

*Solving Optimization Problems By
the Optimization Program
INVERSE*

(FOR VERSION 3.11)

Igor Grešovnik

Ljubljana, 6 October, 2005

Contents:

6. Optimization And Inverse Analyses 3

6.1 Definition of Optimization Problem and its Solution3

6.1.1 Basic Terms3

6.1.2 Definition of the Problem in the Command file3

6.1.3 Defining the Direct Analysis4

6.1.4 Implicit Gradient Calculation5

6.2 Optimization algorithms.....8

6.2.1 `optfsqp1 { numob numnonineq numlinineq numnoneq numlineq eps epseqn maxit grad { initial } { lowbound } { upbound } }`8

6.2.2 `inverse { methodspec params }`.....9

6.3 Auxiliary tools11

6.3.1 Testing the Direct Analysis.....11

6.3.2 Tabulating Functions11

6.4 Approximation tools14

6.4.1 Smooth approximation.....14

6. OPTIMIZATION AND INVERSE ANALYSES

6.1 Definition of Optimization Problem and its Solution

6.1.1 Basic Terms

We state the optimization problem quite generally as

$$\begin{array}{ll}
 \text{minimise} & f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n \\
 \text{subject to} & c_i(\mathbf{x}) = 0, \quad i \in E \\
 \text{and} & c_j(\mathbf{x}) \geq 0, \quad j \in I, \\
 \text{where} & l_i \leq x_i \leq u_i, \quad j = 1, 2, \dots, n,
 \end{array} \tag{6.1}$$

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 equality and inequality constraints, respectively. Sometimes the algorithm can in addition take the advantage of explicitly stated eventual linear constraint functions, such as in the case of fsqp.

6.1.2 Definition of the Problem in the Command file

The optimization problem and its solution procedure must be defined in the shell command file, which is interpreted by the interpreter.

The command file typically consists of three parts: the preparation part, the analysis block and the final action part. In the **preparation part** variables are typically allocated, data initialized and functions defined for use at a later time. The **analysis block** defines how direct analysis is performed. This block is interpreted every time the direct analysis is performed, either run from within some algorithm or as a consequence of user request. In the **action part** the optimization algorithms that lead to problem solution are

run. Test analyses at different parameter sets or some other tests (e.g. tabulating of the objective function) can also be run in this part.

The preparation part and analysis block can usually be swapped. Individual allocations and definitions can be performed right before they are used, although the command file usually looks clearer if this is done in one place. The user must be careful about putting definitions and allocations in the analysis block because this block is iteratively interpreted. What concerns tasks that do not need to be performed in every analysis, it is better if they are invoked outside the analysis block so that they are performed only once.

6.1.3 Defining the Direct Analysis

The term “direct analysis” refers to the evaluation of the objective and constraint functions and possibly their gradients at a given set of optimization parameters. User defines how the direct analysis is performed in the analysis block of the shell command file. This is the block of code in the argument block of the **analysis** command, i.e. within the curly brackets that follow this command.

The analysis block is interpreted by the shell interpreter every time the direct analysis is performed. Direct analysis can be called by an optimization algorithm or by some other function invoked by the interpreter. Typical examples are tabulating functions or the **analyse** function for performing test direct analyses.

Data transfer between the direct analyses and the functions that invoke them is implemented through global shell variables with a pre-defined meaning. The shell takes care that the current set of optimization parameters is always in the vector variable **parammom** when the direct analysis is invoked. In the analysis block the user can therefore obtain parameter values from this variable using the interpreter and expression evaluator functions for accessing variables. In the similar way it is expected that after the direct analysis is performed its results will appear in the appropriate global shell variables. User must take care of that in the analysis block by storing results in these variables. For example, value of the objective function must appear in scalar variable **objectivemom**, values of constraint functions must appear in scalar variable **constraintmom**, objective function gradient in vector variable **gradobjectivemom**, gradients of constraint functions in vector variable **gradconstraintmom**, simulated measurements (in the case of inverse analyses) in vector variable **measmom**, etc. These variables with a pre-defined meaning are treated just like other user-defined variables and the same functions can be used for their manipulation. There are however some particularities in behaviour of variable manipulation functions in the case of variables with a pre-defined meaning. Rules are more or less the same, there is only some additional intelligence incorporated, which enables user not to specify dimensions that are already known to the shell. For details, see the “Shell Variables with a Pre-defined Meaning” chapter of the “User Defined Variables in the Optimization Shell *Inverse*” manual.

Within the analysis block the user is expected to run a numerical simulation with parameters found in vector **parammom**, combine its results to evaluate the requested function values (objective and constraint functions and their derivatives) and store these results in the appropriate variables with a pre-defined meaning. This can include a number of sub-tasks, for example parameter dependent domain transformation in the case of shape optimization problems (this is reduced to finite element mesh transformation in some cases). Interfacing the simulation programme, i.e. changing input data according to parameter values, running the programme and obtaining results, is usually an important issue, as well as combining of these partial results according to problem definition in order to derive final results. Several modules of the shell provide tools for performing such sub-task, and the user can combine these tools using the file interpreter according to the character of problems that are being solved.

All tools and algorithms of the shell are accessed through the shell file interpreter. This, together with the expression evaluator (the “calculator”) and interpreter flow control functions, gives the user a great flexibility at defining different optimization problems and also the solution procedures. The shell is in the first place designed for use with simulation programmes. For test purposes, however, the user can define optimization problems in such way that evaluation of objective and other functions do not include numerical simulation. The functions are in this case defined analytically using shell variables and expression evaluator. Such examples can be found in the directory of training examples (subdirectory “opt”).

6.1.4 Implicit Gradient Calculation

Some optimization algorithms need gradients of the objective and constraint functions beside their values. Most commonly, these should be calculated in the *analysis* block and stored in the appropriate pre-defined variables (e.g. *gradobjectivemom* or *gradconstraintmom*, see the manual on variables, chapter on pre-defined variables). This essentially means that the algorithm for calculation of the objective and constraint functions must be differentiated with respect to the design parameters. This is sometimes difficult to achieve, especially when some numerical simulation is used as a “black box” and the user does not have access to its source code.

The derivatives can always be obtained numerically e.g.¹ by sequentially perturbing values of individual parameters, calculating the functions at perturbed parameters and dividing the difference with respect to the response at original parameters by the perturbation (i.e. difference in parameter value or step size). This can be eventually programmed within the analysis block of the interpreter. Doing so, however, can significantly reduce the clearness and readability of the analysis block.

¹ This scheme is called the finite difference method. There are also more complex schemes for numerical derivative calculation, all of which include repeating calculation of function values at a number of perturbed parameters, but differ significantly in sampling strategies and underlying mathematics. Description of these schemes exceeds the scope of this manual.

The tools have been providing for automatic implicit numerical calculation of the derivatives. When implicit derivative calculation is switched on, on any request for performing the analysis at given parameter values, the (non-derivative) analysis is actually performed with the original and perturbed parameter values. Numerical approximation of gradients of the objective and constraint functions is calculated on the basis of the results and stored to the appropriate pre-defined variables (most commonly *gradobjectivemom* and *gradconstraintmom*) together with function values at the original parameters of the request (*objectivemom* and *constraintmom* are commonly used to store these).

The interpreter functions for providing implicit numerical gradient calculation are described below.

6.1.4.1 *analysisnumgradfvec* { *stepvec* }

Installs the implicit numerical calculation of gradients of the objective and constraint functions (if defined) with respect to optimization parameters by the forward difference scheme. This applies to the functions that are calculated by the direct analysis *direct analysis*, which includes interpretation of the *analysis* block. *stepvec* must be a vector value argument that specifies the step size for each parameter. Its dimension must therefore be the same as the number of parameters (i.e. the dimension of the pre-defined vector *parammom*). If *stepvec* is not specified, then the default step size (10^{-4}) is taken for derivation with respect to all parameters. It is usually a very bad idea not to specify the step sizes because the accuracy of the derivatives depend essentially on it, and the optimal step size may vary drastically from case to case since it depends on scaling of the design parameters and on the level of noise of the differentiated functions.

After the call to the function, every direct analysis at a given set of parameters is replaced by a number of plain analyses. The first one is performed at the requested parameters and n others are performed at the parameter sets in which one parameter is perturbed by the appropriate step size as specified by *stepvec*, n being the number of parameters. After this, the function values calculated with the requested parameter values are stored as usual (e.g. in the pre-defined variables *objectivemom* or/and *constraintmom*). In addition, numerical approximations to the parameter gradients of these values are calculated and stored at the appropriate place² (e.g. in the pre-defined variables *gradobjectivemom* and *gradonstraintmom*). See the manual on variables, chapter on pre-defined variables for more details regarding the meaning of specific pre-defined variables and rules for their manipulation.

The accuracy of the numerically calculated derivatives crucially depends on the step size. The derivative calculation is mathematically exact for linear functions, and therefore there are two sources of error. The first one is because the function is normally not linear and this contributes larger errors where the step size gets large and the function deviates more from the linear model. The second source is due to the noise in the function value. If there is no other source of noise, at least the function values are inexact because

² This would normally be done explicitly by the appropriate interpreter code in the *analysis* block.

of finite precision that is used for all computer operations. Errors in calculated derivatives that come from this source are amplified when the step size is reduced, and die away when the step size gets large compared to the amplitude of noise. Therefore, there exists an optimal step size which is large enough with respect to noise amplitude and yet small enough that the function is adequately approximated by a linear model within the step size. The user should provide the step size that is not necessarily optimal, but is a good compromise for both sources of error. When it is hard to estimate the level of noise, the step size should be taken that is a bit smaller than the tolerance for the optimum, and the tolerance should be set rather conservatively in order to avoid failure of algorithms due to excessive noise.

6.1.4.2 analysisnumgradfd { *stepsize* }

Does the same as **analysisnumgradfvec**, except that the step size for all parameters are set equal to *stepsize*, which is a scalar value argument. If *stepsize* is not specified then a default step size (10^{-4}) is taken. However, it is usually a very bad idea not to specify the step size because the accuracy of the derivatives depend essentially on it, and the optimal step size may vary drastically from case to case since it depends on scaling of the design parameters and on the level of noise of the differentiated functions.

6.1.4.3 analysisplain { }

Cancels the implicit numerical differentiation of the objective function (and constraint functions if defined) and places instead the original analysis function, which performs the direct analysis (including interpretation of the *analysis* block) at only one set of design parameters.

6.1.4.4 analysisnumgradprn { *doprn* }

If the counter value argument *doprn* is different than 0 then reporting on gradient calculation is switched on, which can be used for control.

6.2 Optimization algorithms

6.2.1 `optfsqp1` { *numob numnonineq numlineq numnoneq numlineq* *eps epseqn maxit grad { initial } { lowbound } { upbound } }*

Performs the *fsqp* (feasible sequential quadratic programming) optimization algorithm, which is the basic and most powerful optimization algorithm of the shell.

numob is the number of objective functions (usually one), *numnonineq* the number of non-linear inequality constraints, *numlineq* the number of linear inequality constraints, *numnoneq* the number of non-linear equality constraints and *numlineq* the number of linear equality constraints. *eps* is the final norm requirement for the Newton direction and *epseqn* maximum violation of nonlinear equality constraints at an optimal point. Both criteria must be satisfied to stop the algorithm (the second one is in effect only if there are equality constraints). *maxit* is the maximum number of iterations. *grad* specifies if gradients are provided by direct analyses (1) or should be calculated numerically (0). *initial* is the initial guess and *lowbound* and *upbound* are vectors of lower and upper bounds on parameters. All three vectors must be in curly brackets. The components which are not specified in the *lowbound* or *upbound* vectors are not bounded below or above, respectively. Dimensions must be specified for all three vectors, and all components must be specified for *initial*.

Warning:

Inequality constraints are stated by an opposite sign as in (6.1), namely

$$c_j(\mathbf{x}) \leq 0, j \in I \quad (6.2)$$

In variables which hold values or derivatives of constraint functions, these must appear in the appropriate order, the same as in the argument block of the function. First must be non-linear inequality constraints, then linear inequality constraints, then non-linear equality constraints and finally linear equality constraints (if any of these are specified, of course).

Remarks:

See introductory section for how the problem should be defined! You can also take a look at `inquick2.pdf`, which can be obtained at

<http://www.c3m.si/inverse/doc/other/index.html> .

A detailed description of the `fsqp` algorithm can be found at

<http://www.isr.umd.edu/Labs/CACSE/FSQP/fsqp.html> .

6.2.2 Optsimplex { *tol maxit startguess* }

Performs unconstrained minimization by the Nelder-Mead simplex method. Scalar argument *tol* is a tolerance, counter argument *maxit* is maximal number of iterations and matrix argument *startguess* is a matrix whose rows are co-ordinates of apices of the initial simplex. One should take care that *startguess* represents a simplex with non-zero volume, which means that all vectors along the edges of the simplex joining in a given common apex are linearly independent.

6.2.3 inverse { *methodspec params* }

This function performs different types of optimization algorithm. *methodspec* determines which optimization algorithm is used. It is followed by parameter specifications *params*, which are dependent on the type of algorithm used.

methodspec begins either with string *1d* or *nd*, indicating whether we will solve one-dimensional (one parameter) or multi-dimensional problems, respectively. The second part of *methodspec* is a string that specifies the method more precisely. Method and parameter specifications for different methods are described below.

6.2.3.1 inverse { *1d parabolic x0 step0 tol maxitbrac maxit* }

Performs minimization of the objective function of one parameter. Successive three points quadratic approximations of the objective function are used where possible. The minimization is performed in two steps.

In the first step, the interval containing a local minimum is searched for. This is achieved by searching for combination of three points such that the middle point has the lowest value of the objective function. The first point is given by the user (*x0*), and the second two points are obtained by adding the initial bracketing step (*step0*) to that point once and twice, respectively. Then the three points are moved, if necessary, until the bracketing condition is reached (i.e. the middle point has the lowest value of the objective function).

In the second step, the bracketing interval that contains the three bracketing points is narrowed in such a way that the bracketing condition remains satisfied. In each iteration a new point is added in the larger of the two intervals defined by the three bracketing points. Among four points we obtain this way, those three which satisfy the bracketing condition and define the smallest interval are kept for the next iteration. The point that is added is usually chosen by finding the minimum of quadratic parabola that through the current bracketing points. This is not done if one of the two intervals becomes much smaller, since in such cases successive quadratic approximations can converge slowly.

$x0$ is the initial point, and $step0$ is the initial step of the bracketing stage. The second and the third point of the initial bracketing triple are obtained by adding $step0$ to $x0$ once and twice, respectively. tol is the tolerance for function minimum. The algorithm terminates when the difference between the highest and the lowest value of the objective function in the current three bracketing points is below tol . $maxitbrac$ is the maximal allowed number of iterations at searching for bracketing triple. If the algorithm fails to find the three points satisfying the bracketing condition in $maxitbrac$ iterations, it terminates and reports an error. $maxit$ is the maximal allowed number of iterations in the second stage.

6.2.3.2 **inverse** { *nd simplex tol maxit startguess* }

Performs minimization of the objective function by simplex method. Apices of a simplex is successively moved in such a way that the simplex moves and shrinks toward function minimum. Simplex is a geometrical body in an n -dimensional space that has $n+1$ dimensions.

tol is tolerance for function minimum. The algorithm terminates when the difference between the greatest and the least value of the objective function in simplex apices becomes less than tol . $maxit$ is the maximal allowed number of iterations. If the minimum is no reached in $maxit$ iterations, the algorithm terminates and reports an error. $startguess$ is the starting guess, containing the initial simplex. This must be a matrix of dimensions $numparam+1 \times numparam$. Rows of this matrix represent apices of the initial simplex.

Warning:

Use `optsimplex` instead of this command!

inverse is becoming an obsolete command and will be replaced by some other commands in the future. However, the command will remain implemented in the programme and will behave in the same way through a lot of future versions.

6.3 Auxiliary tools

6.3.1 Testing the Direct Analysis

6.3.1.1 analyse { }

Performs the direct analysis at parameters *parammom*. The pre-defined vector *parammom* must therefore be set before this function is called. The values of the pre-defined global variables that hold analysis results are printed to the programme's standard output and output file.

6.3.2 Tabulating Functions

6.3.2.1 tab1d { *kindspec point0 point1 numpt factor printparam printmeas* }

Runs a set of direct analyses along a line in the parameter space and prints the requested results to the programme's standard output and output file. *kindspec* is a string that specifies what kind of table of direct analyses should be made and can be either *noncent* or *cent*. *noncent* means that *numpt* direct analyses with sampling points on a line between *point0* and *point1* will be performed, while *cent* means that sampling points will lie on the line whose centre is *point0* and one of its two endpoints is *point1*. *point0* is the base point and *point1* the final point in the parameter space. Both points must be specified as vectors of parameters. *numpt* specifies the number of sampling points. *factor* is the factor by which the distances between successive sampling points are extended. If *factor* is 1 then points will be equidistantly distributed, if it is different than 1 then the distances between successive points will decrease or increase from *point0* towards *point1*. *printparam* and *printmeas* specify whether parameters and measurements should be printed, respectively; values different than zero indicate that the appropriate quantities should be printed.

example: `tab1d { cent 4 {0 2 3 4} 4 {2 0 4 3} 8 1 0 0 }`

- *tab1d* by Domen Cukjati.

6.3.2.2 **tab2d** { *kindspec point0 point1 numpt1 factor1 point2 numpt2 factor2 printparam printmeas* }

Runs *numpt1* x *numpt2* direct analyses with sampling points being nodes of a planar grid of points, lying on a parallelogram in the parameter space, and prints the requested results to the programme's standard output and output file. *kindspec* is a string that specifies what kind of table of direct analyses should be made and can be either *noncent* or *cent*. **noncent** means that sampling points will lie in the parallelogram defined by two vectors which are defined by basic point *point0* and final points *point1* and *point2*. **cent** on the other hand means that parallelogram is also defined by the same points, but basic point *point0* lies in the middle of the parallelogram. Parallelogram is four times bigger in this case. All three points should be specified as vectors of parameters. *numpt1* specifies the number of sampling points in the first direction and *numpt2* in the second one. *factor1* is the factor by which the distance between successive sampling points in the first direction is extended and *factor2* is for the second direction. If any factor is equal to 1, points will be equidistantly distributed. *printparam* and *printmeas* specify whether parameters and measurements should be printed, respectively; values different than zero indicate that the appropriate quantities should be printed.

example: `tab2d { noncent 4 {0 0 3 4} 4 {1 0 3 4} 5 2 4 {0 1 3 4} 5 1 1 1 }`

- *tab2d* by Domen Cukjati.

6.3.2.3 **linetab** { *kindspec args* }

Runs a set of direct analyses along a line in the parameter space and prints the requested results to the programme's standard output and output file. *kindspec* is a string that specifies what kind of table of direct analyses should be made. The remaining arguments *args* specify the line along which the table is made, number of points, etc. *kindspec* can be either *lin* or *exp*. The meaning of the remaining arguments for different types of table is explained below.

6.3.2.3.1 **linetab** { *lin numpt ppar pmeas point1 point21* }

Runs *numpt* direct analyses with sampling points equidistantly distributed along the straight line between *point1* and *point2*. *ppar* and *pmeas* specify whether the parameters and measurements should be printed in table lines, respectively (values different than zero indicate that the appropriate quantities should be printed). In any case, before the print-out of the table values, parameters are printed that correspond to the sampling points along the line. Points are indexed by proportional factors from 0 (for *point1*) to 1 (for *point2*).

6.3.2.3.2 **linetab** { *exp numpt pppar pmeas factor point1 point2* }

Runs *numpt* direct analyses with sampling points non-equidistantly distributed along the straight line between *point1* and *point2*. *factor* is the factor for which the distance between the length of the successive sampling interval is extended.

ppar and *pmeas* specify whether the parameters and measurements should be printed in table lines, respectively (values different than zero indicate that the appropriate quantities should be printed). In any case, before the print-out of the table values, parameters are printed that correspond to the sampling points along the line. Points are indexed by proportional factors from 0 (for *point1*) to 1 (for *point2*).

Older tabulating functions:

6.3.2.4 *tab1d0* { *which val1 val2 numpt pobjective pmeas* }

Runs a set of *numpt* direct analyses so that parameter *which* is varied between *val1* and *val2* with a constant step. *pobjective* and *pmeas* specify if the objective function and measurements should be printed, respectively (values different than zero indicate that the appropriate quantities should be printed).

At the sampling points, parameters other than *which* are taken from the pre-defined vector *parammom*.

6.3.2.5 *tab2d0* { *whichx x1 x2 numptx whichy y1 y2 numpty pobjective pmeas* }

Runs a set of *numptx*numpty* direct analyses organised in a two-dimensional table so that parameters *whichx* and *whichy* are changed. Parameter *whichx* is varied between *x1* and *x2* with *numptx* equidistant sampling values, and parameter *whichy* is varied between *y1* and *y2* with *numpty* equidistant sampling values. *pobjective* and *pmeas* specify if the objective function and measurements should be printed, respectively (values different than zero indicate that the appropriate quantities should be printed).

At the sampling points, parameters other than *which* are taken from the pre-defined vector *parammom*.

6.3.2.6 *tabline0* { *kindspec args* }

Runs a set of direct analyses along a line in the parameter space and prints the requested results to the programme's standard output and output file. *kindspec* is a string

that specifies what kind of table of direct analyses should be made. The remaining arguments *args* specify the line along which the table is made, number of points, etc. *kindspec* can be either *lin* or *exp*. The meaning of the remaining arguments for both possibilities is explained below.

6.3.2.6.1 **tabline0** { *lin numpt pobjective pmeas point1 point2* }

Runs *numpt* direct analyses with sampling points equidistantly distributed along the straight line between *point1* and *point2*. *pobjective* and *pmeas* specify if the objective function and measurements should be printed, respectively (values different than zero indicate that the appropriate quantities should be printed).

6.3.2.6.2 **tabline0** { *exp numpt factor pobjective pmeas point1 point2* }

Runs *numpt* direct analyses with sampling points non-equidistantly distributed along the straight line between *point1* and *point2*. *pobjective* and *pmeas* specify if the objective function and measurements should be printed, respectively (values different than zero indicate that the appropriate quantities should be printed). *factor* is the factor for which the distance between the length of the following sampling interval is extended.

6.4 Approximation tools

6.4.1 Smooth approximation

6.4.1.1 **smoothapproxsimp** { *samples rweight point which valspec <gradspec>* }

Calculates an approximation of a sampled function. The moving least squares method is applied, which approximates the function locally by low order (square in this case) polynomial. Coefficients of approximation are not constant, but depend on the position of the point. This is so because weights assigned to sampling points for calculating the least squares approximation depend on the relative position of the point of approximation with respect to these sampling points.

Arguments:

samples: Matrix argument – sampled data. Each matrix row corresponds to a sampling point, and columns of a row contain the co-ordinates of the sampling points followed by sampled values (more than one functions may be samples).

rweight: Vector argument that contains effective radii of the weights in individual co-ordinate directions. Weight corresponding to a sampling point fall from 1 (size of the weight exactly in the sampling point) to $1/e$ at the distance r_i from a sampling point in the co-ordinate direction i , where r_i is the component i of *rweight*.

point: Vector argument – point of approximation.

which: counter argument, specifies which sampled function should be approximated (usually it is 1, meaning the first function).

valspec: Scalar element specification, specifies an element of a scalar variable to which the approximated value is stored.

gradspec: Vector element specification. If *gradspec* is specified then gradient of the approximation is also calculated and stored to *gradspec*.

Example:

```
Setmatrix {samp 100 3 {} }
setvector {point 3 {10, 1.3, 55 }}
setvector {rweight 3 {0.5, 0.5, 0.5}}
. . . *{ sampling functions }
setcounter {which 1}
setcounter {val 0}
smoothapproxsimp{samp rweight point #which val[] }
```