The rgenoud Package

November 5, 2006

Version 4.2-0

Date 2006-11-02

Title R version of GENetic Optimization Using Derivatives

Author Walter R. Mebane, Jr. <wrm1@cornell.edu>, Jasjeet Singh Sekhon <sekhon@berkeley.edu>

Maintainer Jasjeet Singh Sekhon <sekhon@berkeley.edu>

Depends R (>= 1.7.1)

Suggests snow (>= 0.2-1)

Description A genetic algorithm plus derivative optimizer

License GPL version 2 or newer

URL http://sekhon.berkeley.edu/rgenoud/

R topics documented:

genoud 1

Index **12**

`genoud` *GENetic Optimization Using Derivatives*

Description

`Genoud` is a function that combines evolutionary algorithm methods with a derivative-based, quasi-Newton method to solve difficult unconstrained optimization problems. `Genoud` is made to solve 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 non-linear in the model's parameters, the function to be optimized will usually not be globally concave and may contain irregularities such as saddlepoints or discontinuous jumps. 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 GAs) are for many problems needlessly poor at local hill climbing. Most statistical problems are regular in the neighborhood of the solution. Therefore, for some portion of the search space derivative information is useful. `Genoud`, via the `cluster` option, supports the use of multiple computers, CPUs or cores to perform parallel computations.

Usage

```
genoud(fn, nvars, max=FALSE, pop.size=1000, max.generations=100, wait.generations=1
               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,
               data.type.int=FALSE, hessian=FALSE, unif.seed=812821, int.seed=53058,
               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,
               cluster=FALSE, balance=FALSE, debug=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.

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

`nvars` This is the number of variables the function to be minimized (or maximized) takes.

`max` Maximization (*TRUE*) or Minimizing (*FALSE*). This variable tells `genoud` if it is to minimize or maximize 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 (Multiple Point 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
: This vector contains the starting values which genoud will use at startup. The starting.values vector is a way for the user to insert *one* individual into the starting population. 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 $\times 2$ matrix. The first column is the lower bound, and the second column is 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 × 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 return the gradient for the BFGS optimizer. It 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 will be 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.

  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.

lexical
: This option enables lexical optimization. This is where there are multiple fit criterion and the parameters are chosen so as to maximize them in lexical order—i.e., the second fit criterion is only considered if the parameters has the same fit for the first. The fit function used with this option should return a vector of fits in the order of precedence. This option can take on the values of FALSE, TRUE or an integer equal to the number of fit criterion which are returned by fn. The par object which is returned by genoud will include all of the fit criterion 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*.

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. Setting BFGS to *FALSE* does not mean that the BFGS will never be used. In particular, Operator 9 (Local-Minimum Crossover) 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` is informed that it is dealing with integer values. *Use of the integer data type is supported only as a beta feature.* Some of the included operators will not work well with integer type 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 her 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).

`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 sets the seed for the floating-point random number generator which `genoud` uses. The default value of this seed is 81282. `genoud` uses its own internal random number generator (a Tausworthe-Lewis-Payne) to allow for recursive and parallel calls to `genoud`. It does not use the seed set by the `set.seed` function.

`int.seed` This sets the seed for the integer random number generator which `genoud` uses. The default value of this seed is 53058. `genoud` uses its own internal random number generator (a Tausworthe-Lewis-Payne) to allow for recursive and parallel calls to `genoud`. It does not use the seed set by the `set.seed` function.

`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.

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 output.path and 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 is the full (relative) path to where genoud's output is to go. If the value of output.path= "stdout", then genoud's output will go to standard output in UNIX and to the GUI console in Windows. Also see output.append and project.path.

output.append

If output is being sent to a file (see output.path), this logical variable tells genoud whether it should append to the file if it already exists or if it should overwrite an existing file.

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 it's 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 all other values of print.level, 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.

P4 Non-Uniform Mutation. Please see the Operators Section for details about operators and how they are weighted.

P5 This is the polytope crossover. Please see the Operators Section for details about operators and how they are weighted.

P6 Multiple Point Simple Crossover. Please see the Operators Section for details about operators and how they are weighted.

P7 Whole Non-Uniform Mutation. Please see the Operators Section for details about operators and how they are weighted.

P8 Heuristic Crossover. Please see the Operators Section for details about operators and how they are weighted.

P9 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.

cluster This can either be an object of the 'cluster' class returned by one of the makeCluster commands in the snow package or a vector of machine names so genoud can setup the cluster automatically. If it is the later, the vector should look like: 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. genoud will setup a SOCK cluster by a call to makeSOCKcluster. 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 see http://sekhon.berkeley.edu/rgenoud/R/genoud_cluster_manual.R. For an example of how to get around a firewall by ssh tunneling see: http://sekhon.berkeley.edu/rgenoud/R/genoud_cluster_manual_tunnel.R.

balance This logical flag controls if load balancing is done across the cluster. Load balancing can result in better cluster utilization; however, increased communication can reduce performance. This options 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.

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

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

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 request, 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 variables contains the number of generations `genoud` ran for.

`peakgeneration` This variable contains the generation number at which `genoud` found the solution.

`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$\cdots$P9) are weights. `Genoud` calculates the sum of $s = P1 + P2 + \cdots + P9$. Each operator is assigned a weight equal to $W_n = \frac{s}{P_n}$. The number of times an operator is called usually equals $c_n = W_n \times pop.size$.

Operators 6 (Multiple Point 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. Multiple Point 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/genoud/node7.shtml.

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 increase. These and the other options should be adjusted for the problem at hand. Please pay particular attentions 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.

Author(s)

Walter R. Mebane, Jr., Cornell University, ⟨wrm1@cornell.edu⟩, http://macht.arts.cornell.edu/wrm1/

Jasjeet S. Sekhon, UC Berkeley, ⟨sekhon@berkeley.edu⟩, http://sekhon.berkeley.edu/

References

Sekhon, Jasjeet Singh and Walter R. Mebane, Jr. 1998. "Genetic Optimization Using Derivatives: Theory and Application to Nonlinear Models." *Political Analysis*, 7: 187-210. http://sekhon.berkeley.edu/genoud/genoud.pdf

Mebane, Walter R., Jr. and Jasjeet S. Sekhon. 2004. "Robust Estimation and Outlier Detection for Overdispersed Multinomial Models of Count Data." *American Journal of Political Science*, 48 (April): 391-410. http://sekhon.berkeley.edu/papers/MebaneSekhon.multinom.pdf

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);

#maximize a univariate normal mixture which looks like a claw
claw <- function(xx) {
  Nd <- function(x, mu, sigma) {
    w <- (1.0/sqrt(2.0*pi*sigma*sigma)) ;
    z <- (x-mu)/sigma;
    w <- w*exp(-0.5*z*z) ;
    as.double(w);
  }
  x <- xx[1];
  y <- (0.46*(Nd(x,-1.0,2.0/3.0) + Nd(x,1.0,2.0/3.0)) +
           (1.0/300.0)*(Nd(x,-0.5,.01) + Nd(x,-1.0,.01) + Nd(x,-1.5,.01)) +
           (7.0/300.0)*(Nd(x,0.5,.07) + Nd(x,1.0,.07) + Nd(x,1.5,.07))) ;
  as.double(y);
}
 claw1   <- genoud(claw, nvars=1,P9=100,max=TRUE);

## 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
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)) ;
      as.double(w);
    }
  x1 <- xx[1]+1;
  x2 <- xx[2]+1;

  y <- (0.5*mNd2(x1,x2,0.0,0.0,1.0,1.0,0.0) +
            0.1*(mNd2(x1,x2,-1.0,-1.0,0.1,0.1,0.0) +
                 mNd2(x1,x2,-0.5,-0.5,0.1,0.1,0.0) +
                 mNd2(x1,x2,0.0,0.0,0.1,0.1,0.0) +
                 mNd2(x1,x2,0.5,0.5,0.1,0.1,0.0) +
                 mNd2(x1,x2,1.0,1.0,0.1,0.1,0.0)));

  as.double(y);
}
biclaw1 <- genoud(biclaw, default.domains=20, nvars=2,P9=100,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, *7*
makeSOCKcluster, *7*

optim, *9*

set.seed, *5*

tempdir, *6*