{x_j} - \frac{1}{x_j^{(k)}}\right) \quad \text{for } i = 0, 1,\ldots,m,$$

where $b_{ij} = \dfrac{\partial f_i}{\partial\left(\frac{1}{x_j}\right)}$, calculated at $\mathbf{x} = \mathbf{x}^{(k)}$.

while the exact objective function (assumed to be the structural weight which is a linear function of the elemental sizes) is used in $P^{(k)}$,ie. $\tilde{f}_i^{(k)}(\mathbf{x}) = f_i(\mathbf{x})$.

This method is probably the most widely used method for element sizing problems, especially since Fleury suggested an efficient dual method for solving the subproblems. If the constraints are on nodal displacements and elemental stresses, while the design variables are elemental sizes and the objective function is the structural weight, then this method of linearization in reciprocal variables is much more reliable and efficient than SLP. The main reason for this is that nodal displacements and elemental stresses are more close to be linear in $1/x_j$ than in x_j.

5. FEASIBLE SEQUENTIAL QUADRATIC PROGRAMMING (FSQP)

FSQP is a set of FORTRAN subroutines for the minimization of the maximum of a set of smooth objective functions (possibly a single one) subject to general smooth constraints. If the initial guess provided by the user is infeasible for some inequality constraints or some linear equality constraint, the program first generates a feasible

point for these constraints; subsequently the successive iterates generated by FSQP all satisfy these constraints. Nonlinear equality constraints are turned into inequality constraints (to be satisfied by all iterates) and the maximum of the objective functions is replaced by an exact penalty function which penalizes nonlinear equality constraint violations only [10].

The user has the option of either requiring that the (modified) objective function decreases at each iteration after feasibility for nonlinear inequality and linear constraints have been reached (monotone line search), or requiring a decrease within at most four iterations (nonmonotone line search). The user must provide subroutines that define the objective functions and constraint functions and may either provide subroutines to compute the gradients of these functions or require that FSQP estimate them by forward finite differences. FSQP implements two algorithms based on Sequential Quadratic Programming (SQP), modified so as to generate feasible iterates. In the first one (monotone line search), a certain Armijo type arc search is used with the property that the step of one is eventually accepted, a requirement for superlinear convergence. In the second one the same effect is achieved by means of a (nonmonotone) search along a straight line.

The merit function used in both searches is the maximum of the objective functions if there is no nonlinear equality constraint. If the initial guess provided by the user is infeasible for nonlinear inequality constraints and linear constraints, FSQP first generates a point satisfying all these constraints by iterating on the problem of minimizing the maximum of these constraints. Then, using Mayne-Polak's scheme nonlinear equality constraints are turned into inequality constraints.

The resulting optimization problem therefore involves only linear constraints and nonlinear inequality constraints. Subsequently, the successive iterates generated by FSQP all satisfy these constraints. The user has the option of either requiring that the exact penalty function (the maximum value of the objective functions if without nonlinear equality constraints) decreases at each iteration after feasibility for original nonlinear inequality and linear constraints have been reached, or requiring a decrease within at most three iterations. He must provide subroutines that define the objective functions and constraint functions and may either provide subroutines to compute the gradients of these functions or require that FSQP estimate them by forward finite differences.

Thus, FSQP solves the original problem with nonlinear equality constraints by solving a modified optimization problem with only linear constraints and nonlinear inequality constraints. For the transformed problem, it implements algorithms that are described and analyzed in refinements.

These algorithms are based on a Sequential Quadratic Programming (SQP)

iteration, modified so as to generate feasible iterates. An Armijo-type line search is used to generate an initial feasible point when required. After obtaining feasibility, either (i) an Armijo-type line search may be used, yielding a monotone decrease of the objective function at each iteration; or (ii) a nonmonotone line search and analyzed, may be selected, forcing a decrease of the objective function within at most four iterations. In the monotone line search scheme, the SQP direction is first tilted if nonlinear constraints are present to yield a feasible direction, then possibly "bent" to ensure that close to a solution the step of one is accepted, a requirement for superlinear convergence. The nonmonotone line search scheme achieves superlinear convergence with no bending of the search direction, thus avoiding function evaluations at auxiliary points and subsequent solution of an additional quadratic program. After turning nonlinear equality constraints into inequality constraints, these algorithms are used directly to solve the modified problems. For the solution of the quadratic programming subproblems, FSQP is set up to call QLD which is provided with the FSQP distribution for the user's convenience.

User-Supplied Subroutines

At least two of the following four Fortran 77 subroutines, must be provided by the user in order to define the problem. The name of all four routines can be changed at the user's will, as they are passed as arguments to FSQP.

Subroutine **obj** to be provided by the user, computes the value of the objective functions.

```
      subroutine obj(nparam,j,x,fj)
      integer nparam,j
      double precision x(nparam),fj
c     for given j, assign to fj the value of the jth objective
c     evaluated at x
      return
      end
```

nparam~dimension of {**x**}, j~number of the objective to be computed, **x**~current iterate, fj~value of the j th objective function at {**x**}.

The subroutine **constr**, to be provided by the user, computes the value of the constraints. If there are no constraints, a (dummy) subroutine must be provided anyway due to Fortran 77 compiling requirement.

```
      subroutine constr(nparam,j,x,gj)
      integer nparam,j
      double precision x(nparam),gj
c     for given j, assign to gj the value of the jth constraint
c     evaluated at x
```

```
      return
      end
```

nparam~dimension of {**x**}, j~number of the constraint to be computed, x~current iterate, gj~value of the j th constraint at {**x**}.

The order of the constraints must be as follows. First the {nineqn} (possibly zero) nonlinear inequality constraints, then the {nineq-nineqn} (possibly zero) linear inequality constraints, finally, the {neqn} (possibly zero) nonlinear equality constraints, followed by the {neq-neqn} (possibly zero) linear equality constraints.

The subroutine {**gradob**} computes the gradients of the objective functions. The user may omit to provide this routine and require that forward finite difference approximation be used by FSQP via calling {grobfd} instead~(see argument {gradob} of FSQP.

The specification of {gradob}is as follows

```
      subroutine gradob(nparam,j,x,gradfj,dummy)
      integer nparam,j
      double precision x(nparam),gradfj(nparam)
      double precision dummy
      external dummy
c     assign to gradfj the gradient of the jth objective function
c     evaluated at x
      return
      end
```

nparam~dimension of {**x**}, j~number of objective for which gradient is to be computed, x~current iterate, gradfj~gradient of the j th objective function at **x**.

The subroutine {**gradcn**} computes the gradients of the constraints. The user may omit to provide

this routine and require that forward finite difference approximation be used by FSQP via calling grcnfd instead (see argument gradcn} of FSQP

The specification of gradcn is as follows

```
      subroutine gradcn(nparam,j,x,gradgj,dummy)
      integer nparam,j
      double precision x(nparam),gradgj(nparam)
      double precision dummy
      external dummy
c     assign to gradgj the gradient of the jth constraint
c     evaluated at x
      return
      end
```

nparam~dimension of {**x**}, j~number of constraint for which gradient is to be computed, x~current iterate, gradgj~gradient of the j th constraint evaluated at {**x**}.

Organization of FSQPD and Main Subroutines

{**mainorg**} FSQP first checks for inconsistencies of input parameters using the subroutine {**check**}. It then checks if the starting point given by the user satisfies the linear constraints and if not, generates a point satisfying these constraints using subroutine {**initpt**}. It then calls FSQPD1 for generating a point satisfying linear and nonlinear inequality constraints. Finally, it attempts to find a point satisfying the optimality condition using again FSQPD1.

FSQPD1 uses the following subroutines:

{**dir**} compute various directions d_K^o, d_1^o and d_K.

{**step**} compute a step size along a certain search direction.

{**hesian**} Perform the Hessian matrix updating.

{**out**} Print the output for {iprint=1} or {iprint}=2.

{**grobfd**} (optional)~compute the gradient of an objective function by forward finite differences.

{**grcnfd**} (optional)~compute the gradient of a constraint by forward finite differences.

{**nineqn**} the number of nonlinear constraints,

{**ncallf**} the total number of evaluations of the objective function,

{**ncallg**} the total number of evaluations of the (scalar) nonlinear constraint functions,

{**iter**} the total number of iterations,

{**objective**} the final value of the objective,

{**ktnorm**} the norm of Kuhn-Tucker vector at the final iterate,

{**eps**} the norm requirement of the Kuhn-Tucker vector,

{**SCV**} the sum of feasibility violation of linear constraints.

Programming Tips

The order in which FSQP evaluates the various objectives and constraints during the line search varies from iteration to iteration, as the functions deemed more likely to cause rejection of the trial steps are evaluated first. On the other hand, in many applications, it is far more efficient to evaluate all (or at least more than one) of the objectives and constraints concurrently, as they are all obtained as by-products of expensive simulations (e.g., involving finite element computation). This situation can be accommodated as follows. Whenever a function evaluation has been performed, store in a common block the value of {**x**} and the corresponding values of all objectives and constraints (alternatively, the values of all "simulation outputs"). Then, whenever a function evaluation is requested by FSQP, first check whether the

same value of {**x**} has just been used and, if so, entirely bypass the expensive simulation. Note that, if gradients are computed by finite differences, it will be necessary to save the past {nparam}+1 values of {**x**} and of the corresponding objective/constraint values.

It is important to keep in mind some limitations of FSQP.

First, similar to most codes targeted at smooth problems, it is likely to encounter difficulties when confronted to nonsmooth functions such as, for example, functions involving matrix eigenvalues. Second, because FSQP generates feasible iterates, it may be slow if the feasible set is very "thin" or oddly shaped.

Third, concerning equality constraints, if $h_j(\mathbf{x}) \geq 0$ for all $\mathbf{x}$ and if $h_j(\mathbf{x}_0)=0$ for some j at the initial point $\mathbf{x}_0$, the interior of the feasible set defined by $h_j(\mathbf{x}) \leq 0$ for such j is empty. This may cause difficulties for FSQP because, in FSQPD, $h_j(\mathbf{x})=0$ is directly turned into $h_j(\mathbf{x}) \leq 0$ for such j.

The user is advised to either give an initial point that is infeasible for all nonlinear equality constraints or change the sign of h_j so that $h_j(\mathbf{x})<0$ can be achieved at some point for all such nonlinear equality constraint.

A common failure mode for FSQP, corresponding to {inform}=5 or 6, is that of the QP solver in constructing {d0} or {d1}. This is often due to linear dependence (or almost dependence) of gradients of equality constraints or active inequality constraints. Sometimes this problem can be circumvented by making use of a more robust (but likely slower) QP solver. The developers have designed an interface, available upon request, that allows the user to use QPSOL instead of QLD. The user may also want to check the Jacobean matrix and identify which constraints are the culprit. Eliminating redundant constraints or formulating the constraints differently (without changing the feasible set) may then be the way to go.

Finally, when FSQP fails in the line search ({inform}=4), it is typically due to inaccurate computation of the search direction. Two possible reasons are: (i) insufficient accuracy of the QP solver; again, it may be appropriate to substitute a different QP solver. (ii) insufficient accuracy of gradient computation, e.g., when gradients are computed by finite differences. A remedy may be to provide analytical gradients or, more astutely, to resort to "automatic differentiation".

6. REFERENCES

1. Jármai,K.: **Single- and multicriteria optimization as a tool of decision support system.** *Computers in Industry,* Elsevier Applied Science Publishers, 1989, Vol.11, No.3. p.249-266.
2. Farkas,J.: **Optimum design of steel structures**. *Akademiai Kiadó*, Budapest,

Ellis Horwood Ltd. Chichester, 1984.

3. Farkas,J.,Jármai,K.: **Minimum cost design of laterally loaded welded rectangular cellular plates.** *Structural Optimization '93, The World Congress on Optimal Design of Structural Systems*, Rio de Janeiro, Aug. 2-6. 1993. Proceedings Vol. 1. p.205-212.
4. Jármai,K.: **The efficiency of the optimization techniques in the economic design of steel structures.** *15th IFIP conference on System Modelling and Optimization*, Zürich, September 2-6. 1991. Proceedings p. 504-505.
5. Jármai,K.: **Decision support system on IBM PC for design of economic steel structures, applied to crane girders.** *Thin-Walled Structures*, Elsevier Applied Science Publishers, 1990, Vol.10, p.143-159.
6. Fiacco,A.V.,McCormick.G.P.: **Nonlinear programming: sequential unconstrained minimization techniques.** New York. Wiley. 1986.
7. Golomb,S.W.,Baumert,L.D.: **Backtrack programming**. *J.Assoc. Computing Machinery*, 1965. Vol.12. p.516-524.
8. Svanberg, K.: **The method of moving asymptotes.** *Int.J. Num. Meth. in Engineering*, Vol.24, p.359-373, 1987.
9. Svanberg, K.: **MMA with some extensions**, *Optimization of Large Structural Systems,* Lecture Notes from the NATO/DFG ASI, Berchtesgaden, Sep-Oct 1991.
10. Zhou,J.L.,Tits,A.: **User's guide for FSQP Version 3.0: a Fortran code for solving optimization problems**. *Techn. Report SRC-TR-90-60 rlt.,*Systems Research Center, University of Maryland, College Park, 1992.

7. ACKNOWLEDGEMENTS

The author would like to thank Andre L. Tits and Jian L. Zhou Univ. of Maryland for the possibility of using the CFSQP algorithm.

This work received support from the Hungarian Fund for Scientific Research Grant OTKA T 4407.

Thanks to Gyula Szikszai, PhD student for his work in writing the C version of SUMT and Backtrack.