Documentation for SEARCH, Julia Version

Introduction

SEARCH is a computer program for function optimization. An earlier version was written in Fortran. The Julia version described here incorporates various improvements. SEARCH permits bounds and linear constraints to be imposed on parameters and, in statistical applications, computes asymptotic standard errors and correlations of parameter estimates. On an algorithmic level, SEARCH follows closely the prescriptions set out in a classic paper by Powell [17] on recursive quadratic programming. There are, however, two major departures in SEARCH from what Powell describes. One is the application of the sweep operator for solving quadratic programming subproblems [12, 13]. The other is SEARCH's added flexibility in approximating the Hessian matrix (second differential) of the objective function. In many statistical applications, the expected information matrix is available and provides an excellent approximate Hessian. For example, it is straightforward to compute the expected information matrix for any exponential family of probability distributions [11]. For problems having no natural approximate Hessian, SEARCH iteratively constructs an approximation by Davidon's quasi-Newton update [6]. SEARCH can also be used to sample a function over a user defined grid of points without attempting to fully optimize the function.

Setting up a problem for SEARCH is relatively simple. You must define two Julia functions, which we generically call **initial** and **fun**. In **fun** you must supply the function to be optimized and possibly its first two differentials. Exact derivatives are always helpful if correctly specified. Initial estimates of parameters, their bounds, constraints, and names should be assigned in **initial**. If you wish to evaluate the function over a grid in parameter space, then the grid must be defined in **initial** as well. Reasonable default values are provided by SEARCH for most vector variables, so the code in **initial** is often fairly short.

Fourteen sample problems of how SEARCH can be found in the Julia program search_examples.jl. Output is collected in the file Search.out. The first ten problems rely on the Davidon update to the Hessian. Problem 11 is a typical non-linear regression problem, problems 12 and 13 are maximum likelihood problems, and problem 14 is an exercise in adaptive robust re-

gression. Each of these problems will contribute to your understanding of the mechanics of using SEARCH.

Setting up a Problem in SEARCH

To call SEARCH, you should start with the code

```
include("search.jl")
using ParameterData
using Optimize
keyword = Dict{ASCIIString, Any}()
keyword = optimization_dictionary(keyword)
outfile = "Search.out"
io = open(outfile, "w")
keyword["output_unit"] = io
```

from the sample application. The first three lines alert Julia to the existence of SEARCH and its two organizing modules **ParameterData** and **Optimize**. If search.jl exists outside your working directory, then the include command must give the full path name. Lines 4 and 5 above construct a dictionary of keywords controlling the operation of SEARCH. The last three lines of the code are optional and redirect output from your terminal (SEARCH's default) to the named file Search.out. At the bottom of the sample program calling SEARCH, you will find the code

```
parameter = initial1(keyword)
(best_point, best_value) = optimize(fun1, parameter)
parameter = initial2(keyword)
(best_point, best_value) = optimize(fun2, parameter)
parameter = initial3(keyword)
(best_point, best_value) = optimize(fun3, parameter)
parameter = initial4(keyword)
(best_point, best_value) = optimize(fun4, parameter)
parameter = initial5(keyword)
(best_point, best_value) = optimize(fun5, parameter)
parameter = initial6(keyword)
(best_point, best_value) = optimize(fun6, parameter)
parameter = initial7(keyword)
(best_point, best_value) = optimize(fun7, parameter)
parameter = initial8(keyword)
```

```
(best_point, best_value) = optimize(fun8, parameter)
parameter = initial9(keyword)
(best_point, best_value) = optimize(fun9, parameter)
parameter = initial10(keyword)
(best_point, best_value) = optimize(fun10, parameter)
parameter = initial11(keyword)
(best_point, best_value) = optimize(fun11, parameter)
parameter = initial12(keyword)
(best_point, best_value) = optimize(fun12, parameter)
parameter = initial13(keyword)
(best_point, best_value) = optimize(fun13, parameter)
parameter = initial14(keyword)
(best_point, best_value) = optimize(fun14, parameter)
close(io)
```

running the various problems and closing the output file. The variables best_point and best_value return the optimal point and the optimal function value for each problem.

As already mentioned, the program that calls SEARCH must include the functions **initial** and **fun**. SEARCH visits **initial** once before commencing optimization or grid evaluation. Consider the following code from **initial11** of search_examples.jl on nonlinear regression:

```
function initial11(keyword::Dict{ASCIIString, Any})
#
#  Reset defaults for relevant keywords.
#
  keyword["goal"] = "minimize"
  keyword["constraints"] = 0
  keyword["parameters"] = 4
  keyword["cases"] = 21
  keyword["standard_errors"] = true
  keyword["title"] = "nonlinear least squares"
#
# Create the parameter data structure and change defaults.
#
  parameter = set_parameter_defaults(keyword)
  parameter.par[1] = 10.
  parameter.par[2] = -0.1
  parameter.par[3] = 5.
  parameter.par[4] = -0.01
```

```
  parameter.max[2] = 0.
  parameter.max[4] = 0.
  return parameter
end
```

The first two commands of **initial11** are unnecessary since SEARCH's default is minimization without constraints. The next two commands set the number of parameters to 4 and the number of cases (in least squares) to 21. Lines 5 and 6 ask for parameter standard errors and create a problem title. Once these decisions are made, the parameters are initialized and their bounds set. All pertinent information is then returned to SEARCH in the parameter data structure.

If you want SEARCH to sample function values, then set the variable travel to "grid" and define the grid points by appropriate code in **initial**. For instance, the code

```
  n = parameter.points
  p = parameter.parameters
  parameter.grid = rand(n, p)
```

specifies a uniformly distributed random grid of n points over the cube $[0,1]^p$. By convention, SEARCH's interprets the plural of noun as the number of instances of that noun.

SEARCH provides default values for the arrays in **initial**. The grid and parameter arrays are filled with 0's; parameter names are set to par 1, par 2, and so forth. Parameter lower bounds are set to $-\infty$ and upper bounds to $+\infty$. The constraint coefficients in parameter.constraint and the constraint levels in parameter.constraint_level are filled with 0's. Unless you are performing regression, parameter.cases should be 0. Julia deduces the type of a variable from the values that fill it. Thus, avoid decimal numbers in initializing integer variables such as parameter.constraints. The variables parameter.travel and parameter.title are character strings. The variable parameter.standard_errors is logical.

In problem 12 on allele frequency estimation, we constrain the parameters to sum to 1.0. This is accomplished via the commands

```
  fill!(parameter.constraint, 1.0)
  parameter.constraint_level[1] = 1.0
```

in **initial12**. Fixing a parameter at a specific value during a search can be accomplished by adding a linear equality constraint. For instance, to fix parameter 3 in problem 12 at the value 1/3, you should create a second constraint by equating parameter.constraints to 2 and including the code

```
  parameter.constraint[2, 3] = 1.0
  parameter.constraint_level[2] = 1.0 / 3.0
```

in **initial12**. The first argument of parameter.constraint conveys the constraint index and the second argument the component to be revised. Note that indices for Julia arrays start at 1 rather than 0.

The function **fun** is visited multiple times to retrieve the objective function and its first two differentials. The current parameter values are passed in the double precision vector par, an abbreviation for parameter.par. Either differential is optional. At the bottom of **fun**, one of the three commands: return (f, nothing, nothing), return (f, df, nothing), or return (f, df, d2f) must appear, depending on how many differentials are computed. The code

```
function fun1(par::Vector{Float64})
#
# Define a function to be optimized. Include as needed
# the first and second differentials.
#
   pars = length(par)
   df = zeros(pars)
   f = 100 * (par[2] - par[1]^2)^2 + (1.0 - par[1])^2
   df[1] = -400.0 * (par[2] - par[1]^2) * par[1] -
     2.0 * (1.0 - par[1])
   df[2] = 200.0 * (par[2] - par[1]^2)
   return (f, df, nothing)
end
```

from **fun1** clearly conveys how one defines the objective function and its first differential. In this instance, SEARCH implements quasi-Newton minimization to estimate the two parameters. The approximate Hessian starts as the identity matrix.

Objective functions can be quite elaborate. The more complicated the function, the more attractive it becomes to let SEARCH compute derivatives numerically. The code

```
function fun11(par::Vector{Float64})
#
# Define a function to be optimized. Include as needed
# the first and second differentials.
#
  count = [15.1117,11.3601,9.7652,9.0935,8.4820,7.6891,
```

```
    7.3342,7.0593,6.7041,6.4313,6.1554,5.9940,5.7698,
    5.6440,5.3915,5.0938,4.8717,4.5996,4.4968,4.3602,4.2668]
  weight = [.004379,.007749,.010487,.012093,.013900,.016914,
    .018591,.020067,.022249,.024177,.026393,.027833,.030039,
    .031392,.034402,.038540,.042135,.047267,.049453,.052600,
    .054928]
  time = [2.,4.,6.,8.,10.,15.,20.,25.,30.,40.,50.,60.,70.,
    80.,90.,110.,130.,150.,160.,170.,180.]
#
# This problem of exponential fitting revolves around the
# Gauss-Newton method.
#
  pars = length(par)
  f = 0.0
  df = zeros(pars)
  d2f = zeros(pars,pars)
  dg = zeros(pars)
#
# Loop over all cases. The regression function is denoted by g.
#
  for i = 1:length(count)
    g = par[1] * exp(par[2] * time[i]) +
      par[3] * exp(par[4] * time[i])
    dg[1] = exp(par[2] * time[i])
    dg[2] = time[i]* par[1]* exp(par[2] * time[i])
    dg[3] = exp(par[4] * time[i])
    dg[4] = time[i] * par[3] * exp(par[4] * time[i])
    residual = count[i] - g
    f = f + weight[i] * residual^2
    df = df - 2.0 * weight[i] * residual * dg
    d2f = d2f + 2.0 * weight[i] * dg * dg'
  end
  return(f, df, d2f)
end
```

defining problem 11 on nonlinear regression illustrates the Gauss-Newton method and how it approximates the Hessian matrix. More will be said about this later. For convenience **fun11** stores the data in addition to performing its primary tasks. More complex problems require global arrays to store data and longer calling programs.

Hessian Approximations

The second-order information encoded in the Hessian matrix is crucial in achieving fast convergence. For maximum likelihood problems, Fisher's scoring algorithm approximates the observed information matrix (Hessian of the negative loglikelihood) by the expected information matrix. This substitution is advantageous because it is often far easier to compute the expected information matrix than the observed information matrix. Indeed, the former can be expressed exclusively in terms of the first differential (score) of the loglikelihood. Furthermore, the expected information matrix is automatically positive definite. Away from the maximum likelihood point, the observed information can be negative definite, and Newton's method sometimes moves downhill instead of uphill.

Table 1: Score and Information for Some Exponential Families

Family	$L(\theta)$	$\nabla L(\theta)$	$J(\theta)$
Binomial	$x \ln \frac{p}{1-p} + n \ln(1-p)$	$\frac{x-np}{p(1-p)} \nabla p$	$\frac{n}{p(1-p)} \nabla p dp$
Multinomial	$\sum_i x_i \ln p_i$	$\sum_i \frac{x_i}{p_i} \nabla p_i$	$\sum_i \frac{n}{p_i} \nabla p_i dp_i$
Poisson	$-\mu + x \ln \mu$	$-\nabla \mu + \frac{x}{\mu} \nabla \mu$	$\frac{1}{\mu} \nabla \mu d\mu$
Exponential	$-\ln \mu - \frac{x}{\mu}$	$-\frac{1}{\mu} \nabla \mu + \frac{x}{\mu^2} \nabla \mu$	$\frac{1}{\mu^2} \nabla \mu d\mu$

Computation of the expected information matrix is straightforward for exponential families of distributions [11]. Because of the form of the expected information matrix, several authors have noted the close relationship between the scoring algorithm and iteratively reweighted least squares [3, 4, 5, 9, 11, 16]. This fact forms the basis of the popular statistical package GLIM [1]. However, it is sometimes awkward to fit maximum likelihood problems into the framework of iteratively reweighted least squares, and the present approach offers greater flexibility.

Some examples of expected information matrices are set out in Table 1. In the table dp denotes the differential of the function p. This is a row vector with the gradient ∇p as transpose. Thus, $\nabla p \, dp$ is a rank-one matrix. For the binomial distribution, p is the success probability, n is the number of trials, and x is the observed number of successes. The same conventions hold

for the multinomial distribution on a category-by-category basis, provided the entries of the table are summed over all categories. For the Poisson and exponential distributions, x is the observed random variable and μ is its mean.

As an example, consider again the typical multinomial problem of estimating allele frequencies at the ABO locus. The code below appears in function **fun12**.

```
count = [182., 60., 17., 176.]
pars = length(par)
cases = sum(count)
f = 0.0
df = zeros(pars)
d2f = zeros(pars,pars)
for i = 1:4
   dq = zeros(pars)
   if i == 1
      q = par[1]^2 + 2.0 * par[1] * par[3]
      dq[1] = 2.0 * (par[1] + par[3])
      dq[3] = 2.0 * par[1]
   elseif i == 2
      q = par[2]^2 + 2.0 * par[2] * par[3]
      dq[2] = 2.0 * (par[2] + par[3])
      dq[3] = 2.0 * par[2]
   elseif i == 3
      q = 2.0 * par[1] * par[2]
      dq[1] = 2.0 * par[2]
      dq[2] = 2.0 * par[1]
   else
      q = par[3]^2
      dq[3] = 2.0 * par[3]
   end
   f = f + count[i] * log(q)
   df = df + count[i]* dq / q
   d2f = d2f - (cases / q) * dq * dq'
end
return(f, df, d2f)
```

In this maximum likelihood problem, we set

```
keyword["goal"] = "maximize"
```

One can also work problem 12 without computing either first or second derivatives. To see the impact of this choice, just replace the final statement of **fun12** by return(f, nothing, nothing). The same maximum point is reached, but parameter asymptotic standard errors change slightly. The substitution return(f, df, nothing) produces asymptotic standard errors closer to the original choice return(f, df, d2f).

For weighted least squares problems, one can approximate the Hessian of the weighted residual sum of squares by the usual Gauss-Newton formula

$$d^2 f \approx 2\sum_{i=1}^{n} w_i dg_i \nabla g_i,$$

where w_i and g_i are respectively the weight and the regression function for case i of n cases. The Gauss-Newton algorithm amounts to scoring except that the hidden variance parameter is not explicitly updated at each iteration. For this reason, computation of asymptotic standard errors requires an adjustment to the inverse of the approximate Hessian. Providing SEARCH with the number of cases allows it to perform the necessary adjustment. If you wish to do weighted least squares, only the regression function g_i and its first differential need need to be changed in **fun11**. For unweighted least squares, you should set each case weight equal to 1. The Gauss-Newton algorithm converges in one step for linear least squares.

Typical Output of SEARCH

We list below the output on ABO allele frequency estimation from problem 12 of search_examples.jl. When you run SEARCH, it is always advisable to check that the echoed parameter bounds, linear equality constraints, initial values, and names agree with the values you assumed are coded in **initial**. The output file also echoes a problem title, the choice of minimization or maximization, and whether the grid or search option has been chosen. If a search is successful, then the parameter estimates appear at the final iteration of the output. The parameter asymptotic standard errors and correlations in this example are based on the inverse of the expected information matrix. In some problems, not this one, standard errors are given as 0. This happens when a parameter estimate coincides with a boundary. Asymptotic correlations are also given as 0 whenever either parameter coincides with a boundary. When both parameters occupy interior points, their asymptotic

correlation reflects the extent to which independent information on them is available in the data.

```
Title: ABO frequency estimation
Grid or search option: search
Minimize or maximize: maximize

Parameter minima and maxima:

    A freq      B freq      O freq

  1.0000e-06  1.0000e-06  1.0000e-06
     Inf         Inf         Inf

Parameter constraints:

    A freq      B freq      O freq      level

  1.0000e+00  1.0000e+00  1.0000e+00  1.0000e+00

iter  steps   function    A freq      B freq      O freq

  1    0   -6.7815e+02   3.3333e-01  3.3333e-01  3.3333e-01

  2    0   -5.2256e+02   2.4598e-01  3.5632e-02  7.1839e-01

  3    0   -4.9256e+02   2.6383e-01  9.1204e-02  6.4497e-01

  4    0   -4.9254e+02   2.6444e-01  9.3159e-02  6.4240e-01

  5    0   -4.9254e+02   2.6444e-01  9.3169e-02  6.4239e-01

  6    0   -4.9254e+02   2.6444e-01  9.3169e-02  6.4239e-01

  7    0   -4.9254e+02   2.6444e-01  9.3169e-02  6.4239e-01

  8    1   -4.9254e+02   2.6444e-01  9.3169e-02  6.4239e-01

The maximum function value of -492.53532 occurs at iteration 8.
```

```
The asymptotic standard errors of the parameters:

    A freq      B freq      O freq

  1.6218e-02  1.0100e-02  1.7576e-02

The asymptotic correlation matrix of the parameters:

    A freq      B freq      O freq

    1.0000

   -0.1713      1.0000

   -0.8243     -0.4166      1.0000
```

Sample Applications of SEARCH

Fourteen sample problems illustrate the mechanics of setting up a SEARCH run. Hock and Schittkowski [10] list a number of test problems in nonlinear optimization. SEARCH reproduces problems 1, 3, 4, 5, 9, 21, 28, 36, 24, and 38 from this source. The correct minima are reached in all instances. Many of the problems have both bounds and linear equality constraints. The code requires little explication except for mentioning that linear inequality constraints occur in a few problems. Linear inequality constraints are accommodated by introducing a dummy parameter for each inequality expression. For instance, in problem 6 (Hock and Schittkowski problem 21) there are originally only two parameters. However, it is mandated that the linear combination 10∗par[1] - par[2] be at least 10. This linear inequality can be dealt with by introducing a third parameter, a lower bound on this dummy parameter, and a linear equality constraint that forces the dummy parameter to equal 10∗par[1]-par[2]. The relevant code from **initial6** is

```
parameter.par[1] = 3.
parameter.par[2] = 1.
parameter.par[3] = 29.
parameter.min[3] = 10.
parameter.constraint[1,1] = 10.
```

```
parameter.constraint[1,2] = -1.
parameter.constraint[1,3] = -1.
```

By default, parameter.constraint_level[1] = 0.0.

Problem 11 involves nonlinear least squares. Various aspects of this example have been discussed previously. The original problem comes from the BMDP manual [7], where a sums of exponentials model is fitted to data on radioactivity in the blood of a baboon. (See the writeup of the program BMDP3R.) The response variable is time, and the predictor variable is radioactive counts divided by 10,000. A weight is assigned to each case. The code found in **fun11** for this problem was discussed earlier. Since the eigenvalue parameters 2 and 4 must be negative, we take the precaution of imposing maxima of 0 on them in **initial11**. Our converged parameter estimates and their asymptotic standard errors match those in the BMDP manual.

The genetics example of problem 12, was discussed previously and is featured in the book [18]. There are three alleles - A, B, and O - at the ABO locus. Phenotype A corresponds to the genotypes A/A and A/O, phenotype B to the genotypes B/B and B/O, phenotype AB to the genotype A/B, and phenotype O to the genotype O/O. The frequency of a genotype is determined by the Hardy-Weinberg product law for allele frequencies. In the present data set, there are 182 people with phenotype A, 60 with phenotype B, 17 with phenotype AB, and 176 with phenotype O. Our parameter estimates match those of Rao [18]. Our parameter asymptotic standard errors also match Rao's. When the second differential is not supplied, and the default quasi-Newton algorithm is applied, there are minor differences in parameter asymptotic standard errors because SEARCH now employs the observed information matrix. The quasi-Newton option also shows an inferior rate of convergence compared with scoring.

In survival analysis, observations are often censored owing to the limited time a patient can be followed. Sample problem 13 analyzes clinical trial data on the efficacy of a drug in prolonging remissions of acute leukemia [8]. In **initial13** patients are divided into two groups of 21 patients each. Every patient either dies sometime during the study, and his time of death is recorded in the variable **time**; or he is alive at the end of the observation period, and **time** specifies the total observation time. A very simple and convenient model is to suppose the survival times are exponentially distributed for the treatment and control groups, with a different mean for each group. Those individuals who die before the end of the study represent observations from one or the other of these two exponential distributions.

Those individuals who survive beyond the end of the study represent observations from binomial distributions whose success probabilities are just the right tail probabilities of the corresponding exponential distributions. This verbal description makes it clear that two fundamentally different likelihood models are necessary, depending on whether an individual is censored or uncensored. These ideas are implemented in **fun13**, drawing on the entries of our table of loglikelihoods, scores, and expected information matrices. Note that the binomial distributions used have exactly one success and no failures.

Our code imposes the constraint par[1] = par[2]. If a second search is conducted dropping the constraint, then one can test the null hypothesis. Twice the difference in loglikelihoods, 2 (116.767-108.524) = 16.486, asymptotically follows a chi-square distribution with one degree of freedom. On the basis of this likelihood ratio test, one can reject the hypothesis that the two groups have the same mean survival times.

Lange and Sinsheimer [15] develop a theory of adaptive robust regression based on the slash distribution. Problem 14 reproduces their analysis of some classical data of Quetelet, who examined births and deaths by hour over a thirty year period at a certain hospital in Brussels. Following Berk [2], we postulate a simple linear regression model with deaths per hour as the response and births per hour as the predictor. Two apparent outliers, one at noon and the other at midnight, create the spurious impression that births and deaths are correlated. The adaptive robust method used here gives a much smaller and nonsignificant regression coefficient of deaths on births. Ordinary least squares suggests a significant regression coefficient. Although standard errors for the robust model are omitted here, they are available with additional computation [15]. Close examination of the SEARCH output reveals that most iterations take the maximum number of steps. In this problem one should turn off step halving by setting **max_steps** = 0 within search.jl. The EM algorithm employed in problem 13 guarantees that the observed loglikelihood increases at each iteration.

The EM algorithm alternates between updating the regression coefficients and ν and σ, the slash and scale parameters. The latter two parameters are updated in **fun14**. They are defined as global variables in **initial14** and initialized there. A similar algorithm is available for the t distribution [14]. Both of the slash and the t algorithms update the regression coefficients by weighted least squares. The parameter ν characterizes the degree of departure from normally distributed errors. As ν tends to ∞, sampling errors become more normal. As ν tends to 0, the errors exhibit increasing kurtosis. Our implementation of the EM algorithm requires the logarithm

of the incomplete gamma integral and its derivative. Code is appended to SEARCH to compute these special functions.

Fine-Tuning Convergence

If you have difficulty obtaining convergence during a search, we offer the following suggestions. First, make certain you understand your model and the optimization problem it entails. Check that your Julia code in **fun** correctly defines the function you wish to optimize. If you provide the partial derivatives of the function, check this code as well. It does no harm to work a problem both with and without supplying derivatives. Comparison of the two results may indicate that the derivatives are coded improperly. You may wonder why the option of furnishing exact derivatives is provided at all. In some cases SEARCH will take appreciably less time per iteration if exact derivatives are provided. Furthermore, there is always some error in approximating derivatives numerically. In a few problems where convergence or computation of parameter asymptotic standard errors is a delicate matter, exact derivatives can make a difference.

If you are certain the function and its derivatives are defined properly, and you still have difficulty obtaining convergence, then do not lose hope. You might want to try different starting values for the parameters. In many problems there are simple and natural estimates of parameters. Even if these estimates are suboptimal relative to maximum likelihood or least squares estimates, they may furnish ideal starting values to insure rapid and stable convergence. To prevent SEARCH from taking inordinately large steps and drifting away from the region of the true optimum, one can specify a finite value for the variable **max_step_length** in search.jl. Parameter increments exceeding this maximum are then contracted.

Another helpful tactic is to put reasonable and fairly tight bounds on some of the parameters. If at convergence, one of the parameters ends up on what you consider an artificial boundary, try enlarging the bounds on this parameter, but now start SEARCH from where you just converged. In many cases there are natural boundaries outside of which your function may not be defined. Be certain to define the parameter bounds slightly inside these natural bounds. It is fairly common for intermediate iterations to occur on a boundary even if the true optimum is not on the boundary.

Another device to force smaller steps is to increase the number of allowable step decrements per iteration. The actual number of steps taken per

iteration is reported under the heading **steps** in the output. If in a search the number of steps at some iteration attains the maximum allowable, but the function value does not decrease at this iteration, then you might try increasing **max_steps** and rerunning SEARCH. This may result in very short steps while a decent approximation to the Hessian of your function is built up. Provided derivatives are computed accurately, a sufficient number of step decrements is guaranteed to give a decrease in function value. With short steps, it may take a large number of iterations to achieve convergence. The variable **max_iters** in SEARCH can be reset to a higher value in such unusual circumstances.

If SEARCH appears to be converging prematurely, then you can manipulate the convergence criterion to force it to take more iterations. Convergence within SEARCH is controlled by the two variables, **conv_crit** and **conv_tests**. If the absolute difference in function values between the previous and current iterations is less than **conv_crit** for **conv_tests** consecutive iterations, then SEARCH stops. Both of these variables can be reset by users in the search.jl code.

SEARCH can and will converge to a local optimum that is greater than the global optimum. This is a problem that besets all optimization algorithms that depend on local information like derivatives and Hessians. In some instances, one can rule out multiple optima on the basis of convexity or other considerations. In the absence of a priori guarantees of a single optimum, the only remedy is to restart the search algorithm at a number of different points in parameter space. A desperation move is to sample the function surface on a large random grid of points.

Occasionally, SEARCH will output a message to the effect that the parameter asymptotic covariance matrix cannot be computed. This may be an indication that the likelihood surface is extremely flat or that adequate convergence to a optimum has not occurred. In some problems it is possible that all parameters cannot be separately estimated. Such problems are said to be under-identified. Review your model and check that all parameters can be estimated when you get this message.

References

[1] Baker RJ, Nelder JA (1978) *The GLIM System, Release 3.* Numerical Algorithms Group, Oxford

[2] Berk RA (1989) A primer on robust regression. *UCLA Statistics Series* 12

[3] Bradley EL (1973) The equivalence of maximum likelihood and weighted least squares estimates in the exponential family. *J Amer Stat Assoc* 68:199–200

[4] Charnes A, Frome EL, Yu PL (1976) The equivalence of generalized least squares and maximum likelihood in the exponential family. *J Amer Stat Assoc* 71:169–171

[5] Cox C (1985) Computation of maximum likelihood estimates by iteratively reweighted least squares: a spectrum of examples. *BMDP Technical Report 82* BMDP Statistical Software, Los Angeles

[6] Dennis JE Jr, More JJ (1977) Quasi-Newton methods, motivation and theory. *SIAM Review* 19:46–89

[7] Dixon WD, Brown MB, Engelman L, Frane JW, Hill MA, Jennrich RI, Toporek JD (1983) *BMDP Statistical Software 1983*. University of California Press

[8] Gehan EA (1965) A generalized Wilcoxon test for comparing arbitrarily singly-censored samples. *Biometrika* 52:203–223

[9] Green PJ (1984) Iteratively reweighted least squares for maximum likelihood estimation and some robust and resistant alternatives (with discussion). *J Roy Stat Soc B* 46:149–192

[10] Hock W, Schittkowski K (1981) Test Examples for Nonlinear Programming Codes. *Lecture Notes in Economics and Mathematical Systems*, Springer

[11] Jennrich RI, Moore RH: Maximum likelihood estimation by means of nonlinear least squares. *Proc Stat Comput Section, Amer Stat Assoc* 57–65

[12] Jennrich RI, Sampson PF (1978) Some problems faced in making a variance component algorithm into a general mixed model program. *Proceedings of the Eleventh Annual Symposium on the Interface*. Gallant AR, Gerig TM Editors, Institute of Statistics, North Carolina State University

[13] Lange K (2010) *Numerical Analysis for Statisticians*, 2nd ed. Springer, New York

[14] Lange K, Little RJA, Taylor JMG (1989) Robust modeling using the t distribution. *J Amer Stat Assoc* 84:881–896

[15] Lange K, Sinsheimer JS (1993) Normal/independent distributions and their applications in robust regression. *J Comput Graphical Stat* 2:175–198

[16] Nelder JA, Wedderburn RWM (1972) Generalized linear models. *J Roy Stat Soc Ser A* 135:370–384

[17] Powell MJD (1978) A fast algorithm for nonlinearly constrained optimization calculations. *Proceedings of the 1977 Dundee Conference on Numerical Analysis*, Watson GA Editor, Berlin, Springer

[18] Rao CR (1973) *Linear Statistical Inference and Its Applications. 2nd ed* New York, Wiley