Package ‘rgenoud’

January 24, 2019

Version 5.8-3.0
Date 2019-01-22
Title R Version of GENetic Optimization Using Derivatives
Author Walter R. Mebane, Jr. <wmebane@umich.edu>,
Jasjeet Singh Sekhon <sekhon@berkeley.edu>
Maintainer Jasjeet Singh Sekhon <sekhon@berkeley.edu>
Description A genetic algorithm plus derivative optimizer.
Depends R (>= 2.15), utils
Suggests testthat
License GPL-3
URL http://sekhon.berkeley.edu/rgenoud/
NeedsCompilation yes
Repository CRAN
Date/Publication 2019-01-24 22:40:02 UTC

R topics documented:

  genoud .......................................................... 1

Index

<table>
<thead>
<tr>
<th>genoud</th>
<th>GENetic Optimization Using Derivatives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GENetic Optimization Using Derivatives</td>
</tr>
</tbody>
</table>

Description

Genoud is a function that combines evolutionary search algorithms with derivative-based (Newton or quasi-Newton) methods to solve difficult optimization problems. Genoud may also be used for optimization problems for which derivatives do not exist. Genoud, via the cluster option, supports the use of multiple computers, CPUs or cores to perform parallel computations.
Usage

```r
genoud(fn, nvars, max=FALSE, pop.size=1000, max.generations=100,
wait.generations=10, hard.generation.limit=TRUE, starting.values=NULL,
MemoryMatrix=TRUE, Domains=NULL, default.domains=10,
solution.tolerance=0.001, gr=NULL, boundary.enforcement=0, lexical=FALSE,
gradient.check=TRUE, BFGS=TRUE, data.type.int=FALSE, hessian=FALSE,
unif.seed=round(runif(1, 1, 2147483647L)),
int.seed=round(runif(1, 1, 2147483647L)), print.level=2, share.type=0,
instance.number=0, output.path="stdout", output.append=FALSE,
project.path=NULL, P1=50, P2=50, P3=50, P4=50, P5=50, P6=50, P7=50,
P8=50, P9=0, P9mix=NULL, BFGSburnin=0, BFGSfn=NULL, BFGShelp=NULL,
control=list(), optim.method=ifelse(boundary.enforcement < 2, "BFGS",
"L=BFGS-B"), transform=FALSE, debug=FALSE, cluster=FALSE, balance=FALSE,
...)
```

Arguments

**fn**  
The function to be minimized (or maximized if max=TRUE). The first argument of the function must be the vector of parameters over which minimizing is to occur. The function must return a scalar result (unless lexical=TRUE).

For example, if we wish to maximize the sin() function. We can simply call genoud by `genoud(sin, nvars=1, max=TRUE)`.

**nvars**  
The number of parameters to be selected for the function to be minimized (or maximized).

**max**  
Maximization (TRUE) or Minimizing (FALSE). Determines if genoud minimizes or maximizes the objective function.

**pop.size**  
Population Size. This is the number of individuals genoud uses to solve the optimization problem. There are several restrictions on what the value of this number can be. No matter what population size the user requests, the number is automatically adjusted to make certain that the relevant restrictions are satisfied. These restrictions originate in what is required by several of the operators. In particular, operators 6 (Simple Crossover) and 8 (Heuristic Crossover) require an even number of individuals to work on—i.e., they require two parents. Therefore, the pop.size variable and the operators sets must be such that these three operators have an even number of individuals to work with. If this does not occur, the population size is automatically increased until this constraint is satisfied.

**max.generations**  
Maximum Generations. This is the maximum number of generations that genoud will run when attempting to optimize a function. This is a soft limit. The maximum generation limit will be binding for genoud only if hard.generation.limit has been set equal to TRUE. If it has not been set equal to TRUE, two soft triggers control when genoud stops: wait.generations and gradient.check.

Although the max.generations variable is not, by default, binding, it is nevertheless important because many operators use it to adjust their behavior. In essence, many of the operators become less random as the generation count gets
closer to the max.generations limit. If the limit is hit and genoud decides to continue working, genoud automatically increases the max.generation limit.

Please see MemoryMatrix for some important interactions with memory management.

wait.generations
If there is no improvement in the objective function in this number of generations, genoud will think that it has found the optimum. If the gradient.check trigger has been turned on, genoud will only start counting wait.generations if the gradients are within solution.tolerance of zero. The other variables controlling termination are max.generations and hard.generation.limit.

hard.generation.limit
This logical variable determines if the max.generations variable is a binding constraint for genoud. If hard.generation.limit is FALSE, then genoud may exceed the max.generations count if either the objective function has improved within a given number of generations (determined by wait.generations) or if the gradients are not zero (determined by gradient.check).

Please see MemoryMatrix for some important interactions with memory management.

starting.values
A vector or matrix containing parameter values which genoud will use at startup. Using this option, the user may insert one or more individuals into the starting population. If a matrix is provided, the columns should be the variables and the rows the individuals. genoud will randomly create the other individuals.

MemoryMatrix
This variable controls if genoud sets up a memory matrix. Such a matrix ensures that genoud will request the fitness evaluation of a given set of parameters only once. The variable may be TRUE or FALSE. If it is FALSE, genoud will be aggressive in conserving memory. The most significant negative implication of this variable being set to FALSE is that genoud will no longer maintain a memory matrix of all evaluated individuals. Therefore, genoud may request evaluations which it has already previously requested.

Note that when nvars or pop.size are large, the memory matrix consumes a large amount of RAM. Genoud’s memory matrix will require somewhat less memory if the user sets hard.generation.limit equal to TRUE.

Domains
This is a nvars ×2 matrix. For each variable, in the first column is the lower bound and in the second column the upper bound. None of genoud’s starting population will be generated outside of the bounds. But some of the operators may generate children which will be outside of the bounds unless the boundary.enforcement flag is turned on.

If the user does not provide any values for Domains, genoud will setup default domains using default.domains.

For linear and nonlinear constraints please see the discussion in the Note section.
default.domains

If the user does not want to provide a Domains matrix, domains may nevertheless be set by the user with this easy to use scalar option. Genoud will create a Domains matrix by setting the lower bound for all of the parameters equal to \(-1 \times \text{default.domains}\) and the upper bound equal to default.domains.

solution.tolerance

This is the tolerance level used by genoud. Numbers within solution.tolerance are considered to be equal. This is particularly important when it comes to evaluating wait.generations and conducting the gradient.check.

gr

A function to provide the gradient for the BFGS optimizer. If it is NULL, numerical gradients will be used instead.

boundary.enforcement

This variable determines the degree to which genoud obeys the boundary constraints. Notwithstanding the value of the variable, none of genoud’s starting population values will be outside of the bounds.

boundary.enforcement has three possible values: 0 (anything goes), 1 (partial), and 2 (no trespassing):

- **0: Anything Goes** This option allows any of the operators to create out-of-bounds individuals and these individuals will be included in the population if their fit values are good enough. The boundaries are only important when generating random individuals.

- **1: Partial enforcement** This allows operators (particularly those operators which use the derivative based optimizer, BFGS) to go out-of-bounds during the creation of an individual (i.e., out-of-bounds values will often be evaluated). But when the operator has decided on an individual, it must be in bounds to be acceptable.

- **2: No Trespassing** No out-of-bounds evaluations will ever be requested. In this case, boundary enforcement is also applied to the BFGS algorithm, which prevents candidates from straying beyond the bounds defined by Domains. Note that this forces the use of the L-BFGS-B algorithm for optim. This algorithm requires that all fit values and gradients be defined and finite for all function evaluations. If this causes an error, it is suggested that the BFGS algorithm be used instead by setting boundary.enforcement=1.

lexical

This option enables lexical optimization. This is where there are multiple fit criteria and the parameters are chosen so as to maximize fitness values in lexical order—i.e., the second fit criterion is only relevant if the parameters have the same fit for the first etc. The fit function used with this option should return a numeric vector of fitness values in lexical order. This option can take on the values of FALSE, TRUE or an integer equal to the number of fit criteria which are returned by fn. The value object which is returned by genoud will include all of the fit criteria at the solution. The GenMatch function makes extensive use of this option.

gradient.check

If this variable is TRUE, genoud will not start counting wait.generations unless each gradient is solution.tolerance close to zero. This variable has no effect if the max.generations limit has been hit and the hard.generation.limit
option has been set to TRUE. If BFGSburnin < 0, then it will be ignored unless gradient.check = TRUE.

BFGS

This variable denotes whether or not genoud applies a quasi-Newton derivative optimizer (BFGS) to the best individual at the end of each generation after the initial one. See the optim.method option to change the optimizer. Setting BFGS to FALSE does not mean that the BFGS will never be used. In particular, if you want BFGS never to be used, P9 (the Local-Minimum Crossover operator) must also be set to zero.

data.type.int

This option sets the data type of the parameters of the function to be optimized. If the variable is TRUE, genoud will search over integers when it optimizes the parameters.

With integer parameters, genoud never uses derivative information. This implies that the BFGS quasi-Newton optimizer is never used—i.e., the BFGS flag is set to FALSE. It also implies that Operator 9 (Local-Minimum Crossover) is set to zero and that gradient checking (as a convergence criterion) is turned off. No matter what other options have been set to, data.type.int takes precedence—i.e., if genoud is told that it is searching over an integer parameter space, gradient information is never considered.

There is no option to mix integer and floating point parameters. If one wants to mix the two, it is suggested that the user pick integer type and in the objective function map a particular integer range into a floating point number range. For example, tell genoud to search from 0 to 100 and divide by 100 to obtain a search grid of 0 to 1.0 (by .1).

Alternatively, the user could use floating point numbers and round the appropriate parameters to the nearest integer inside fn before the criterion (or criteria if lexical = TRUE) is evaluated. In that case, the transform option can be used to create the next generation from the current generation when the appropriate parameters are in the rounded state.

hessian

When this flag is set to TRUE, genoud will return the hessian matrix at the solution as part of its return list. A user can use this matrix to calculate standard errors.

unif.seed

This should not be set. To set the seed, one should use the set.seed() function directly. It only exists to ensure compatibility with functions which are used to call the older version of this function.

int.seed

See unif.seed.

print.level

This variable controls the level of printing that genoud does. There are four possible levels: 0 (minimal printing), 1 (normal), 2 (detailed), and 3 (debug). If level 2 is selected, genoud will print details about the population at each generation. The print.level variable also significantly affects how much detail is placed in the project file—see project.path. Note that R convention would have us at print level 0 (minimal printing). However, because genoud runs may take a long time, it is important for the user to receive feedback. Hence, print level 2 has been set as the default.
share.type

If share.type is equal to 1, then genoud, at startup, checks to see if there is an existing project file (see project.path). If such a file exists, it initializes its original population using it. This option can be used neither with the lexical nor the transform options.

If the project file contains a smaller population than the current genoud run, genoud will randomly create the necessary individuals. If the project file contains a larger population than the current genoud run, genoud will kill the necessary individuals using exponential selection.

If the number of variables (see nvars) reported in the project file is different from the current genoud run, genoud does not use the project file (regardless of the value of share.type) and genoud generates the necessary starting population at random.

instance.number

This number (starting from 0) denotes the number of recursive instances of genoud. genoud then sets up its random number generators and other such structures so that the multiple instances do not interfere with each other. It is up to the user to make certain that the different instances of genoud are not writing to the same output file(s): see project.path.

For the R version of genoud this variable is of limited use. It is basically there in case a genoud run is being used to optimize the result of another genoud run (i.e., a recursive implementation).

output.path

This option is no longer supported. It used to allow one to redirect the output. Now please use sink. The option remains in order to provide backward compatibility for the API.

output.append

This option is no longer supported. Please see sink. The option remains in order to provide backward compatibility for the API.

project.path

This is the path of the genoud project file. The project file prints one individual per line with the fit value(s) printed first and then the parameter values. By default genoud places its output in a file called "genoud.pro" located in the temporary directory provided by tempdir. The behavior of the project file depends on the print.level chosen. If the print.level variable is set to 1, then the project file is rewritten after each generation. Therefore, only the currently fully completed generation is included in the file. If the print.level variable is set to 2, then each new generation is simply appended to the project file. For print.level=0, the project file is not created.

P1

This is the cloning operator. genoud always clones the best individual each generation. But this operator clones others as well. Please see the Operators Section for details about operators and how they are weighted.

P2

This is the uniform mutation operator. One parameter of the parent is mutated. Please see the Operators Section for details about operators and how they are weighted.

P3

This is the boundary mutation operator. This operator finds a parent and mutates one of its parameters towards the boundary. Please see the Operators Section for details about operators and how they are weighted.
Non-Uniform Mutation. Please see the Operators Section for details about operators and how they are weighted.

This is the polytope crossover. Please see the Operators Section for details about operators and how they are weighted.

Simple Crossover. Please see the Operators Section for details about operators and how they are weighted.

Whole Non-Uniform Mutation. Please see the Operators Section for details about operators and how they are weighted.

Heuristic Crossover. Please see the Operators Section for details about operators and how they are weighted.

Local-Minimum Crossover: BFGS. This is rather CPU intensive, and should be generally used less than the other operators. Please see the Operators Section for details about operators and how they are weighted.

This is a tuning parameter for the P9 operator. The local-minimum crossover operator by default takes the convex combination of the result of a BFGS optimization and the parent individual. By default the mixing (weight) parameter for the convex combination is chosen by a uniform random draw between 0 and 1. The P9mix option allows the user to select this mixing parameter. It may be any number greater than 0 and less than or equal to 1. If 1, then the BFGS result is simply used.

The number of generations which are run before the BFGS is first used. Premature use of the BFGS can lead to convergence to a local optimum instead of the global one. This option allows the user to control how many generations are run before the BFGS is started and would logically be a non-negative integer. However, if BFGSburnin < 0, the BFGS will be used if and when wait.generation is doubled because at least one gradient is too large, which can only occur when gradient.check = TRUE. This option delays the use of both the BFGS on the best individual and the P9 operator.

This is a function for the BFGS optimizer to optimize, if one wants to make it distinct from the fn function. This is useful when doing lexical optimization because otherwise a derivative based optimizer cannot be used (since it requires a single fit value). It is suggested that if this functionality is being used, both the fn and BFGSfn functions obtain all of the arguments they need (except for the parameters being optimized) by lexical scope instead of being passed in as arguments to the functions. Alternatively, one may use the BFGShelp option to pass arguments to BFGSfn. If print.level > 2, the results from the BFGS optimizer are printed every time it is called.

An optional function to pass arguments to BFGSfn. This function should take an argument named ‘initial’, an argument named ‘done’ that defaults to FALSE, or at least allow ... to be an argument. BFGSfn must have an argument named ‘helper’ if BFGShelp is used because the call to optim includes the hardcoded expression helper = do.call(BFGShelp, args = list(initial = foo.vals), envir = environment(fn)), which evaluates the BFGShelp function in the environment of BFGSfn (fn is just a wrapper for BFGSfn) at par = foo.vals where foo.vals contains the starting values for the BFGS algorithm. The ‘done’ argument to BFGSfn is used if the user requests that the Hessian be calculated at the genoud solution.
control A list of control parameters that is passed to `optim` if `BFGS = TRUE` or `P9 > 0`. Please see the `optim` documentation for details.

optim.method A character string among those that are admissible for the method argument to the `optim` function, namely one of "BFGS", "L-BFGS-B", "Nelder-Mead", "CG", or "SANN". By default, `optim.method` is "BFGS" if `boundary.enforcement < 2` and is "L-BFGS-B" if `boundary.enforcement = 2`. For discontinuous objective functions, it may be advisable to select "Nelder-Mead" or "SANN". If selecting "L-BFGS-B" causes an error message, it may be advisable to select another method or to adjust the control argument. Note that the various arguments of `genoud` that involve the four letters "BFGS" continue to be passed to `optim` even if `optim.method` != "BFGS".

transform A logical that defaults to `FALSE`. If `TRUE`, it signifies that `fn` will return a numeric vector that contains the fit criterion (or fit criteria if `lexical = TRUE`), followed by the parameters. If this option is used, `fn` should have the following general form in its body:

```r
par <- myTransformation(par)
criter <- myObjective(par)
return( c(criter, par) )
```

This option is useful when parameter transformations are necessary because the next generation of the population will be created from the current generation in the transformed state, rather than the original state. This option can be used by users to implement their own operators.

There are some issues that should be kept in mind. This option cannot be used when `data.type.int = TRUE`. Also, this option coerces `MemoryMatrix` to be `FALSE`, implying that the cluster option cannot be used. And, unless `BFGSfn` is specified, this option coerces `gradient.check` to `FALSE`, `BFGS` to `FALSE`, and `P9` to `0`. If `BFGSfn` is specified, that function should perform the transformation but should only return a scalar fit criterion, for example:

```r
par <- myTransformation(par)
criter <- myCriterion(par)
return(criter)
```

Finally, if `boundary.enforcement > 0`, care must be taken to assure that the transformed parameters are within the `domains`, otherwise unpredictable results could occur. In this case, the transformations are checked for consistency with `domains` but only in the initial generation (to avoid an unacceptable loss in computational speed).

debug This variable turns on some debugging information. This variable may be `TRUE` or `FALSE`.

cluster This can either be an object of the cluster class returned by one of the `makeCluster` commands in the `parallel` package or a vector of machine names so `genoud` can setup the cluster automatically. If it is the latter, the vector should look like:

```r
c("localhost","musil","musil","deckard")
```

This vector would create a cluster with four nodes: one on the localhost another on "deckard" and two on the machine named "musil". Two nodes on a given machine make sense if the machine has two or more chips/cores.
will setup a SOCK cluster by a call to `makePSOCKcluster`. This will require
the user to type in her password for each node as the cluster is by default
created via ssh. One can add on usernames to the machine name if it differs
from the current shell: "username@musil". Other cluster types, such as PVM
and MPI, which do not require passwords can be created by directly calling
`makeCluster`, and then passing the returned cluster object to `genoud`. For an
example of how to manually setup up a cluster with a direct call to `makeCluster`
For an example of how to get around a firewall by ssh tunneling see:

**balance**

This logical flag controls if load balancing is done across the cluster. Load
balancing can result in better cluster utilization; however, increased communi-
cation can reduce performance. This option is best used if the function being
optimized takes at least several minutes to calculate or if the nodes in the cluster
vary significantly in their performance. If `cluster==FALSE`, this option has no
effect.

Further arguments to be passed to `fn` and `gr`.

**Details**

`genoud` solves problems that are nonlinear or perhaps even discontinuous in the parameters of
the function to be optimized. When a statistical model’s estimating function (for example, a log-
likelihood) is nonlinear in the model’s parameters, the function to be optimized will generally not be
globally concave and may have irregularities such as saddlepoints or discontinuities. Optimization
methods that rely on derivatives of the objective function may be unable to find any optimum at all.
Multiple local optima may exist, so that there is no guarantee that a derivative-based method will
converge to the global optimum. On the other hand, algorithms that do not use derivative information
(such as pure genetic algorithms) are for many problems needlessly poor at local hill climbing.
Most statistical problems are regular in a neighborhood of the solution. Therefore, for some portion
of the search space, derivative information is useful for such problems. `genoud` also works well for
problems that no derivative information exists. For additional documentation and examples please
see [http://sekhon.berkeley.edu/rgenoud](http://sekhon.berkeley.edu/rgenoud).

**Value**

`genoud` returns a list with 7 objects. 8 objects are returned if the user has requested the hessian to
be calculated at the solution. Please see the `hessian` option. The returned objects are:

- **value**: This variable contains the fitness value at the solution. If lexical optimization
  was requested, it is a vector.

- **par**: This vector contains the parameter values found at the solution.

- **gradients**: This vector contains the gradients found at the solution. If no gradients were
  calculated, they are reported to be NA.

- **generations**: This variable contains the number of generations `genoud` ran for.

- **peakgeneration**: This variable contains the generation number at which `genoud` found the solu-
  tion.

- **hessian**: This object contains the hessian matrix at the solution. It is an (n x n) matrix where n
  is the number of parameters.

- **call**: This object contains the call to `genoud` that produced the result.

- **parameters**: This object contains the parameters used in the `genoud` function call.

- **evaluations**: This object contains the number of function evaluations performed.

- **fitness**: This object contains the fitness value at each evaluation.

- **gradients**: This object contains the gradients at each evaluation.

- **par**: This object contains the parameter values at each evaluation.

- **call**: This object contains the call to `genoud` that produced the result.

- **parameters**: This object contains the parameters used in the `genoud` function call.

- **evaluations**: This object contains the number of function evaluations performed.

- **fitness**: This object contains the fitness value at each evaluation.

- **gradients**: This object contains the gradients at each evaluation.

- **par**: This object contains the parameter values at each evaluation.

- **call**: This object contains the call to `genoud` that produced the result.

- **parameters**: This object contains the parameters used in the `genoud` function call.

- **evaluations**: This object contains the number of function evaluations performed.

- **fitness**: This object contains the fitness value at each evaluation.

- **gradients**: This object contains the gradients at each evaluation.

- **par**: This object contains the parameter values at each evaluation.
pop.size This variable contains the population size that genoud actually used. See pop.size for why this value may differ from the population size the user requested.

operators This vector reports the actual number of operators (of each type) genoud used. Please see the Operators Section for details.

hessian If the user has requested the hessian matrix to be returned (via the hessian flag), the hessian at the solution will be returned. The user may use this matrix to calculate standard errors.

**Operators**

Genoud has nine operators that it uses. The integer values which are assigned to each of these operators (P1 · · · P9) are weights. Genoud calculates the sum of \( s = P1 + P2 + · · · + P9 \). Each operator is assigned a weight equal to \( W_n = \frac{s}{P_n} \). The number of times an operator is called usually equals \( c_n = W_n \times \text{pop.size} \).

Operators 6 (Simple Crossover) and 8 (Heuristic Crossover) require an even number of individuals to work on—i.e., they require two parents. Therefore, the pop.size variable and the operators sets must be such that these three operators have an even number of individuals to work with. If this does not occur, genoud automatically upwardly adjusts the population size to make this constraint hold.

Strong uniqueness checks have been built into the operators to help ensure that the operators produce offspring different from their parents, but this does not always happen.

Note that genoud always keeps the best individual each generation.

Genoud’s 9 operators are:

1. Cloning
2. Uniform Mutation
3. Boundary Mutation
4. Non-Uniform Crossover
5. Polytope Crossover
6. Simple Crossover
7. Whole Non-Uniform Mutation
8. Heuristic Crossover
9. Local-Minimum Crossover: BFGS

For more information please see Table 1 of the reference article: [http://sekhon.berkeley.edu/papers/rgenoudJSS.pdf](http://sekhon.berkeley.edu/papers/rgenoudJSS.pdf).
Note

The most important options affecting performance are those determining population size (`pop.size`) and the number of generations the algorithm runs (`max.generations`, `wait.generations`, `hard.generation.limit` and `gradient.check`). Search performance is expected to improve as the population size and the number of generations the program runs for increase. These and the other options should be adjusted for the problem at hand. Please pay particular attention to the search domains (`Domains` and `default.domains`). For more information please see the reference article.

Linear and nonlinear constraints among the parameters can be introduced by users in their fit function. For example, if the sum of parameters 1 and 2 must be less than 725, the following can be placed in the fit function the user is going to have `genoud` maximize:

```
if ((parm1 + parm2) > 725) { return(-99999999) }
```

In this example, a very bad fit value is returned to `genoud` if the linear constrain is violated. `genoud` will then attempt to find parameter values that satisfy the constraint.

Alternatively, one can use lexical optimization where the first criterion is a binary variable that equals 1.0 iff \((\text{parm1} + \text{parm2}) < 725\) and the second criterion is the fit function, which should also be passed to `BFGSfn`. All candidates where \((\text{parm1} + \text{parm2}) \geq 725\) will be ranked below all candidates where \((\text{parm1} + \text{parm2}) < 725\) and within these two groups, candidates will be ranked by their fit on the second criterion. The optimal candidate is thus the one with the best fit on the second criterion among candidates that satisfy this restriction.

Author(s)

Walter R. Mebane, Jr., University of Michigan, <wmebane@umich.edu>, http://www-personal.umich.edu/~wmebane/

Jasjeet S. Sekhon, UC Berkeley, <sekhon@berkeley.edu>, http://sekhon.berkeley.edu/

References


See Also

optim.

Examples

#maximize the sin function
sin1 <- genoud(sin, nvars=1, max=TRUE)

#minimize the sin function
sin2 <- genoud(sin, nvars=1, max=FALSE)

## Not run:
#maximize a univariate normal mixture which looks like a claw
claw <- function(xx) {
  x <- xx[1]
  y <- (0.46*(dnorm(x,-1.0,2.0/3.0) + dnorm(x,1.0,2.0/3.0)) +
       (1.0/30.0)*(dnorm(x,-0.5,.01) + dnorm(x,-1.0,.01) + dnorm(x,-1.5,.01)) +
       (7.0/30.0)*(dnorm(x,0.5,.07) + dnorm(x,1.0,.07) + dnorm(x,1.5,.07)))
  return(y)
}
claw1 <- genoud(claw, nvars=1, pop.size=3000, max=TRUE)

## End(Not run)

## Not run:
#Plot the previous run
xx <- seq(-3,3,.05)
plot(xx,lapply(xx,claw),type="l",xlab="Parameter",ylab="Fit",
     main="GENoud: Maximize the Claw Density")
points(claw1$par,claw1$value,col="red")

# Maximize a bivariate normal mixture which looks like a claw.
biclaw <- function(xx) {
  mNd2 <- function(x1, x2, mu1, mu2, sigma1, sigma2, rho)
  {
    z1 <- (x1-mu1)/sigma1
    z2 <- (x2-mu2)/sigma2
    w <- (1.0/(2.0*pi*sigma1*sigma2*sqrt(1-rho*rho)))
    w <- w*exp(-0.5*(z1*z1 - 2*rho*z1*z2 + z2*z2)/(1-rho*rho))
    return(w)
  }
  x1 <- xx[1]+1
  x2 <- xx[2]+1
  y <- (0.5*mNd2(x1,x2,0.0,0.0,0.0,1.0,0.0,0.0) +
       0.1*(mNd2(x1,x2,-1.0,-1.0,0.0,1.0,0.0,0.0) +
            mNd2(x1,x2,0.0,0.0,0.0,1.0,0.1,0.0) +
            mNd2(x1,x2,0.5,0.5,0.1,0.1,0.0,0.0) +
            mNd2(x1,x2,1.0,1.0,0.0,1.0,0.1,0.0)))
  return(y)
biclaw1 <- genoud(biclaw, default.domains=20, nvars=2, pop.size=5000, max=TRUE)

## End(Not run)
# For more examples see: http://sekhon.berkeley.edu/rgenoud/R.
Index

*Topic **nonlinear**
  genoud, 1
*Topic **optimize**
  genoud, 1

GenMatch, 4
  genoud, 1

makeCluster, 8, 9
makePSOCKcluster, 9

optim, 4, 7, 8, 12

sink, 6

tempdir, 6