Modest

User's Guide

September 25, 2006

by H. Haario

ProfMath Oy
Maljatie 8
00430 Helsinki
Finland

Phone: +(358)-400-814092

NOTICE

The information contained in this document is subject to change without notice.

ProfMath MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. ProfMath shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Modest User's Guide
September 25, 2006

Modest Tutorial

Modest

1 Installation

This software consists of two Fortran libraries **modestr.lib** and **solvers.lib** and the executable **nmlio** for namelist input interpretation. In addition, a number of demonstration and testing examples are given. Matlab functions are given for graphical display of the results in the Matlab environment.

It is advisable to organize the files in different directories, for example in PC environment as

```
c:\modest    modestr.lib
             solvers.lib
             nmlio.exe

c:\modest\mfiles    all the Matlab m-files
c:\modest\box       all the 'box'  examples
c:\modest\impl      all the 'impl' examples
```

The example collections also contain ready made routines for compiling, linking and running the examples, as well as for display in Matlab. If used with Matlab, remember to add the directory of the m–files to the Matlab path.

2 Overview

The Modest (MOdel ESTimation) program package has been designed for parameter estimation of mechanistic mathematical models as well as for experimental design. The software has gradually evolved as a by–product of real industrial modeling work over several years. During that period a great number of modeling tasks of various types has been solved. We might say that the present version of Modest is a parametrization of that work: each feature in the software is there because it has been found necessary or useful in some modeling project. On the other hand, all the modeling tasks, that previously required tailor made solutions in each project, can now be solved in a unified manner in the Modest environment.

The code is written in standard Fortran 77. It is portable and tested in, e.g., SUN, HP, IBM and CONVEX Unix environments and in VAX VMS. The standard working environment of the authors is, however, a 486/66 PC with the Microsoft Power Station Fortran compiler. There are no limitations – except for the memory available in the machine used – for the maximum size of problems that can be handled.

The software is able to deal with explicit algebraic, implicit algebraic (systems of nonlinear equations) and ordinary differential equations. As the standard way to handle PDE systems we use the Numerical Method of Lines, which transforms a PDE system to a number of ODE components. In addition, any model with a solver provided may be dealt with as an algebraic system.

The software consists of two parts: the IO interphase and the core Modest executable. The input definitions may be given in two different ways: using a Fortran namelist or a Windows interphase (in PC). Both ways create template subroutines for the user written model code. These are compiled and linked with the Modest core library.

The core executable gives a numerical ascii file for output. The results may be viewed in any graphical environment. Macros are available for the Windows interphase, Matlab and Excel.

As the basic numerical tools we use well tested public domain software (Blas, Linpack, Eispack, LSODE).

Below is a list of the main tasks covered by Modest, together with the basic functions

SIMULATION

- numerical solutions at given data points
- numerical solutions at given data points, with varying model parameter values

PARAMETER ESTIMATION

- fitting the model to data
- standard confidence interval statistics

OPTIMAL DESIGN OF EXPERIMENTS

- optimize experimental conditions for accurate parameter estimates
- local and global criteria

SENSITIVITY ANALYSIS

- global sensitivity within user given bounds
- graphical display, 1D and 2D (contour) plots for all objective functions

OPTIMIZATION

- general, user given objective function
- multicriteria optimization

3 Basic concepts

Model types

A mechanistic mathematical model is based on the physico-chemical explanations of the observed phenomenon. A mechanistic model can be formally written in the form:

$$s = f(x, \theta, c) \tag{1}$$

$$y_p = g(s), \tag{2}$$

where the quantities are defined as follows:

s	the state of the system
y_p	the observed (response) variables,as predicted by the model
x	the experimental (design) variables
θ	the estimated parameters
c	the constants

The function f describes the model itself, while the function g gives the information of the available observations. It is often impossible to directly measure all the states s of the system. The relation between the observations y and the state s can be rather complex and is described by a separate observation function g.

The physical and chemical phenomena can be either static or dynamic. Correspondingly, the model may consist of *algebraic* or *ordinary differential* equations. Steady state balance models may also lead to system of *nonlinear algebraic* equations of the implicit form $f(s, x, \theta, c) = 0$. In case the model is described by differential equations or in implicit form it has to be solved in the form $s = f(x, \theta, c)$ using an appropriate numerical solver. Note also that many steady state phenomena, like transport of materia in pipes, are described by differential equations.

Example 1 We will clarify the notations using a very basic example. A consecutive chemical reaction A $\rightarrow$ B $\rightarrow$ C is considered. If the concentrations of A and B are denoted by c_A and c_B, respectively, the generation rates of A and B in a batch reactor may be given by the differential equations

$$\frac{dc_A}{dt} = -k_1 c_A \tag{3}$$

$$\frac{dc_B}{dt} = k_1 c_A - k_2 c_B \tag{4}$$

In the beginning of the experiment ($t = 0$) the concentrations of A and B have certain given values $c_A(0)$, $c_B(0)$. In the notation of $(1) - (2)$, the state of the system is now given by the vector

$$s = \begin{pmatrix} c_A \\ c_B \end{pmatrix},$$

i.e., $s(1) = c_A$ and $s(2) = c_B$. Let us assume that only the concentration of B (c_B) can be analyzed. The observation function then is $g(s) = s(2)$, i.e.,

$$y_p(1) = s(2)$$

The unknown reaction rates should be adjusted in such a way that the calculated concentrations y_p become as close as possible to the experimentally observed concentrations y. The reaction rates generally depend on temperature, so the parameters should be rewritten in temperature dependent terms. The Arhenius law

$$k = Ae^{-\frac{E}{RT}} \tag{5}$$

describing the temperature dependence of the reaction rate constant k in terms of the activation energy E and amplitude factor A. So the parameters to be estimated are, tentatively, given by

$$\theta = \begin{pmatrix} A_1 \\ E_1 \\ A_2 \\ E_2 \end{pmatrix}.$$

To find out the dependence, we consider the reactions at different fixed temperatures. Let us assume that, except for the initial concentrations, all the other factors that might influence the reaction, are kept constant. If the samples of B are taken at fixed time points t_j, the role of the experimental variables is then played by the temperature and the initial values. So, in this case,

$$x = \begin{pmatrix} T \\ c_A(0) \\ c_B(0) \end{pmatrix}$$

Note that we might also consider the sampling instants t_j as variable experimental conditions.

Algebraic models If the model can be written in the *explicit* form $s = f(x, \theta, x)$ it is called algebraic. The values s are obtained with direct substitutions in the formulae of the model, no solvers are needed.

As an example we may write the analytical solution for the differential system (3) in Example 1. With initial values $s_1(0) = 1, s_2(0) = 0$ one easily verifies that

$$s_1(t) = e^{-k_1 t} \tag{6}$$
$$s_2(t) = \frac{k_1}{k_2 - k_1}(e^{-k_1 t} - e^{-k_2 t}) \tag{7}$$

Implicit algebraic models Models written in form of a nonlinear algebraic equations system are called implicit algebraic. A nonlinear equation solver is needed to get numerical values of s from the system $f(s, x, \theta, c) = 0$. An example, albeit a rather trivial one, is obtained from the previous example by writing it in the form

$$s_1(t) - e^{-k_1 t} = 0 \tag{8}$$
$$s_2(t) - \frac{k_1}{k_2 - k_1}(e^{-k_1 t} - e^{-k_2 t}) = 0 \tag{9}$$

A more challenging model is solved in the demonstrations.

Ode models An example of ordinary differential equations, generally written in the form

$$\frac{ds}{dt} = f(x, s, \theta, c), \quad s(0) = s_0, \tag{10}$$

we already met in Example 1. The ode systems arising from chemical kinetics are often very *stiff* and require a high quality numerical solver.

PDE models Differential equations containing derivatives with respect to several coordinates - time, spatial coordinates – are called PDE, Partial Differential Equation, systems. As a typical example we might consider a substance flowing in a pipe of length L in the direction z with velocity v,

$$c_t = -vc_z + Ec_{zz} + R(c), \quad c|_{z=0} = c_0, \quad c_z|_{z=L} = 0.$$

Here E denotes the dispersion coefficient and R the reaction rate. Both initial and boundary conditions are needed for the solution. As the routine method we use the numerical method of lines (NUMOL, see [1]).

The structure of data and model components

Several observations y can be made of the different response variables during a single experiment. Moreover, several experiments are usually needed for the model identification. The data structure so created is thus typically three dimensional, the dimension indexes giving the number of response *components*,

the number of *observations* in each data set, and the number of the *data set*. The number of observations may change from data set to another. Even the number of observed components might change from one observation to another.

The concept of data set is, especially, relevant with models described by differential equations. A data set then corresponds to one integration of the system (10). We also use the word *batch* as a synonym to data set.

Let $nsets$ denote the number of data sets, $nobs(k)$ the number of observations in data set k, and $nydata(j,k)$ the number of observed response components in data set k and at observation j. The whole data set so created can be given by the matrix

$$y(i_y, j, k), \qquad i_y = 1, ..., nydata(j,k) \tag{11}$$
$$j = 1, ..., nobs(k) \tag{12}$$
$$k = 1, ..., nsets \tag{13}$$

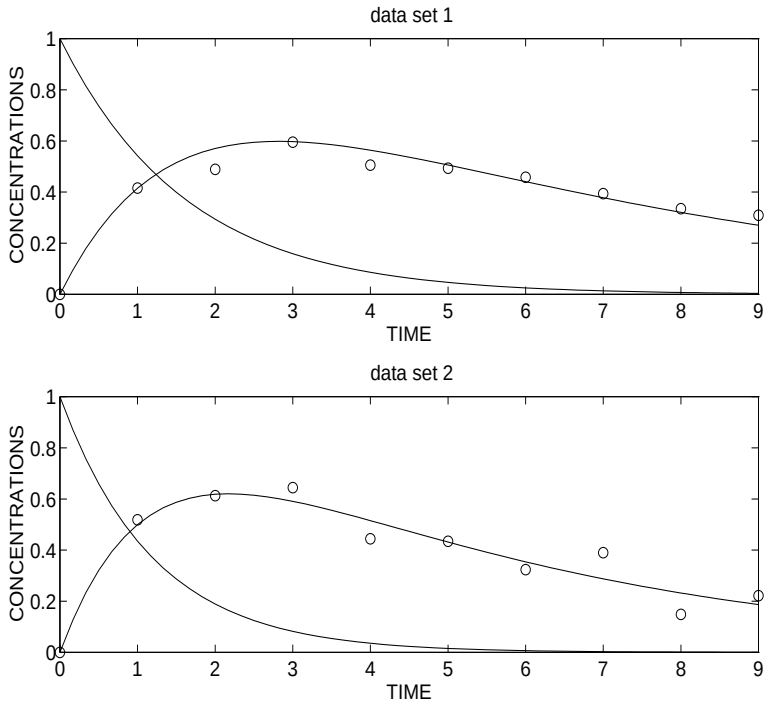
In a similar way the components of the model system may be given by

$$s(i, j, k), \qquad i = 1, ..., nstates \tag{14}$$
$$j = 1, ..., nobs(k) \tag{15}$$
$$k = 1, ..., nsets, \tag{16}$$

where $nstates$ denotes the number of components in the model system.

The figure below exhibits the state variables, observation variables and observations in two data sets for the model presented in Example 1.

Figure 1



The values of data are given in *data files* in the usual observation matrix form: all the values belonging to a given observation j are given at one row of the file. Different data sets may either be given in different files or pooled all consequtively in one file. Below is a typical example, the same data as presented in Figure 1.

```
C  FILE:  EXAMPLE1.DAT.  Pooled data for 2 data sets. Initial values
                         at time = 0 given in the IO interface
!  time             B
   1.0000000e+00    4.1553047e-01
   2.0000000e+00    4.8858236e-01
   3.0000000e+00    5.9496475e-01
   4.0000000e+00    5.0557022e-01
   5.0000000e+00    4.9319709e-01
   6.0000000e+00    4.5776855e-01
   7.0000000e+00    3.9403871e-01
   8.0000000e+00    3.3477578e-01
   9.0000000e+00    3.0952097e-01

%  time             B
   1.0000000e+00    5.1841137e-01
   2.0000000e+00    6.1318360e-01
```

```
3.0000000e+00    6.4415809e-01
4.0000000e+00    4.4400307e-01
5.0000000e+00    4.3479304e-01
6.0000000e+00    3.2339277e-01
7.0000000e+00    3.8965389e-01
8.0000000e+00    1.4877750e-01
9.0000000e+00    2.2161736e-01
```

Variable types

Named variables In a model different variables appear in different ways, in addition to the roles (θ, x) given in (1). A variable may be just a constant or dummy, auxiliary name in the code. Often, however, we would like to 'play' with an interesting variable: simulate the model with different values of that variable, fit the value of it to given data, optimize some objective function with that variable as an argument, etc. To be able to do all this at the user interface level, without the need of re–editing and re–compiling the model, such a variable must be given a *name* and *type*. The name is up to the user, the type is determined by how the variable behaves in the model or experiments.

Global Typically, we try to estimate certain constants from a number of experiments. The values of these constants remain *fixed in all the experiments*. Such variables are called *global* (note, however, that a variable may be global but not estimated!)

Local A variable which may *change from one data set to another* but remains *constant within each data* set is called *local*.

Control A variable which may *change within data sets, in a prescribed way* is called a *control* variable.

Odevar The integration argument, only for ode models.

Initval The initial values for the system (10) for each data set. Also used as the initial guess for the roots of the nonlinear equations of an implicit algebraic model.

We note that the type Odevar actually could be interpreted as a Control variable, as well as Initval as a Local variable. We have kept the names Odevar and Initval for convenience.

4 Goodness of model

Least squares estimation

The closeness of the data and the values predicted by a model can be measured, in principle, with several criteria. The most common objective function according to which the parameters are estimated is, however, the sum of residual squares.

If the observations y_{ijk} are available at the experimental points x_{jk}, the sum of squares of the residuals between the model and data is given by

$$l(\theta) = ||y - y_p||_w^2 = \sum_{k=1}^{nsets} \sum_{j=1}^{nobs(k)} \sum_{i=1}^{nydata(j,k)} (y_{ijk} - y_{p_{ijk}})^2 w_{ijk} \qquad (17)$$

where the values y_p denote the predictions given by the model with the parameter values θ, and w gives the *weight matrix* for the observations. We have also introduced here the (weighted) norm notation $||y - y_p||_w^2$ as an abbreviation for the sum of squares.

To get the values of y and y_p as close as possible, in an average sense, the above sum is minimized with respect to θ. The different terms in the sum can be weighted by the weight factors w. A basic principle is that every data point should be divided by the estimated standard deviation of it. If all the response components are of comparable magnitude, the simple choice $w = 1$ is often used.

The minimization of l can be performed with a number of different numerical optimization routines.

The most common measure for the goodness of fit is the *coefficient of determination*, the R2–value. The idea is to compare the residuals $y - y_p$ given by the model to the residuals of the simpliest model one may think of, the average value $\bar{y}$ of all the data points. The R2 value is given by the expression

$$R2 = 100(1 - \frac{||y - y_p||^2}{||y - \bar{y}||^2}) \qquad (18)$$

So $R2 \leq 100$ – the closer the value is the number 100, the more perfect the fit. As a rule of thumb one might have that a mechanistic model, with reasonable amount of noise in the data, should have R2 values well above 90 (note that it is quite possible, in an initial stage of modelling, that the R2 value might be negative. Then the fit given by the model is worse than the average value of the data points)

Identifiability

The aim of parameter estimation is to find 'true' values for the model parameters. If we only have one parameter to be estimated, the meaning of 'accurate value' of it is rather clear: the uncertainty or, statistically speaking, variance of it, should be minimized. With many parameters to be determined simultaneously there is, in fact, no unique 'most accurate' θ. Several criteria have been proposed.

The question of identifiability can be addressed using the *maximum likelihood function*. Let us consider, in stead of the sum of squares function l, the function

$$p(\theta) = e^{-\frac{1}{2\sigma^2} l(\theta)}.$$

Here σ is the standard deviation of the pure experimental error. Clearly, the maximum of p is obtained at the minimum of l, i.e. at the solution point of the least squares problem. On the other hand, as the values of l get larger – the fit between the data and model worse – the values of p approach zero. The maximum value of p is the so called maximum likelihood solution of the modeling problem. The name originates from the interpretation: it is the most probable value of parameter θ, based on the experimental observations.

One might consider the function p as a 'probability hill' for the values of θ. The peak of p is the most probable value for θ, the contour lines $p = const$ determine regions of θ, the so called *confidence regions*, giving equally good models, equally good or bad fits between the model and data.

If the values of the function p decrease rapidly, in every direction from the peak point, the parameters are well identified: only the maximum point and points in the immediate vicinity of it give good fits. On the other hand, if the values of p decrease slowly around the maximum, in some direction at least, then quite different values of the parameters θ give an equally good fit to the experimental data, and the model is badly identified. The aim of the experimental design is to optimize the identifiability.

By plotting one or two dimensional contour lines of function p we can study the identifiability of the problem. When several parameters are estimated simultaneously the problem might be well identified with respect to some parameters, and quite badly identified with respect to other parameters. Even more usual is the situation that there are strong correlations between the parameters: the values of certain parameters can, in suitable relation with each other, be considerably altered without essentially affecting the fit between data and model.

The identifiability of a model is often strongly influenced by the parametrization of it. Before embarking in the optimization of the experiments one should thus carefully consider the parametrization: parameter transformations (e.g.,

scaling and centering), combinations of parameters, writing the model in nondimensional form to get nondimensional groups of parameters, etc.

Example 2.

The Arhenius law in the form (5) is an example of bad, correlated parametrization: by suitably increasing both A and E the value of k remains virtually unchanged. It is advisable to write the law in the form

$$k = Ae^{\frac{-E}{RT}} = k_{mean}e^{-zE}$$

where $k_{mean} = Ae^{-E/RT_{mean}}$, $z = 1/R(1/T - 1/T_{mean})$, and T_{mean} is some 'mean' temperature value, between the minimum and maximum used in the experiments. Instead of the the original A, E we now estimate k_{mean}, E (or E/R, in order to avoid dimensional errors).

The two pictures below give the contour lines of Arhenius law with the two parametrizations for the same data, $k = 0.55, 0.7, 0.8267$ at the temperatures $T = 283K, 300K, 313K$.

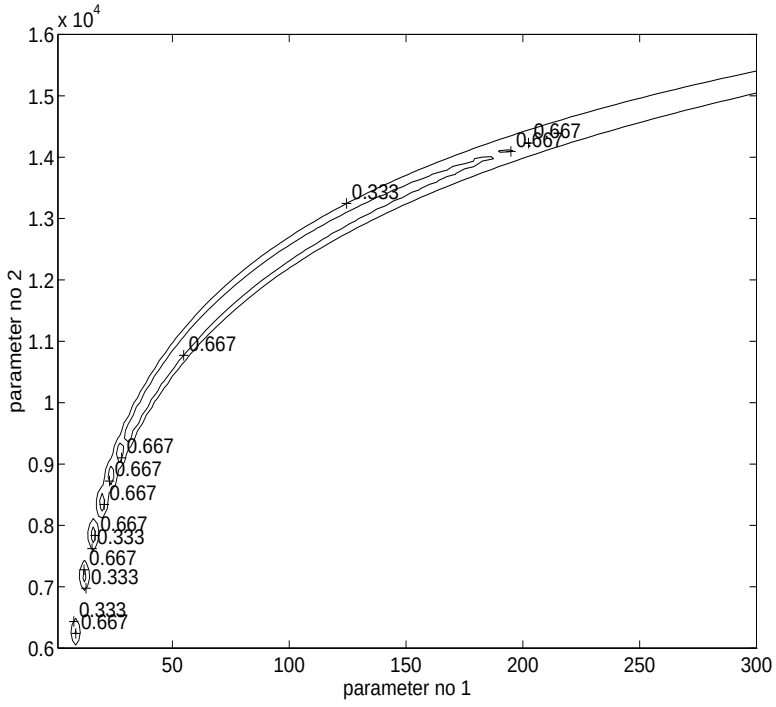


Figure 2.1. Contour for Arhenius parameters A,E

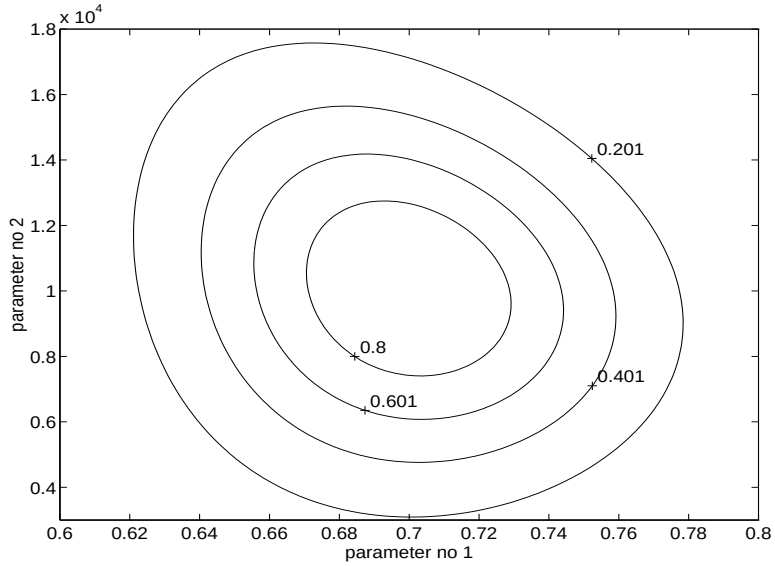


Figure 2.2. Contour for Arhenius parameters k_{mean},E

The pictures give contours for scaled maximum likelihood functions, obtained by dividing the values of $exp(-1/2\sigma^2 l(\theta)$ by the maximal value in the grid. The error level in the observations k_j was taken to be $\sigma = 0.075$.

5 Syntax for named variable input

The user defines the named variables in a character table. This table is, in fact, a part of the input definitions that can either be given in a Fortran namelist or by a Windows user interface. More details of IO is given in section 7.

The format for the named variables is the following

```
name            type          numerical value(s)
```

The different variable types require slightly different syntax rules so we give them separately, by examples.

Global

1 *Scalar name* The variable 'k1mean' gets the type global and value 0.02 by

```
kmean1    global        0.02
```

The numerical value may be given in the formats 2e-2, 2d-2, 0.02, .02 .

2 *Vector name* The names for a number of variables may be given in a vector. The numerical values are separated by empty spaces, e.g.,

```
kmean(1:5)    global       1e-2 2e-2 3e-2 4e-2 5e-2
```

The *only allowed separator* for numerical values is *the empty space* – e.g., the comma sign is reserved for other purposes. The numerical values may be given at one or several rows. If all the components have the same numerical value – as an initial guess for optimization, for example – this can be indicated by a semicolon as

```
kmean(1:5)    global       1e-2;
```

The situation where all components, starting from a certain index, assume the value of the last given value, can be specified as

```
kmean(1:5)    global       1e-2 2e-2 3e-2;
```

That is,$kmean(3) = kmean(4) = kmean(5) = 3e - 2$.

3, Matrix name A global variable can also be a matrix. A stoichiometric matrix for chemical kinetics might be given in the form

```
N1(1:3,1:4) global    1  0 -1  1
                      0  2 -1  2
                     -1 -2  2  0
```

Local A local variable assumes different values in different data sets. The values are separated by spaces and written on one or several rows:

```
Temp  local     30 40 50 60 70
```

A local variable may also be a vector – but not a matrix. Values for vectorized local names must be given as if the values would be given for respective scalar names, i.e. first all numerical values for the first component, then for the second, etc:

```
Temp(1:2)  local   30 40 50 60 70  % Temp(1) in 5 data sets
                   10 10 20 20 20  % Temp(2) in 5 data sets
```

Truncations using the semicolon work as with global variables. The above example could also have been written as

```
Temp(1:2)  local     30 40 50 60 70
                     10 10 20;
```

Control A control may get different numerical values at each observation point. Usually these values are given in the data file, and it might be tedious to edit them in the input table. For these reasons only the name and type are given at the user interface level, the numerical values must always be read in from files.

As an example, we could define an oscillating input variable 'phase' and time dependent 'feed' for an ode system by

```
phase    control
feed     control
```

Initval Initial values are basically given in the same way as the values for local variables. For instance, the initial values $s0$ for 4 different rund of an ode system with 3 components could be given by

```
s0A  initval  0.1  0.11 0.23 43
s0B  initval  10.0 9.21 6.0  12.0
s0C  intival  0    0    0    0
```

or, alternatively,

```
s0(1:3)  initval  0.1  0.11 0.23 43
                 10.0  9.21 6.0  12.0
                   0;
```

In many cases, however, all or a part of the inital values are directly read in from data files. This can be indicated by writing, in the above format, the word 'file' instead of the numerical value, rules for the semicolon etc remain the same. Suppose the second component above would get the values form data file (whose name is given elsewhere). We would then write

```
s0(1:3)  initval  0.1  0.11 0.23 43
                 file;
                  0;
```

as an abbreviation for

```
s01  initval  0.1  0.11 0.23 43
s02  initval  file file file file
s03  initval  0    0    0    0
```

The layout of the data files is further spesified by the tables 'ncolxy', 'nydata', 'indx' and 'indy', see the namelist FILEPAR in the appendix. Note that the number of 'file' flags for each data set as given above must match the number given by 'nydata'.

Odevar The type Odevar gives the name for the integration argument for an ode system. Often the values, including the initial value for integration, are given in a data file. In this case it is not necessary to define the Odevar at all (altough the column index of the values in the data file has to be given, elsewhere). The default name 'time' is used in case the user should need it explicitly in the model code. Note that the Odevar variable refers to the *observation /output points* explicitly given by the user, either at the user interphase or in data files. The intermediate integration argument values, as employed by the ode solver used, always have the dummy name t in the model code.

The values for Odevar may be given at the user interface in various ways.

1. Only the initial point given. The situation typically arises when the data file does not contain the row for the initial values. As an alternative for editing the data file we may give the initial point (and values, as above) in the model variable table. So

```
length      odevar      0.1;
```

will give the integration argument the name 'length' and the integration starts at the value $length = 0.1$ in all data sets – the semicolon works as for the previous variable types. To make a distinction to the next two cases below (where two or more time points are given) an additional syntax rule is employed: different initial points for different data sets must be separated with ';'. For example,

```
length      odevar      0.1; 0.01; 0.2;
```

would give the above starting point values for data sets 1,2, and 3 – and the values 0.2 for all the following sets.

The default value for the starting point is 0. If this value is valid, the definition of Odevar may, again, be omitted.

All the other integration (observation) points must, in this case, be given in the data files. The number of the points *in the data file* is given by the index $nobs(k), k = 1, ..., nsets$.

2. Only the initial point and final points given. The situation typically arises in

simulations: we only would like to specify the integration interval (or perhaps several of them), i.e., give the initial and final points. The definition

```
length      odevar     0.0 11.0;
```

would give the above interval for all integrations. The semicolon rule works as in the previous case,

```
length      odevar     0.0 11.0; 1 20; 1 50;
```

yields the respective intervals for runs 1,2 and 3 – and the interval [1 50] for all the rest. The number of points where the solution is computed for output (graphical plots) is again given by the index vector *nobs*. For each run k, the solution is given in $nobs(k)$ uniform points of the interval.

3. All observation points given. The solution of a simulation can also be calculated at any specified points, in the form

```
length   odevar  0.0 1.414214 2.718282 3.141592653;
```

The rules for ';' are as above. If, in addition, a smooth solution curve would be desirable, the *dump file* can be employed. For more details see the next section.

6 Tasks in model building

Mathematical and numerical modelling typically consists of several steps: formulation of one or various models to describe the phenomen in question, writing a numerical code for the model and running some preliminary simulations, making experiments to get data for validation of the model, and finally, when a reasonable model has been created, using it for simulations or optimization of a target function.

Suppose we have chosen a model of the form (1). The model building now, in a more narrow sense, means that we should find numerical values for the unknown parameters θ of the model, which make the model agree with empirical data. Moreover, we should be able to estimate how accurate or reliable the parameter values are – and how the inaccuracies effect the reliability of the model predictions in given situations.

Typical stages of the parameter estimation procedure are:

1. Choice of the experimental points x.
2. The experimental work, i.e. the measurement of the y-values.
3 Estimation of the parameters θ, analysis of the accuracy of the result.
4. If the accuracy is not sufficient, additional experiments are made, and the procedure is continued at stage 1.

Depending on the stage, either the parameters θ or the independent variables x are the quantities to be optimized.

In *experimental design* the optimal experimental points are searched, so the variables x are optimized with fixed θ. In *parameter estimation* the parameters θ are optimized, and the variables x have fixed values.

Optimality in parameter estimation means a close fit between the experimental data and the values y calculated by the model. Optimality in experimental design has a different meaning: the points x should be so chosen that the parameters θ would be identified with best possible accuracy – after the new experiments have been performed, in the conditions suggested by the optimized design.

At the beginning of the experimental work we might have no guess for the values of θ. The initial experiments should then be selected with some 'good judgement' or by imitating the 2^N –type plans of empirical statistical models. The subsequent experiments, with estimates for θ already available, can then be optimized using some of the criteria for experimental design of mechanistic

models. This procedure, called *sequential design*, is illustrated below with an example.

The standard way to measure the accuracy of the parameters is to compute the confidence limits for each parameter. However, the confidence limits, as well as most other statistical criteria, are based on the theory valid for model *linear* with respect to the unknown parameters. In case of nonlinear models such criteria may be quite misleading. This is especially true if the parameters are strongly correlated – which, unfortunately, often is the case. It is thus advisable to perform more extensive *sensitivity analysis*: compute the values of the objective function in some environment of the best parameter value found, and produce one or two dimensional graphical pictures of the 'landscape'.

Once a satisfactory model has been found, it remains to use the model, in addition to interesting simulations, for *optimization*. Generally, one would like to achieve an optimal result with minimal negative factors – harmfull side products, costs, etc. The various goals often are contradictory and one is lead to a multitarget optimization task. Typically, the objective function has to be specified by the user. So, while the objective functions for parameter estimation and optimal design are built–in routines in the Modest software, the optimization task requires a user written subroutine. We shall give examples of a method, the *desirability function technique*, which is especially easy to implement and in many cases sufficient for multicriteria optimization.

Let us first consider, in terms of the familiar model of Example 1, in somewhat more detail how the five different tasks – simulation, estimation, optimal design, sensitivity, optimization – can be performed with the Modest software. In the examples we shall exhibit some of the Fortran namelist input formats, since the namelist interphase is available in all computer environments. A more exhaustive list of all the namelists and namelist variables is given in section 7, as well as the description of the Windows interphase.

Simulation

We start with a basic simulation: compute the solution to the system (3) with two different temperatures and integration intervals. The solution is required at 60 evenly spaced points in the respective time intervals. Below are the relevant namelists where we define the task to be performed, the type of the model, the ode solver selected, the model variables and their numerical values, and the number of time points where the solution is to be obtained.

```
&files
nsets   = 2
resultfile = 'boxodsi1.sim'
```

```
 dumpfile   = 'nodump'
/

&problem
 task  = 'simulation'
 model = 'ode'
 odesolver = 'odessa'
/

&modelpar
modelvar = 'k1mean   global   0.7
            E1       global   0.01
            k2mean   global   0.2
            E2       global   0.007
            Tmean    global   300.
            s01      initval  1.2 1.0
            s02      initval  0.5 0.0
            Temp     local    480 200;
            time     odevar   0  7;  0  50;'

 nstates = 2
 nsaux = 3
/

&filepar
 nobs(1) = 60
 nydata(1) = 1
/
```

As a default Modest computes (for an ode system), in addition to the specified observation points, the solution also to a 'dump' file at a user given number *ndumpp* points for each integration. This is most useful for cases where the observation points are scarce and smooth solution curves are wanted. Here we may set the observation points sufficiently dense with 'nobs' so the creation of the dump file is unnecessary and is supressed by setting "dumpfile = 'nodump' ".

Note that the number of observations is above explicitly given only for the first batch by "nobs(1)". As a default, the value of the first batch is given for all the rest, if nothing else is specified. The same holds for the matrix "nydata", which gives the number of observed responses in each observation and data set (batch). It sufficies to give it only for the first observation in the first batch to indicate that it remains constant everywhere.

Note also the variables "nstates" and "nsaux". The former gives the number of components of the ode system, the latter the number of *auxiliary states*. In many situations we want to get the time dependent values of several interesting variables that are not components of the ode system – reaction rates, hold–ups, interphasial areas in multiphase systems, etc. These can be defined as the auxiliary states.

Below are the user written parts of the Fortran codes for this example. For more details of templates, compiling, running etc see Section 7.

```
C     The file boxom.for: the model file for Example1.

*     LOCAL VARIABLES

      real*8    R                ! the gas constant
      real*8    k1,k2            ! thet reaction rate coefficients
      real*8    z                ! the Arhenius transformation
      real*8    A,B              ! concentrations

      A = s(1)
      B = s(2)

      e1 = e1*1.0d+6
      e2 = e2*1.0d+6

      R  =  8.314
      z  = 1.0d0/R * ( 1.0d0/Temp - 1.0d0/Tmean)
      k1 = km1 * dexp(-e1*z)
      k2 = km2 * dexp(-e2*z)

      ds(1) = - k1 * A
      ds(2) =   k1 * A - k2 * B

      s(ns+1) = ds(1)        ! aux states, for demonstration
      s(ns+2) = ds(2)
      s(ns+3) = t


*     File: boxoo.for. Observation function for Example1

*     LOCAL VARIABLES
*     none

         yest(1) = s(2)
```

In the above manner we may run a given model in various experimental conditions. In addition, we often want to simulate the model with varying model parameter values (or, indeed, with any named variable values). In Modest this can be achieved by defining a *target* table for the parameters to be varied. The lower and upper bounds for each parameter are given in *target*, too. The number of evaluation steps between the bounds are given –for each parameter, in the order of appearance in *target* – in the vector *nevastep*.

```
simulation example.

- 2 batch runs with different initial values and temperatures.
- Multiple simulations with parameter values given by 'target'
- Integration interval and initial values given in the namelists.
- No data files, no dump file


&files
 nsets      = 2
 resultfile = 'boxodsi5.sim'
 dumpfile   = 'nodump'
/

&problem
 task  = 'sim'
 model = 'ode'
 odesolver = 'odessa'
/

&modelpar
modelvar = 'k1mean   global   0.7
            E1       global   0.01
            k2mean   global   0.2
            E2       global   0.007
            Tmean    global   300.
            case     global   1
            s01      initval  1.0  1.0;
            s02      initval  0.0  0.1;
            Temp     local    320  330
            time     odevar   0 7;'
  target  = 'E1      0.005  0.1
             E2      0.001  0.07'
  nevastep = 2 2
```

```
  nstates = 2
  nsaux = 3
/


&filepar
 nobs(1) = 60
 nydata(1) = 1
/
```

Above, we have chosen the minimum number of steps, 2. The solution is then calculated at the lower and upper limits – in addition to the reference values given in *modelvar*.

Arbitrary evalutation points for the *target* parameters may be given in an $evalfile$, specified in the $files$ namelist. Each row of such a file gives one simulation point of the model.

Some of the simulation curves produced by the example – the effect of changing the activation energy of the second reaction in the first data set – are shown in Figure 3.

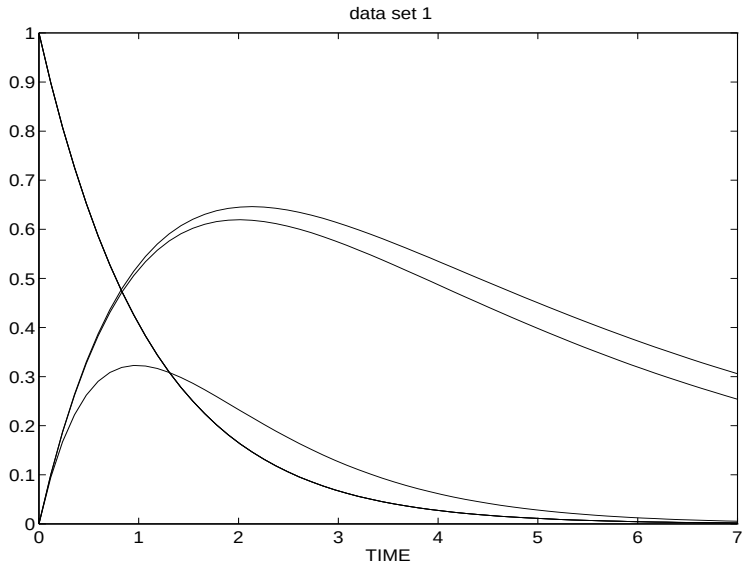


Figure 3

Parameter estimation

Let us now fit the parameters $k1_{mean}, E1, k2_{mean}, E2$ of our Example 1 to a data consisting of observation of both concentration components A and B. Suppose we have the initial guesses $1, 0.01, 1, 0.01$ for the respective parameters

– note that we here use scaled values for the activation energies. In addition to the model variable values we have to give the names of the parameters to be estimated. The parameters and simple bounds for them are specified via the tabel *target*, exactly as in the 'multiple' simulation case above. The relevant namelist definitions then read as

```
estimation of kinetic parameters

2 data sets, combined in the file 'boxodes1.dat'.

s0:  given in the namelist 'modelpar'
     Note: only given for the 1.batch, copied for the 2.batch as default
t0:  default value 0 used, could be given in 'modelvar' table


&files
 nsets       = 2
 datafile(1) = 'boxodes1.dat',
 reportfile  = 'boxodes1.sta'
 resultfile  = 'boxodes1.est'
 optfile     = 'boxodes1.arg'
 ndumpp      = 60
/

&problem
 task  = 'est'
 model = 'ode'
 odesolver = 'odessa'
 optimizer = 'simflex'
/

&modelpar
 nstates = 2
 nsaux   = 3

 modelvar = 'k1mean   global   1.0
             E1       global   0.01
             k2mean   global   1.0
             E2       global   0.01
             Tmean    global   300.
             case     global   2.
             Temp     local    283.   313.;
             time     odevar   .1; .01;
```

```
            s0(1:2)  initval  1.0;
                              0.0; '

  target  = 'k1mean       0.01  100.
            E1            1.e-4  1.0
            k2mean        0.01  100.
            E2            1.e-4  1.0'

 dstep = 1e-6
/

&filepar
 combined = 1

 nobs(1)   = 9
 ncolxy(1) = 3
 nydata(1) = 2
 indx(1,1) = 1
 indy(1,1) = 2
 indy(2,1) = 3
/

&print
 echo = 1
 echodata = 1
 optmonit = 1
 stats = 1
 jacout = 1
/

&design
/

&simflex
 abstols = 1.00E-08
 reltols = 1.00E-08
 sizes = .1
 itmaxs = 100
/
```

We choose the task performed by setting $task = est$. Then we also have to choose an optimizer which, starting from the initial guesses of the *target* variables as given in the *modelvar* table, tries to find the minimum of the least

squares sum (17). In all the examples here we use a version of the *simplex* optimizer. The idea of this optimizer is straightforward: it first computes the values of the objective function (17) in a 'simplex' consisting of $nest+1$ points, where *nest* is the number of estimated parameters. It then starts, comparing the values in the $nest+1$ points, to 'roll' towards the lowest level of the valley in the landscape of the objective function.

A few parameters have to be set to tune the optimizer. *sizes* sets the initial size of the simplex, in relation to the bounds given in *target*. *itmaxs* gives the maximum number of search iterations. Roughly, in each iteration the objective function (17) is called $nest + 1$ times. *abstols* and *reltols* give the absolute and relative stopping criteria for the iteration – if the absolute or relative differences in consequtive objective function values goes below the tolerances, the iteration is stopped. By setting the tolerances small in enough, as here, we ensure that the *itmaxs* number of iterations is performed.

The standard statistical analysis – R2 value of the fit, confidence intervals for parameters, correlation matrix for parameters – of the fit is written in the file given by *reportfile*, supposing that additionally the namelist parameters *stats* and *jacout* assume the value 1. With $jacout = 0$ no Jacobian is calculated and only the R2 value and parameter values are written. The Jacobian is computed numerically, the step size is given with *dstep*.

Below is a part of the output in the *reportfile* for our example.

```
Total SS (corrected for means)  .1487E+01
Residual SS                     .1007E+00
Std. Error of estimate          .5611E-01

Explained (%):  93.23

Estimated   Estimated  Est. Relative Parameter/
Parameters  Std Error  Std Error (%) Std. Error

 .755E+00    .432E-01        5.7         17.5
 .627E-02    .270E-02       43.1          2.3
 .205E+00    .969E-02        4.7         21.1
 .433E-02    .230E-02       53.1          1.9

The correlation matrix of the parameters:

1.000
 .303  1.000
-.079  -.049  1.000
-.048  -.067   .143  1.000
```

```
The eigenvalues of the correlation matrix:

1.369  1.077   .861   .693
```

For further use, e.g., for continued estimation, the optimized parameter values are also written in a separate file if a name for the file is given in $optfile$.

The namelist $filepar$ contains all the details on how the data is organized in the files. $nsets$ gives the number of data sets (batches) as described in (11). The sets of data may either be in one or all in separate files. The former case is chosen if $combined = 1$, the latter with $combined = 0$. $nobs$ gives the number of observations in each data set (if no initial values are given in a file, they are *not* counted in $nobs$). $ncolxy$ gives the number of *all* columns in a file, including possible irrelevant columns – number of observations, unused data, etc. The indexes $indx$ of the control variables and $indy$ of the response variables give the respective column indexes, the matrix $nydata$ the number of response components at each observation. As before, the values at the first observation/data set are copied for all the rest if not otherwise specified.

Note that we may give a weight value for each observation by setting $usew = 1$. Again, the weight matrices may either be given in one file or all separately. With $combined = 1$ they must be given in the data file, after the data points. With $combined = 0$ they are given in separate file(s). If $usew = 0$ no weights need to be given. This is the default case and means that all observations are 'weighted' by 1.

The namelist *print* also contains the echoing flags *echo*, *echodata* and *optmonit*. With $echo = 1$ the number of observations and the indexes $indx$ and $indy$ are echoed when the data is read from the files. With $echo = 0$ the printing is supressed. Similarly, the parameter *echodata* determines whether the data values are echoed or not. If *optmonit* recieves a positive (integer) value, the maximum and minimum values of the objective function as well as the argument values of the minimum are echoed at each *optmonit*:th iteration.

The initial values may have several roles in estimation. Above they were supposed to be (exactly) known, and the estimation was based on the residual squared sums (17) including all the observations *after* the initial points. The initial values might also be unknown. In that case the values of them could also be estimated, just by adding the names and bounds in the *target* table:

```
target  = 's0(1)   1     0.8 1.2
          s0(2)   1     0.0 0.2
          s0(1)   2     0.8 1.2
          s0(2)   2     0.0 0.2'
```

Note that we must give the index of the data set for each target variable that changes with the set. A third possibility, more often met in practise, is that the values used as initial values are observed, noisy data just as all the rest of the data. We might then simply interprete the initial values as known and estimate the parameters as before. However, if the error in the initial values is substantial, this might lead to distorted results. We then actually should estimate the values, but also use the available data values in the objective function (17). This can be achieved by setting $obss0 = 1$ – the default setting $obss0 = 0$ drops the initial point from the sum.

Below is such an example. The model and data are essentially the same as in the earlier case, only the data values for the initial points are added. Moreover, they are weighted differently as the rest of the data.

```
FILE:     BOXFODE2.NML
Kinetic parameters estimation example.

Estimate: the kinetic parameters and unknown initial values
Data:     measured values for component 2, including initial states
t0,s0:    given in the data sets.


&files
 nsets       = 2
 datafile(1) = 'boxodes2.d1'
 datafile(2) = 'boxodes2.d2'
 weightfile(1) = 'boxodes2.w'
 weightfile(2) = 'boxodes2.w'
 reportfile  = 'boxodes2.sta'
 resultfile  = 'boxodes2.est'
 optfile     = 'boxodes2.arg'
/

&problem
 task  = 'est'
 model = 'ode'
 odesolver = 'odessa'
 optimizer = 'simflex'
/


&modelpar
nstates = 2
nsaux   = 3
```

```
modelvar = 'k1mean   global   1.0
            E1       global   0.01
            k2mean   global   1.0
            E2       global   0.01
            Tmean    global   300.
            case     global   2.
            Temp     local    283.   313.;
            s0(1:2)  initval  file;'

 target  = 'k1mean       0.01   100.
            E1           1.e-4  1.0
            k2mean       0.01   100.
            E2           1.e-4  1.0
            s0(2)  1     0      0.2
            s0(2)  2     0      0.2'
 /

&filepar

 combined  = 0
 nobs(1)   = 10
 obss0     = 1
 usew      = 1

 ncolxy(1)   = 3
 nydata(1) = 2
 indx(1,1)   = 1
 indy(1,1)   = 2
 indy(2,1)   = 3
/

 &print
  echo     = 1
  echodata = 1
  optmonit = 1
  jacout   = 1
  stats    = 1
 /

 &design
 /
```

```
&simflex
 abstols = 1.0e-6
 reltols = 1.0e-6
 sizes   = .2
 itmaxs = 200
/
```

To avoid too many parameters in the estimation, it might be advisable to estimate first just the kinetic parameters with reasonable fixed initial values, and let everything 'float' only at a final stage of estimation.

Sensitivity analysis

Standard statistical information about the accuracy of the parameter values are given in the reportfile. For nonlinear models such information should, however, be regarded as preliminary in most cases. In the example above the relative standard error for both activation energies was estimated to be roughly 50%. How good or bad would the fit be if the parameters would change that amount? The statistical numbers given in the reportfile are based on *local* information, numerically computed derivatives of the model at the optimized argument value θ_0. To be able to see how the fit changes in some environment of the point θ_0 one must perform *global* calculations in the environment.

A more global picture of the identifiablity of the parameters is given by plotting 1 and 2 dimensional profiles of the least squares objective function. For the activation energies this is done in Figure 4. We notice that the activation energy of the second reaction, especially, is not yet well determined. The contours plotted present the scaled maximum likelihood surface. The size of the pure experimental error is taken as estimated by the fit and given in the $reportfile$. For more details of the plotting macros see Section 7.

The relevant input definitions required to produce the data for the plots are given below

```
FILE:     BOXODSE1.NML

Kinetic parameters, sensitivity example. Parameter identifiability
by the data in 'boxodes2.dat' after the estimation by 'boxodes2.nml'

Model:    The Box kinetics with varying temperature
Simulate: parameters (E,km)
Data:     measured values for all states
s0:       given as data in the data sets
```

```
 &files
  datafile(1) = 'boxodes2.dat'
  resultfile  = 'boxodse1.sen'
 /

 &problem
  task  = 'sen'
  model = 'ode'
  odesolver = 'odessa'
 /

 &modelpar
modelvar = 'k1mean   global   .725E+00
            E1       global   .676E-02
            k2mean   global   .217E+00
            E2       global   .117E-02
            Tmean    global   300.
            case     global   2.
            Temp     local    293.   313.;
            s01      initval  file;
            s02      initval  file;'

 target  = 'k1mean      0.5      0.9
            E1          0.001    0.02
            k2mean      0.1      0.4
            E2          0.001    0.02'

  nevastep = 12 12 12 12
  gridvar  = 'k1mean k2mean
              E1     E2'

  nstates = 2
  nsaux   = 3
 /
```

The meaning of most variables has already been explained. The only new input is *gridvar*. This character table gives the names of the target variable pairs whose contours are computed, inside the bounds determined by *target*.

1D least squares profiles and a 2D contour are presented below.

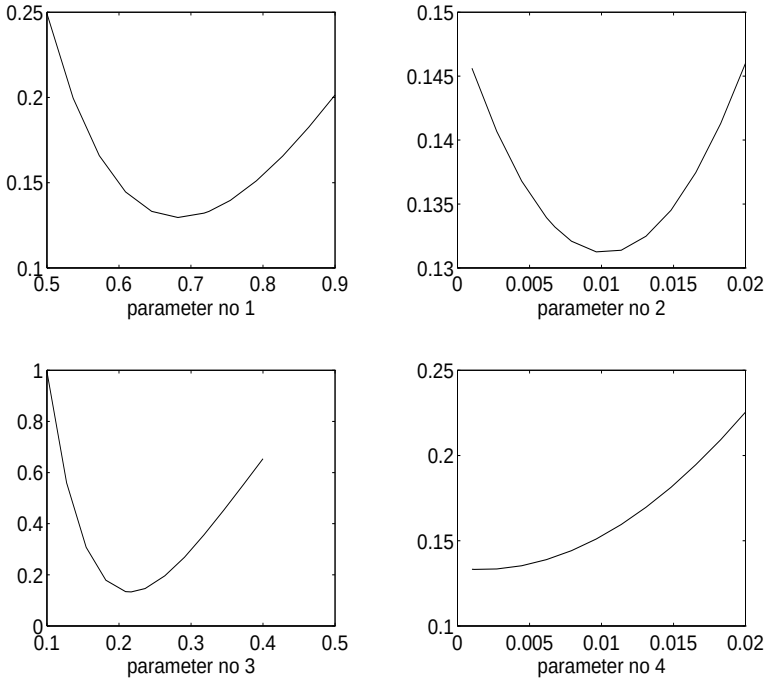


Figure 4.1. LSQ profiles for individual parameters.

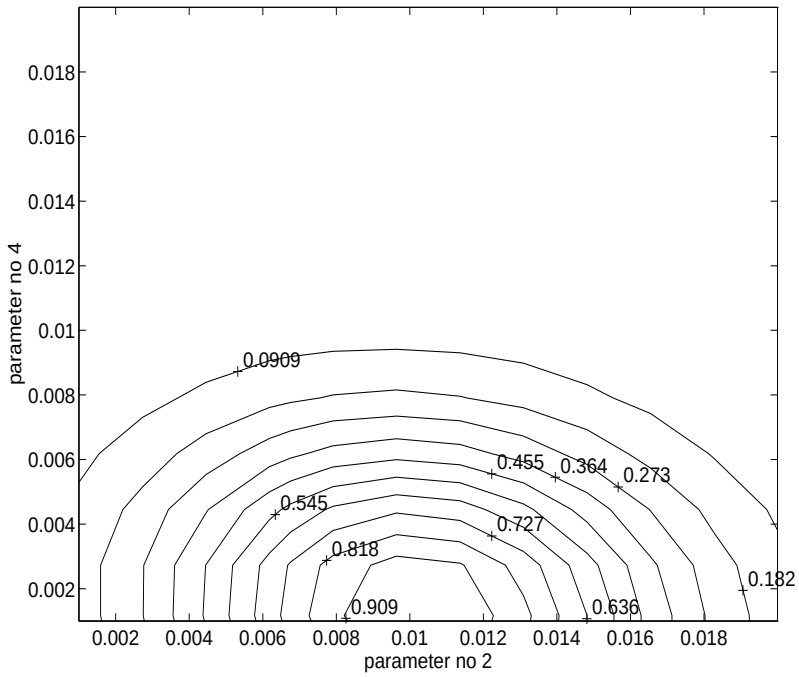


Figure 4.2. max. likelihood contour for activation energies

We notice that the activation energy of the second reaction, especially, is not yet well determined. The contour plotted presents the scaled maximum like-

lihood surface. The size of the pure experimental error is taken as estimated by the fit and given in the $reportfile$. For more details of the plotting macros see Appendix 2.

Optimal design

The goal of experimental design is to determine the values of x, i.e. the experimental conditions, which guarantee the most accurate values of θ with a minimum number of experiments. The accuracy is, at least in principle, determined by the confidence regions. While true nonlinear confidence regions only can be plotted in dimensions 1 and 2, we always may linearize the model with respect to θ and compute the volume of an approximative confidence region, the confidence ellipsoid. Clearly, a good design should make the volume small – all the 'good' θ points concentrated around the best fit, in a small region. Technically, the minimization of the approximative confidence volume leads to the maximization of the determinant of the *information matrix* of the model. The information matrix is numerically computable by the model equations. This is the most popular criterion, the criterion of D-optimality, used in the experimental design for mechanistic models.

If J denotes the *Jacobian matrix* of the model, whose components are the derivatives of the response variables with respect to the parameters,

$$J_{jp} = \frac{\partial y_p(x_j, \theta)}{\partial \theta_p},$$

the D–optimal design maximizes the objective function

$$\det |J^T J|$$

with respect to the variables x.

The eigenvalues of the information matrix $J^T J$ correspond to the lenghts of semiaxes of the confidence ellipsoid. The minimization of the volume thus amounts to the minimization of the product of the semiaxes. If some of the semiaxes are relatively small as compared to the largest ones, the product might be small and yet the ellipsoid quite elongated in some direction in the parameter space. Then the D–optimality does not produce a reasonable optimal design. Instead of minimizing the volume one might try to effect the shape of the confidence ellipsoid, e.g., by minimizing the ratio between the largest and smallest semiaxes. Considering the various possibilities of constructing objective funcions for optimal design leads to a number of criteria (A–, E–, G–, etc optimality, commonly named as 'alphabetical' criteria). See [3], [5] for more details and the Appendix for the criteria available in Modest.

Let us return to our example. In order to increase the accuracy of the parameters we optimize the conditions for the next experiment. The variables to be optimized are the temperature and the initial concentrations of both reactants A and B. They are allowed to vary in the limits $0 \leq c_A, c_B \leq 1.2$, $280 \leq T \leq 330$. The relevant input definitions for the optimization read as

```
Experimental design example

Model:    The Box kinetics with varying temperatures

Continuation for the estimation in 'boxodes1' (Arhenius parameters).
The parameter values used are those given by 'boxodes1'

Optimize: for a new batch, dataset 3:
          - the initial concentrations, s0
          - the temperature, T
for the purpose of accurate estimation of global parameters
k1mean,E1,k2mean,E2.

Data: measured values of all states, combined in one file

s0: given in data sets as known values (default obss0=0 used)
    The values in dataset 3 are the inital guess for optimization
    The initial guess for T is given in namelist 'modelpar' below


&files
 nsets       = 3
 datafile(1) = 'boxodex4.dat'
 resultfile  = 'boxodex4.exp'
/

&problem
 task  = 'exp'
 model = 'ode'
 odesolver = 'odessa'
 optimizer = 'simflex'
/

&modelpar
  modelvar = 'k1mean   global          .725E+00
              E1       global          .676E-02
              k2mean   global          .217E+00
```

```
              E2       global          .117E-02
              Tmean    global          300
              case     global          2
              Temp     local           293  313  290
              s0(1:2)  initval         file;'

   estvar  = 'k1mean
              E1
              k2mean
              E2       '

   target = 'Temp          3     280 310
             s0(1)         3     0   1.2
             s0(2)         3     0   1.2'

  nstates = 2
  nsaux = 3
/

&filepar
 combined = 1

 nobs(1)  = 10
 ncolxy(1) = 3
 nydata(1) = 2
 indx(1,1) = 1
 indy(1,1) = 2, indy(2,1) = 3
/

&print
 echo     = 1
 echodata = 1
 optmonit = 1
 jacout   = 1
/

for optcrit use here one of the following:
   D
   C
   T
for other choices see boxodex5.nml

&design
```

```
   optcrit = 'D'
 /

 &simflex
  abstols = 1.0e-16
  reltols = 1.0e-16
  sizes   = .2
  itmaxs  = 100
 /
```

The D–optimal suggestion turns out to be an experiment with minimal temperature and maximal initial concentrations. Just like in the task of estimation we could also here compute the contour plots to see more globally the sensitivity of the optimal point, possible local minima/maxima, etc. The only relevant difference to the estimation case would be to set $objfun =' exp'$ in the *problem* namelist, to overwrite the default setting $objfun =' lsq'$.

Following the idea of sequential design, the next task would be to perform the experiment at the point suggested by the optimization – i.e., in our synthetic example, to run the corresponding optimization and add random noise to the 'data'. Then we perform a new estimation for the parameter values, using the current values from the first estimation as the initial guesses.

Finally, we compute the sensitivity plots again by using all the data created so far and the updated parameter estimates as the reference values. Figures 5.1 and 5.2 exhibit the 1D and 2D least squares profile functions. All the parameters now seem to be well identified.

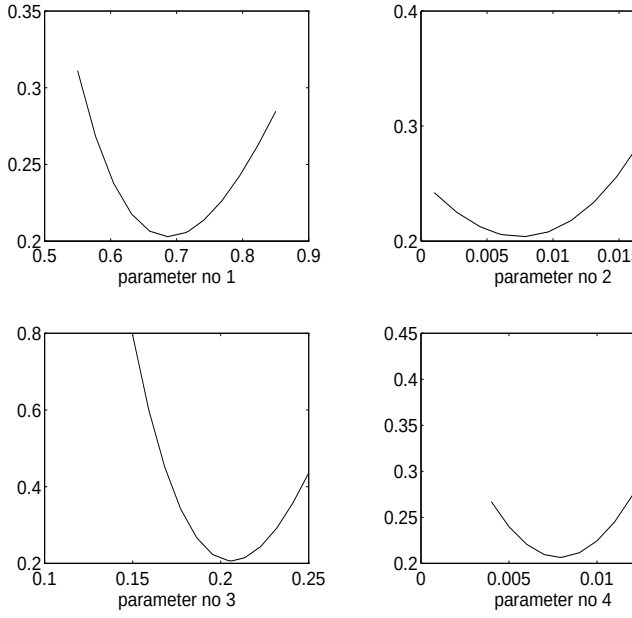


Figure 5.1

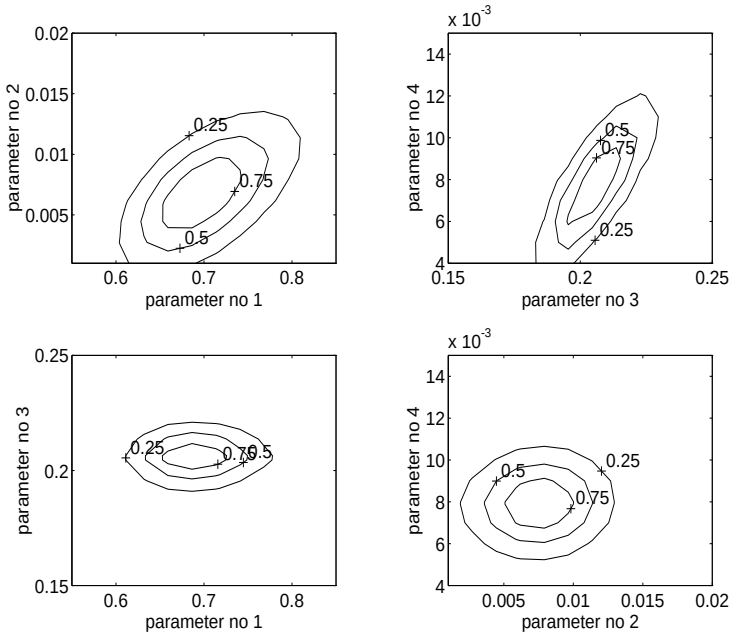


Figure 5.2

Optimization

Having now determined, with sufficient accuracy, the parameters of the model we may start using the model for various optimization tasks. As a simple demonstration example we consider the following problem: with fixed initial values and fixed time interval we should maximize the amount of intermediate stuff B at the end of the batch. Suppose temperature is the variable we may play with, assume we may set the value of it at 10 time points during the batch. The role of temperature is thus changed from the previous tasks: it is no more a *local* variable but a *control* variable.

Below are the settings for this optimization. Note that both the observation index and the data set index have to be specified for a control variable in the target table:

```
Optimal control example
Maximize the yield of the intermediate by temperature profile

Model:    The Box kinetics with controlled temperature
s0:       given in the data set
          The initial guess for T is given in the data set
```

```
&files
 nsets       = 1
 datafile(1) = 'boxodop1.dat'
 resultfile  = 'boxodop1.opt'
 optfile     = 'boxodop1.arg'
/

&problem
 task  = 'opt'
 model = 'ode'
 odesolver = 'odessa'
 optimizer = 'simflex'
/

&modelpar
  modelvar = 'k1mean   global          .725E+00
              E1       global          .676E-02
              k2mean   global          .217E+00
              E2       global          .117E-02
              Tmean    global         300
              case     global         2
              t        odevar
              Temp     control
              s0(1:2)  initval        file;'


   target = 'Temp      1  1     200 310
                       2  1     200 310
                       3  1     200 310
                       4  1     200 310
                       5  1     200 310
                       6  1     200 310
                       7  1     200 410
                       8  1     200 410
                       9  1     200 410
                      10  1     200 410'
  nstates = 2
  nsaux   = 3
/

&filepar
```

```
 nobs(1)  = 10
 ncolxy(1) = 4
 nydata(1) = 2
 indx(1,1) = 1,2
 indy(1,1) = 3, indy(2,1) = 4
/

&print
 echo     = 1
 echodata = 1
 optmonit = 1
 jacout   = 0
/

&design
/

&simflex
 abstols = 1.0e-16
 reltols = 1.0e-16
 sizes   = .2
 itmaxs  = 250
/
```

The optimal temperature profile suggested by the run is of a min/max type: to save the intermediate the batch is first run at a minimal reaction speed, and only at the end of the batch the production of B is boosted with a sharp temperature increase.

7 IO and user written files

The core executable, consisting of the user written modules and the library routines, reads a standard format input ascii file *modtemp.in* and writes standard format ascii files. The input can be created in two different ways, in PC by a Windows user interphase or, in all operating systems, by a Fortran namelist file. We first describe the namelist input, already partially exemplified in the above sections.

Namelist IO

The user edits a file, let us call it 'myprojec.nml', which has to contain all the namelist names given below. Some of the namelists contain information that has to be always given, some are more or less optional. One could classify the namelists in four types: the namelists that name and describe the input/output files ($files, filepar$), the namelists that specify the task, solvers and roles of model variables ($problem, modelpar$), the namelist for printing options ($print$) and the namelists for different solver parameters, one for each solver/optimizer ($simflex, levmar, newton, odessa, euler$).

```
A table for filling the namelists for different tasks of Modest,
notation:

   x     the namelist variables *must*  be edited for any given task
   o     the namelist variables are  *optional*
   -     the namelist variables are  *not needed*


                tasks

namelists      sim    est   exp   sen   opt


project         x     x     x     x     x
files           o     o     o     o     o
problem         x     x     x     x     x
modelpar        x     x     x     x     x
filepar         x     x     x     x     x
print           o     o     o     o     o
design          -     -     o     -     -
simflex         -     o     o     -     o
levmar          -     o     o     -     o
newton          o     o     o     o     o
odessa          o     o     o     o     o
euler           o     o     o     o     o


Description

project         projectname, username, date
files           all the input/output data file names and formats
```

problem	specification of the problem: task, model, solvers, object function type.
modelpar	dimensions of the model. model variable names, types and numerical values. Target variables for optimization, variable directions for simulations and sensitivity.
filepar	file parameters: n of data sets, n of observations, indices for x,y (control,response) variable columns.
print	echoing for model dimensions, data, optimization monitoring
design	choice of experimental design criteria
simflex, levmar	optimizer parameters
newton	nonlinear equation solver parameters
odessa, euler	Ode solver parameters

The variable names in each namelist and details on their use are given in Appendix 1.

If any input data files are needed, they of course must be named in the namelists *files*. In the *project* namelist the variable 'projectname' has a special role: it gives the generic names for all the *output files* used in the project, unless they are specified by the user differently. $result file$ is the basic numerical output file variable, produced in all tasks. So it must always possess a name value. However, if not given by the user, the name is created by the *projectname* and the abbreviation of the task. Similar policy applies to al the other output files, listed below in the example case where $projectname =' myproje'$.

	name	ending for tasks: sim	est	exp	sen	opt
resultfile	myproje.	sim	est	exp	sen	opt
reportfile	myproje.	-	sta	-	-	-
dumpfile	myproje.	dmp	dmp	dmp	dmp	dmp
optfile	myproje.	-	arg	arg	-	arg

Moreover, the *projectname* creates names for the Fortran subroutine templates that are produced by the IO routines.

Fortran subroutines

The user written model is created by editing two or three Fortran subroutines: the *model file*, the *observation file* and, in case the model is given by a differential equation system, the *initialization file*. Schematically, the model subroutines have the following contents for each model type

```
subroutine Falg/Fimp/Fode(argument list)
definitions for argument list
definitions for user variables

user written rows
  s(:) =   as given by user model   /Falg, algebraic model
 ds(:) =   as given by user model   /Fode, Ode system model
  f(:) =   as given by user model   /Fimp, implicit system model

return
end
```

Above, s denotes the states, ds the derivatives of the states and f the expression set equal to zero in the implicit algebraic equation system.

The observation subroutine always is given as

```
subroutine Observations(argument list)
definitions for argument list
definitions for user variables

  user written rows
  yest(:) =  as given by user model

return
end
```

and the initialization routine for Ode models as

```
subroutine Inits0(argument list)
definitions for argument list
definitions for user variables
```

```
        user written rows
        s(:) =  as given by user model

      return
      end
```

The user may edit the files as given above, using, e.g., some previous model as a template. However, to save the user from unnecessary routine work, the IO executable *nmlio* also produces – in addition to reading the namelists and creating the input ascii file *modtemp.in* – templates and an include file for the Fortran subroutines. The file names of the templates are (in case the projectname reads *myproje*, again)

```
myprojem.for         / model
myprojeo.for         / observations
myprojei.for         / initialization
```

The template for the model subroutine has the form

```
      subroutine Falg/Fimp/Fode(argument list)
      definitions for argument list
c     user definitions for user local variables here ...

      include 'myproje.dec'
c     user model rows here ....

      return
      end
```

The include file 'myproje.dec' contains automatically generated definitions for all the named variables given in the modelvar table in the namelist file. Moreover, it contains substitutions of variable values from the type tables internally used by the software to the names given by the user in at the (namelist or Windows) interphase level. In our example, the 'dec' file read as

```
real*8 time
real*8 k1mean
real*8 e1
real*8 k2mean
real*8 e2
real*8 tmean
real*8 temp
```

```
c
      time   = xdata(1,iobs)
      k1mean = gpar(1)
      e1     = gpar(2)
      k2mean = gpar(3)
      e2     = gpar(4)
      tmean  = gpar(5)
      temp   = lpar(1)
```

All the user has to do is to fill in the rows relevant for writing the model and give Fortran type definitions for local auxiliary variable names. The templates for observation and initialization routines are handled in similar way.

NOTE The variable names appearing in the argument lists of the above subroutines are internally used by the code and *must not be used as model variable names in the user written code.*

Creating and running the executable

Working in a modelling project the user typically has to take care of the data files, the input definitions for each run, and the model subroutines. A typical session required to create and run the executable for a new project model might go as follows

1. Collect and name the relevant model variables. Classify the variable types.

2. Write the user written part of the model codes

3. Edit the namelist for a first run

4. Run the namelist with 'nmlio' executable (only). This is done to create the Fortran templates and the declaration include file.

5. Add the user written codes to the templates.

6. Compile the code.

7. You are now ready to run the tasks given by namelist files using the 'nmlio' and 'core' executables jointly.

If the user makes any changes in the code, it naturally has to be re–compiled and linked. However, it might go unnoticed that certain changes in the namelist file also generate a changed code: any new or removed variable definition or change of type in the *modelvar* table creates a change in the 'dec' file. To warn the user of mistakes here the *nmlio* program writes a flag at the beginning of the input ascii file indicating whether the 'dec' file is changed since the last run. If so, the Modest core executable refuses to run the task and instead urges the user to compile the code.

8 References

References

[1] W. E. Schiesser: *The Numerical Method of Lines. Integration of Partial Differential Equations.* Academic Press, Inc. 1991.

[2] PC–MATLAB User's Guide. The Math Works, Inc.,

[3] Atkinson, A.T. and A.N. Donev: *Optimum Experimental Design.* Oxford Stat. Sci. Ser., Clarendon Press, Oxford, 1992.

[4] Bard: *Nonlinear Regression.* Academic Press,1974.

[5] Seber,G.A.F. and C.J. Wild: *Nonlinear Regression* Wiley & Sons, 1989.

[6] Dobson, A.J.: An Introduction to Statistical Modelling, Chapman & Hall, 1983.