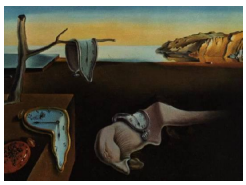

Journal of Statistical Software
June 2011, Volume 42, Issue 11. http://www.jstatsoft.org/

Genetic Optimization Using Derivatives: The rgenoud Package for R

Walter R. Mebane, Jr.
University of Michigan

Jasjeet S. Sekhon
UC Berkeley

Abstract

genoud is an R function that combines evolutionary algorithm methods with a derivative-based (quasi-Newton) method to solve difficult optimization problems. genoud may also be used for optimization problems for which derivatives do not exist. genoud solves problems that are nonlinear or perhaps even discontinuous in the parameters of the function to be optimized. When the function to be optimized (for example, a log-likelihood) is nonlinear in the model's parameters, the function 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. The function supports parallel processing on multiple CPUs on a single machine or a cluster of computers.

Keywords: genetic algorithm, evolutionary program, optimization, parallel computing, R.

1. Introduction

We developed the R package **rgenoud** to solve difficult optimization problems such as often arise when estimating nonlinear statistical models or solving complicated nonlinear, nonsmooth and even discontinuous functions. The **rgenoud** software package is available from the Comprehensive R (R Development Core Team 2011) Archive Network at http://CRAN.R-project.org/package=rgenoud. Optimization difficulties often arise when the objective function (for instance, the log-likelihood) is a nonlinear function of the parameters. In such cases the function to be optimized is usually not globally concave. An objective func-

tion that is not globally concave may have multiple local optima, saddle points, boundary solutions or discontinuities. While the objective function for a statistical model is often concave in a neighborhood of the optimal solution, that neighborhood is often a small proportion of the parameter space of potential interest, and outside that neighborhood the function may be irregular. In such cases, methods of optimization that depend entirely on derivatives can be unreliable and often are virtually unusable. Newton-Raphson and quasi-Newton methods are among the commonly used optimization methods that rely completely on derivatives. Such methods work well when the function to be optimized is regular and smooth over the domain of parameter values that is of interest, but otherwise the methods often fail (Gill, Murray, and Wright 1981). Even in models where such methods can be expected to work most of the time, resampling techniques such as the bootstrap (Efron and Tibshirani 1994) can generate resamples in which derivative-based optimization algorithms encounter severe difficulties. This is unfortunate because the methods most frequently used for optimization in problems of statistical estimation are entirely based on derivatives.

The **rgenoud** package has been used by wide variety of users and developers. More than twelve R packages currently rely upon **rgenoud** including: **anchors** (analyzing survey data with anchoring vignettes, Wand, King, and Lau 2011, 2008; King and Wand 2007); **BARD** (automated redistricting, Altman and McDonald 2011); **boolean** (boolean binary response models, Braumoeller 2003; Braumoeller, Goodrich, and Kline 2010); **DiceOptim** (kriging-based optimization for computer experiments, Ginsbourger and Roustant 2010); **FAiR** (factor analysis, Goodrich 2011); **JOP** (simultaneous optimization of multiple responses, Kuhnt and Rudak 2011); **KrigInv** (kriging-based inversion, Picheny and Ginsbourger 2011); **PKfit** (data analysis in pharmacokinetics, Lee and Lee 2009a); **Matching** (propensity score and multivariate matching, Sekhon 2011b,a); **ivivc** (in vitro-in vivo correlation modeling, Lee and Lee 2009b); **multinomRob** (robust multinomial models, Mebane and Sekhon 2009, 2004); and **Synth** (synthetic control group method for comparative case studies, Abadie and Gardeazabal 2003; Diamond and Hainmueller 2008; Abadie, Diamond, and Hainmueller 2011).

We present in Section 3 an example using benchmark functions taken from Yao, Liu, and Lin (1999), followed by an example motivated by the `multinomRob` package. The benchmark suite includes functions that are high-dimensional, discontinuous or that have many local optima. The `multinomRob` package robustly estimates overdispersed multinomial regression models and uses **rgenoud** to solve a least quartile difference (LQD) generalized S estimator (Mebane and Sekhon 2004). The LQD is not a smooth function of the regression model parameters. The function is continuous, but the parameter space is finely partitioned by nondifferentiable boundaries.

In another paper in this volume, the **Matching** package and its use of **rgenoud** are described in detail (Sekhon 2011a). **Matching** provides functions for multivariate and propensity score matching and for finding optimal covariate balance based on a genetic search algorithm implemented in **rgenoud**. The search over optimal matches is discontinuous so no derivatives are used.[1] The search also involves lexical optimization which is a unique feature implemented in **rgenoud**.[2]

[1]The `BFGS` option of `genoud` is set to `FALSE`, and the 9th operator which depends on derivatives is not used.

[2]Lexical optimization is useful when there are multiple fitness criteria; the parameters are chosen so as to maximize fitness values in lexical order—i.e., the second fit criterion is only considered if the parameters have the same fit for the first. See the `lexical` option and Sekhon (2011a) for details. All of `genoud`'s options are described in the R help file for the function.

The **rgenoud** package implements an updated and extended version of the C program `GENOUD` (Mebane and Sekhon 1997) described in Sekhon and Mebane (1998). The many improvements include among other things the interface with R, which includes the ability to optimize functions written in R, options to optimize both floating point and integer-valued parameters, the ability to optimize loss functions which return multiple fitness values (lexical optimization), the ability to call `genoud` recursively, the ability to have the optimizer evaluate fits only for new parameter values, and the ability to use multiple computers, CPUs or cores to perform parallel computations.

The **rgenoud** program combines an evolutionary algorithm with a quasi-Newton method. The quasi-Newton method is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (Gill *et al.* 1981, p. 119) implemented in R's `optim` function. When the BFGS is being used, our program offers the option of using either **rgenoud**'s built-in numerical derivatives (which are based on code taken from Gill *et al.* 1981, p. 337–344) or user-supplied analytical derivatives.[3] Our program can also work without the BFGS, in which case no derivatives are needed and the optimizer will work even when the function is discontinuous. The primary benefit from using derivatives is that the algorithm will then quickly find a local optimum when a current set of trial solution parameter values is in a smooth neighborhood of the local optimum point. Appropriate use of the BFGS can make the algorithm converge to the global optimum much more quickly. But premature or excessive use of the BFGS can prevent convergence to the global optimum.[4] As always, it is hazardous to rely on an optimizer's default settings. Our program does not eliminate the need for judgment, testing and patience.

As Gill, Murray and Wright observe, "there is no guaranteed strategy that will resolve every difficulty" (Gill *et al.* 1981, p. 285). In this article, we very briefly review the theory of random search algorithms that supports the assertion that **rgenoud** has a high probability of finding the global optimum when such exists. And we present three examples of how to use the `genoud` function: to optimize a simple but fiendish scalar Normal mixture model; to minimize a suite of benchmark functions that have previously been used to test evolutionary programming optimization algorithms; and to optimize a version of the only intermittently differentiable LQD estimator. Additional details on both the theory and performance of `genoud` can be found in our article that describes `GENOUD` (Sekhon and Mebane 1998).

2. Background on genetic optimization

An evolutionary algorithm (EA) uses a collection of heuristic rules to modify a population of trial solutions in such a way that each generation of trial values tends to be, on average, better than its predecessor. The measure for whether one trial solution is better than another is the trial solution's fitness value. In statistical applications, the fitness is a function of the summary statistic being optimized (e.g., the log-likelihood). **rgenoud** works for cases in which a solution is a vector of floating-point or integer numbers that serve as the parameters of a function to be optimized. The search for a solution proceeds via a set of heuristic rules, or *operators*, each of which acts on one or more trial solutions from the current population to produce one or more trial solutions to be included in the new population. EAs do not require derivatives to exist or the function to be continuous in order find the global optimum.

[3] User supplied derivatives may be provides via the `gr` option.

[4] The user can control whether `genoud` uses the BFGS at all (via the `BFGS` option), and if operators that use the BFGS are used (via the `P9` option), how often they are used.

The EA in **rgenoud** is fundamentally a genetic algorithm (GA) in which the code-strings are vectors of numbers rather than bit strings, and the GA operators take special forms tuned for the floating-point or integer vector representation. A GA uses a set of randomized genetic operators to evolve a finite population of finite code-strings over a series of generations (Holland 1975; Goldberg 1989; Grefenstette and Baker 1989). The operators used in GA implementations vary (Davis 1991; Filho and Alippi 1994), but in an analytical sense the basic set of operators can be defined as reproduction, mutation, crossover and inversion. The variation in these operators across different GA implementations reflects the variety of codes best suited for different applications. Reproduction entails selecting a code-string with a probability that increases with the code-string's fitness value. Crossover and inversion use pairs or larger sets of the selected code-strings to create new code-strings. Mutation randomly changes the values of elements of a single selected code-string.

Used in suitable combinations, the genetic operators tend to improve average fitness of each successive generation, though there is no guarantee that average fitness will improve between every pair of successive generations. Average fitness may well decline. But theorems exist to prove that parts of the trial solutions that have above average fitness values in the current population are sampled at an exponential rate for inclusion in the subsequent population (Holland 1975, p. 139–140). Each generation's population contains a biased sample of code-strings, so that a substring's performance in that population is a biased estimate of its average performance over all possible populations (De Jong 1993; Grefenstette 1993).

The long-run properties of a GA may be understood by thinking of the GA as a Markov chain. A state of the chain is a code-string population of the size used in the GA. For code-strings of finite length and GA populations of finite size, the state space is finite. If such a GA uses random reproduction and random mutation, all states always have a positive probability of occurring. A finite GA with random reproduction and mutation is therefore approximately a finite and irreducible Markov chain.[5] An irreducible, finite Markov chain converges at an exponential rate to a unique stationary distribution (Billingsley 1986, p. 128). This means that the probability that each population occurs rapidly converges to a constant, positive value. Nix and Vose (1992) and Vose (1993) use a Markov chain model to show that in a GA where the probability that each code-string is selected to reproduce is proportional to its observed fitness, the stationary distribution strongly emphasizes populations that contain code-strings that have high fitness values. They show that asymptotic in the population size—i.e., in the limit for a series of GAs with successively larger populations—populations that have suboptimal average fitness have probabilities approaching zero in the stationary distribution, while the probability for the population that has optimal average fitness approaches one. If $k > 1$ populations have optimal average fitness, then in the limiting stationary distribution the probability for each approaches $1/k$.

The theoretical results of Nix and Vose imply that a GA's success as an optimizer depends on having a sufficiently large population of code-strings. If the GA population is not sufficiently large, then the Markov chain that the GA approximately implements is converging to a stationary distribution in which the probabilities of optimal and suboptimal states are not

[5] Feller (1970, p. 372–419) and Billingsley (1986, p. 107–142) review the relevant properties of Markov chains. The randomness in an actual GA depends on the performance of pseudorandom number generators. This and the limitations of floating point arithmetic mean it is not literally true that an actual GA has a positive probability of reaching any state from any other state, and some states may in fact not be reachable from a given set of initial conditions.

P1 Cloning. Copy $\mathbf{X}_t$ into the next generation, $\mathbf{X}_{t+1}$.

P2 Uniform mutation. At random choose $i \in \mathbf{N}$. Select a value $\tilde{x}_i \sim U(\underline{x}_i, \overline{x}_i)$. Set $X_i = \tilde{x}_i$.

P3 Boundary mutation. At random choose $i \in \mathbf{N}$. Set either $X_i = \underline{x}_i$ or $X_i = \overline{x}_i$, with probability 1/2 of using each value.

P4 Non-uniform mutation. At random choose $i \in \mathbf{N}$. Compute $p = (1 - t/T)^B u$, where t is the current generation number, T is the maximum number of generations, $B > 0$ is a tuning parameter and $u \sim U(0, 1)$. Set either $X_i = (1-p)x_i + p\underline{x}_i$ or $X_i = (1-p)x_i + p\overline{x}_i$, with probability 1/2 of using each value.

P5 Polytope crossover. Using $m = \max(2, n)$ vectors $\mathbf{x}$ from the current population and m random numbers $p_j \in (0, 1)$ such that $\sum_{j=1}^{m} p_j = 1$, set $\mathbf{X} = \sum_{j=1}^{m} p_j \mathbf{x}_j$.

P6 Simple crossover. Choose a single integer i from $\mathbf{N}$. Using two parameter vectors, $\mathbf{x}$ and $\mathbf{y}$, set $X_i = px_i + (1 - p)y_i$ and $Y_i = py_i + (1 - p)x_i$, where $p \in (0, 1)$ is a fixed number.

P7 Whole non-uniform mutation. Do non-uniform mutation for all the elements of $\mathbf{X}$.

P8 Heuristic crossover. Choose $p \sim U(0, 1)$. Using two parameter vectors, $\mathbf{x}$ and $\mathbf{y}$, compute $\mathbf{z} = p(\mathbf{x} - \mathbf{y}) + \mathbf{x}$. If $\mathbf{z}$ satisfies all constraints, use it. Otherwise choose another p value and repeat. Set $\mathbf{z}$ equal to the better of $\mathbf{x}$ and $\mathbf{y}$ if a satisfactory mixed $\mathbf{z}$ is not found by a preset number of attempts. In this fashion produce two $\mathbf{z}$ vectors.

P9 Local-minimum crossover. Choose $p \sim U(0, 1)$. Starting with $\mathbf{x}$, run BFGS optimization up to a preset number of iterations to produce $\tilde{\mathbf{x}}$. Compute $\mathbf{z} = p\tilde{\mathbf{x}} + (1-p)\mathbf{x}$. If $\mathbf{z}$ satisfies boundary constraints, use it. Otherwise shrink p by setting $p = p/2$ and recompute $\mathbf{z}$. If a satisfactory $\mathbf{z}$ is not found by a preset number of attempts, return $\mathbf{x}$. This operators is extremely computationally intensive, use sparingly.

Table 1: `genoud` operators; adapted from Sekhon and Mebane (1998). Notation: $\mathbf{X} = [X_1, \ldots, X_n]$ is the vector of n parameters X_i. $\underline{x}_i$ is the lower bound and $\overline{x}_i$ is the upper bound on values for X_i. x_i is the current value of X_i, and $\mathbf{x}$ is the current value of $\mathbf{X}$. $\mathbf{N} = \{1, \ldots, n\}$. $p \sim U(0, 1)$ means that p is drawn from the uniform distribution on the $[0, 1]$ interval.

sharply distinguished. Suboptimal populations can be as likely to occur as optimal ones. If the stationary distribution is not favorable, the run time in terms of generations needed to produce an optimal code-string will be excessive. For all but trivially small state spaces, an unfavorable stationary distribution can easily imply an expected running time in the millions of generations. But if the stationary distribution strongly emphasizes optimal populations, relatively few generations may be needed to find an optimal code-string. In general, the probability of producing an optimum in a fixed number of generations tends to increase with the GA population size.

The evolutionary algorithm in **rgenoud** uses nine operators that are listed in Table 1. The operators extend and modify a set of operators used in `GENOCOP` (Michalewicz, Swaminathan, and Logan 1993; Michalewicz 1992). The operators are numbered using syntax matching that used to refer to them by **rgenoud**. The *cloning* operator simply makes copies of the best trial solution in the current generation (independent of this operator, **rgenoud** always retains one instance of the best trial solution). The *uniform mutation*, *boundary mutation* and *non-uniform mutation* operators act on a single trial solution. Uniform mutation changes one parameter in the trial solution to a random value uniformly distributed on the domain specified for the parameter. Boundary mutation replaces one parameter with one of the bounds of its domain. Non-uniform mutation shrinks one parameter toward one of the bounds, with the amount of shrinkage decreasing as the generation count approaches the specified maximum number of generations. *Whole non-uniform mutation* does non-uniform mutation for all the parameters in the trial solution. *Heuristic crossover* uses two trial solutions to produce a new trial solution located along a vector that starts at one trial solution and points away from the other one. *Polytope crossover* (inspired by simplex search, Gill *et al.* 1981, p. 94–95) computes a trial solution that is a convex combination of as many trial solutions as there are parameters. *Simple crossover* computes two trial solutions from two input trial solutions by swapping parameter values between the solutions after splitting the solutions at a randomly selected point. This operator can be especially effective if the ordering of the parameters in each trial solution is consequential. *Local-minimum crossover* computes a new trial solution in two steps: first it does a preset number of BFGS iterations starting from the input trial solution; then it computes a convex combination of the input solutions and the BFGS iterate.

3. Examples

The only function in the **rgenoud** package is `genoud`. The interface of this function is similar to that of the `optim` function in R. But the function has many additional arguments that control the behavior of the evolutionary algorithm.

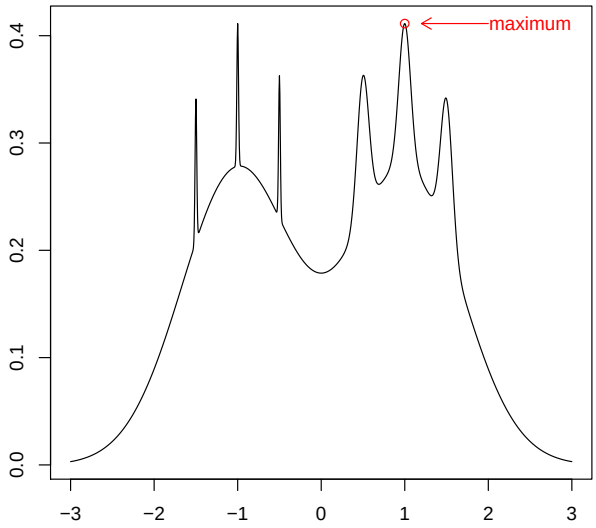

Figure 1: Normal mixture: Asymmetric double claw.

3.1. Asymmetric double claw

Our first example, which we also studied in Sekhon and Mebane (1998), is a normal mixture called the asymmetric double claw (ADC). We plot the function in Figure 1. Mathematically, this mixture is defined as

$$f_{\text{ADC}} = \sum_{m=0}^{1} \frac{46}{100} \text{N}\left(2m-1, \frac{2}{3}\right) + \sum_{m=1}^{3} \frac{1}{300} \text{N}\left(\frac{-m}{2}, \frac{1}{100}\right) + \sum_{m=1}^{3} \frac{7}{300} \text{N}\left(\frac{m}{2}, \frac{7}{100}\right), \quad (1)$$

where N is the normal density.

The asymmetric double claw is difficult to maximize because there are many local solutions. There are five local maxima in Figure 1. Standard derivative-based optimizers would simply climb up the hill closest to the starting value.

To optimize this normal mixture we must first create a function for it

```
R> claw <- function(xx) {
+    x <- xx[1]
+    y <- (0.46 * (dnorm(x, -1, 2/3) + dnorm(x, 1, 2/3)) +
+      (1/300) * (dnorm(x, -0.5, 0.01) + dnorm(x, -1, 0.01) +
+        dnorm(x, -1.5, 0.01)) +
+      (7/300) * (dnorm(x, 0.5, 0.07) + dnorm(x, 1, 0.07) +
+        dnorm(x, 1.5, 0.07)))
+    return(y)
+  }
```

And we now make a call to **rgenoud** using this function:

```
R> library("rgenoud")
R> claw1 <- genoud(claw, nvars = 1, max = TRUE, pop.size = 3000)
```

The first argument of `genoud` is the function to be optimized. The first argument of that function must be the vector of parameters over which optimizing is to occur. Generally, the function should return a scalar result.[6] The second argument of `genoud` in this example (`nvars`) is the number of variables the function to be optimized takes. The third argument, `max = TRUE`, tells `genoud` to maximize the function instead of its default behavior which is to minimize.

The fourth option `pop.size` controls the most important part of the evolutionary algorithm, the population size. This is the number of individuals `genoud` uses to solve the optimization problem. As noted in the theoretical discussion, the theorems related to evolutionary algorithms are asymptotic in the population size so larger is generally better but obviously takes longer. The maximum solution of the double claw density is reliably found by `genoud` even using the default value of `pop.size = 1000`. Reliability does increase as the `pop.size` is made larger. Unfortunately, because of the stochastic nature of the algorithm, it is impossible to generally answer the question of what is the best population size to use.

Other options determine the maximum number of generations the evolutionary algorithm computes. These options are `max.generations`, `wait.generations` and `hard.generation.limit`.

[6]The function to be optimized may return a vector if one wishes to do lexical optimization. Please see the `lexical` option to `genoud`.

The specified termination point also affects how some of the operators perform: the two non-uniform mutation operators introduce smaller ranges of variation in the trial solutions as the generation count approaches the specified `max.generations` value. There are many more options that can be used to fine-tune the behavior of the algorithm.

The output printed by `genoud` is controlled by a `print.level` argument. The default value, `print.level=2`, produces relatively verbose output that gives extensive information about the set of operators being used and the progress of the optimization. Normally R conventions would suggest setting the default to be `print.level=0`, which would suppress output to the screen, but because `genoud` runs may take a long time, it can be important for the user to receive some feedback to see the program has not died and to be able to see where the program got stuck if it eventually fails to make adequate progress.

The output printed by the preceding invocation of `genoud`, which uses the default value for a `print.level` argument, is as follows.

```
Fri Feb  9 19:33:42 2007
Domains:
 -1.000000e+01   <=  X1   <=    1.000000e+01

Data Type: Floating Point
Operators (code number, name, population)
        (1) Cloning.........................  372
        (2) Uniform Mutation................  375
        (3) Boundary Mutation...............  375
        (4) Non-Uniform Mutation............  375
        (5) Polytope Crossover..............  375
        (6) Simple Crossover................  376
        (7) Whole Non-Uniform Mutation......  375
        (8) Heuristic Crossover.............  376
        (9) Local-Minimum Crossover.........  0

HARD Maximum Number of Generations: 100
Maximum Nonchanging Generations: 10
Population size       : 3000
Convergence Tolerance: 1.000000e-03

Using the BFGS Derivative Based Optimizer on the Best Individual Each
Generation.
Checking Gradients before Stopping.
Using Out of Bounds Individuals.

Maximization Problem.
GENERATION: 0 (initializing the population)
Fitness value... 4.112017e-01
mean............ 4.990165e-02
variance........ 9.708147e-03
#unique......... 3000, #Total UniqueCount: 3000
var 1:
```

```
best..... 9.966758e-01
mean..... 3.453097e-02
variance..... 3.373681e+01

GENERATION: 1
Fitness value... 4.113123e-01
mean..... 2.237095e-01
variance..... 2.566140e-02
#unique..... 1858, #Total UniqueCount: 4858
var 1:
best..... 9.995043e-01
mean..... 4.615946e-01
variance..... 7.447887e+00

[...]

GENERATION: 10
Fitness value... 4.113123e-01
mean..... 2.953888e-01
variance..... 2.590842e-02
#unique..... 1831, #Total UniqueCount: 21708
var 1:
best..... 9.995033e-01
mean..... 8.403935e-01
variance..... 5.363241e+00

GENERATION: 11
Fitness value... 4.113123e-01
mean..... 2.908561e-01
variance..... 2.733896e-02
#unique..... 1835, #Total UniqueCount: 23543
var 1:
best..... 9.995033e-01
mean..... 8.084638e-01
variance..... 6.007372e+00

'wait.generations' limit reached.
No significant improvement in 10 generations.

Solution Fitness Value: 4.113123e-01

Parameters at the Solution (parameter, gradient):

 X[ 1] :        9.995033e-01    G[ 1] : -6.343841e-09

Solution Found Generation 1
Number of Generations Run 11
```

```
Fri Feb  9 19:33:45 2007
Total run time : 0 hours 0 minutes and 3 seconds
```

After printing the date and time of the run, the program prints the domain of values it is allowing for each parameter of the function being optimized. In this case the default domain values are being used. Naturally it is important to specify domains wide enough to include the solution. In practice with highly nonlinear functions it is often better to specify domains that are relatively wide than to have domains that narrowly and perhaps even correctly bound the solution. This surprising behavior reflects the fact with a highly nonlinear function, a point that is close to the solution in the sense of simple numerical proximity may not be all that close in the sense of there being a short feasible path to get to the solution.

Next the program prints the Data Type. This indicates whether the parameters of the function to be optimized are being treated as floating point numbers or integers. For more information about this, see the `data.type.int` argument.

The program then displays the number of operators being used, followed by the values that describe the other characteristics set for this particular run: the maximum number of generations, the population size and the tolerance value to be used to determine when the parameter values will be deemed to have converged.

The output then reports whether BFGS optimization will be applied to the best trial solution produced by the operators in each generation. For problems that are smooth and concave in a neighborhood of the global optimum, using the BFGS in this way can help `genoud` quickly converge once the best trial solution is in the correct neighborhood. This run of `genoud` will also compute the gradient at the best trial solution before stopping. In fact this gradient checking is used as a convergence check. The algorithm will not start counting its final set of generations (the `wait.generations`) until each element of the gradient is smaller in magnitude than the convergence tolerance. Gradients are never used and BFGS optimization is not used when the parameters of the function to be optimized are integers.

The next message describes how strictly `genoud` is enforcing the boundary constraints specified by the domain values. By default (`boundary.enforcement=0`), the trial solutions are allowed to wander freely outside the boundaries. The boundaries are used only to define domains for those operators that use the boundary information. Other settings of the `boundary.enforcement` argument induce either more stringent or completely strict enforcement of the boundary constraints. Notice that the boundary constraints apply to the parameters one at a time. To enforce constraints that are defined by more complicated functional or data-dependent relationships, one can include an appropriate penalty function as part of the definition of the function to be optimized, letting that function define an extremely high (if minimizing) or low (if maximizing) value to be returned if the desired conditions are violated.

After reporting whether it is solving a minimization or a maximization problem, `genoud` reports summary statistics that describe the distribution of the fitness value and the parameter values across the trial solutions at the end of each generation. In the default case where `genoud` is keeping track of all the distinct solutions considered over the whole course of the optimizing run, these generational reports also include a report of the number of unique trial solutions in the current population and the number of unique solutions ever considered. The benefit of keeping track of the solutions is to avoid repeatedly evaluating the function being optimized for the identical set of parameter values. This can be an important efficiency when

function evaluations are expensive, as they can be in statistical applications where the data are extensive. This tracking behavior is controlled by the `MemoryMatrix` argument.

Upon convergence, or when the hard maximum generation limit is reached, the program prints the fitness value at the best trial solution and that solution's parameter values. In this case the solution was found after one generation. While the Asymmetric Double Claw might present a difficult challenge for a gradient-based optimizer that uses only local hill climbing, it is an almost trivially simple problem for `genoud`.

3.2. Benchmark functions

The second example is a suite of 23 benchmark nonlinear functions used in Yao *et al.* (1999) to study a pair of evolutionary programming optimization algorithms. Function definitions are in Table 2. Because it includes a random component and so lacks a reproducible minimum value, we ignore the function numbered function 7 in their sequence.[7] Implementations of these functions are available in the supplemental R file provided with this article. These R definitions include the values of the constants used in functions 14, 15 and 19 through 23. The function argument domains are restricted to the specific domains used by Yao *et al.* (1999) via bounds that are stated in the list named `testbounds` in the supplemental file.

As Yao *et al.* (1999) describe, optimizing each of functions 1–13 presents a high-dimensional problem. These functions each have $n = 30$ free parameters. Functions 1–5 are unimodal, with function 5 being a 30-dimensional version of the banana-shaped Rosenbrock function. Function 6 is a step function. Function 6 has one minimum value that occurs when all arguments are in the interval $x_i \in [0, 0.5)$, and the function is discontinuous. Functions 8–13 are multimodal, defined such that the number of local minima increases exponentially with the number of arguments. Yao *et al.* (1999, p. 84) describe these functions as among "the most difficult class of problems for many optimization algorithms (including [evolutionary programming])." Functions 14–23, which have between two and six free parameters each, each have only a few local minima. Nonetheless the evolutionary programming algorithms considered by Yao *et al.* (1999) have trouble optimizing functions 21–23. Although Yao *et al.* (1999, p. 85) state that each of these functions has a minimum value of -10, over 50 replications the two algorithms they consider achieve solutions averaging between -5.52 and -9.10 (Yao *et al.* 1999, p. 88, Table IV).

We use these benchmark functions to illustrate not only how effective `genoud` can be with a range of difficult problems, but also to emphasize an important aspect of how one should think about trying to tune the optimizer's performance. Theory regarding genetic algorithms suggests that optimal solutions are more likely to appear as both the population size of candidate solutions and the number of generations increase. In `genoud` two arguments determine the number of generations. One is `max.generations`: if `hard.generation.limit = TRUE`, then the value of `max.generations` is a binding upper limit. The other is `wait.generations`, which determines when the algorithm terminates if `hard.generation.limit = FALSE`. But

[7]The omitted function is $\sum_{i=1}^{n} ix_i^4 + U(0,1)$, where $U(0,1)$ is a uniformly distributed random variable on the unit interval that takes a new value whenever the function is evaluated. This stochastic aspect means that even given the set of parameters that minimize the nonstochastic component $\sum_{i=1}^{n} ix_i^4$, i.e., $x_i = 0$, the value of the function virtually never attains the minimum possible value of zero. An optimizer that evaluated the function at $x_i = 0$ would not in general obtain a function value smaller than the function value obtained for a wide range of different parameter values. Hence we do not consider this function to be a good test for function optimization algorithms.

No.	Definition	n	Minimum
1	$\sum_{i=1}^{n} x_i^2$	30	0
2	$\sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert$	30	0
3	$\sum_{i=1}^{n} (\sum_{j=1}^{i} x_j)^2$	30	0
4	$\max_i \{\lvert x_i \rvert, 1 \leq i \leq n\}$	30	0
5	$\sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$,	30	0
6	$\sum_{i=1}^{n} (\lfloor x_i + 0.5 \rfloor)^2$	30	0
7	$\sum_{i=1}^{n} i x_i^4 + U(0,1)$	30	0
8	$\sum_{i=1}^{n} -x_i \sin(\sqrt{\lvert x_i \rvert})$	30	-12569.5
9	$\sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i) + 10]$	30	0
10	$-20 \exp\left(-0.2\sqrt{n^{-1}\sum_{i=1}^{n} x_i^2}\right)$ $- \exp(n^{-1}\sum_{i=1}^{n} \cos 2\pi x_i) + 20 + e$	30	0
11	$(1/1000)\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(x_i/\sqrt{i}\right) + 1$	30	0
12	$n^{-1}\pi \left\{10\sin^2(\pi y_1) + \sum_{i=1}^{n-1}(y_i - 1)^2[1 + 10\sin^2(\pi y_{i+1})] + (y_n - 1)^2\right\} + \sum_{i=1}^{n} u(x_i, 10, 100, 4); y_i = 1 + (x_i + 1)/4;$ $u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m, & x_i > a, \\ 0, & -a \leq x_i \leq a, \\ k(-x_i - a)^m, & x_i < -a, \end{cases}$	30	0
13	$\left\{\sin^2(3\pi x_1) + \sum_{i=1}^{n-1}(x_i - 1)^2[1 + \sin^2(3\pi x_{i+1})]\right\}/10$ $+(x_n - 1)[1 + \sin^2(2\pi x_n)]/10 + \sum_{i=1}^{n} u(x_i, 5, 100, 4)$	30	0
14	$\left\{1/500 + \sum_{j=1}^{25} 1/\left[j + \sum_{i=1}^{2}(x_i - a_{ij})^6\right]\right\}^{-1}$	2	1
15	$\sum_{i=1}^{11} [a_i - x_1(b_i^2 + b_i x_2)/(b_i^2 + b_i x_3 + x_4)]^2$	4	0.0003075
16	$4x_1^2 - 2.1x_1^4 + x_1^6/3 + x_1 x_2 - 4x_2^2 + 4x_2^4$	2	-1.0316285
17	$[x_2 - 5.1x_1^2/(4\pi^2) + 5x_1/\pi - 6]^2 + 10[1 - 1/(8\pi)]\cos(x_1) + 10$	2	0.398
18	$[1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \times$ $[30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$	2	3
19	$-\sum_{i=1}^{4} c_i \exp\left[-\sum_{j=1}^{4} a_{ij}(x_i - p_{ij})^2\right]$	4	-3.86
20	$-\sum_{i=1}^{4} c_i \exp\left[-\sum_{j=1}^{6} a_{ij}(x_i - p_{ij})^2\right]$	6	-3.32
21	$\sum_{i=1}^{5} [(x - a_i)'(x - a_i) + c_i]^{-1}$	4	-10
22	$\sum_{i=1}^{7} [(x - a_i)'(x - a_i) + c_i]^{-1}$	4	-10
23	$\sum_{i=1}^{10} [(x - a_i)'(x - a_i) + c_i]^{-1}$	4	-10

Table 2: 23 benchmark functions. The minimum function value within specified bounds is given by Yao *et al.* (1999, p. 85).

even if `hard.generation.limit = TRUE`, then `wait.generations` determines for how many generations the algorithm continues once the best parameter vector and the value of the function being optimized appear to have settled down. The fact that the current best solution is not changing should not be treated as decisive, because this solution may be merely a local optimum or a saddlepoint. If the population size is sufficient, the algorithm tends to build a population of trial solutions that contain parameters in neighborhoods of all the competitive local optima in the domain defined by the parameter boundaries. Even while the current best

solution is stable, the algorithm is improving the solutions near other local optima. So having a higher `wait.generations` value never worsens the algorithm's efficacy.

Increasing `max.generations` may or may not in itself improve optimization. The value of `max.generations` sets the value of T used in the mutation operators—operators 4 and 7 in Table 1. These mutation operators perform random search near a trial solution that has been selected for mutation only when the current generation count is an appreciable fraction of T. So increasing `max.generations` without changing `wait.generations` increases the period during which random search is occurring over a wider domain. For multimodal functions such a wider search may be helpful, but sometimes failing to search more densely near the current trial solutions is not good.

We use `genoud` to minimize the 23 functions using two values for `pop.size` (5000 and 10000) and two values for `max.generations` (30 and 100). Following Yao *et al.* (1999), we replicate each optimization 50 times. The following code describes the computations. The list `testfuncs`, vector `testNparms` and list `testbounds` are defined in the supplemental R file, and it is assume that this file is loaded with the `source("v42i11-extra.R")` command. The vector `gradcheck` is true for all elements except the ones corresponding to functions 6 and 7.

```
R> source("v42i11-extra.R")
R> gradcheck <- rep(TRUE, 23)
R> gradcheck[6:7] <- FALSE
R> sizeset <- c(5000, 10000)
R> genset <- c(30, 100)
R> nreps <- 50
R> gsarray <- array(NA, dim = c(length(sizeset), length(genset), 23, nreps))
R> asc <- function(x) as.character(x)
R> dimnames(gsarray) <- list(asc(sizeset), asc(genset), NULL, NULL)
R> for (gsize in sizeset) {
+    for (ngens in genset) {
+      for (i in 1:23) {
+        for (j in 1:nreps) {
+          gsarray[as.character(gsize), as.character(ngens), i, j] <-
+            genoud(testfuncs[[i]], nvars = testNparms[i], pop.size = gsize,
+              max.gen = ngens, hard.gen = TRUE, Domains = testbounds[[i]],
+              solution.tol = 1e-06, boundary = 1, gradient.check =
+              gradcheck[i], print = 0)$value
+        }
+      }
+    }
+  }
```

Using `genoud` to minimize the benchmark functions produces excellent results, at least when the `pop.size` and `max.generations` arguments are sufficiently large. Table 3 reports the mean function values for each configuration of the arguments. These values may be compared both to the true function minima given by Yao *et al.* (1999) (see the rightmost column in Table 2) and to the average minimum function values Yao *et al.* (1999) report for their "fast" evolutionary programming (FEP) algorithm, which appear in the last column of Table 3. The `genoud` averages for the `max.gen = 100` configurations equal or are close to the true minima

	genoud				
	pop.size = 5000		pop.size = 10000		
No.	max = 30	max = 100	max = 30	max = 100	FEP
1	1.453e-32	1.658e-32	2.416e-32	6.134e-32	5.7e-4
2	6.55e-16	7.212e-16	9.652e-16	1.043e-15	8.1e-3
3	4.633e-18	3.918e-18	3.787e-18	4.032e-18	1.6e-2
4	6.203e-17	6.542e-17	9.453e-17	7.85e-17	0.3
5	0.07973	5.887e-08	8.133e-08	8.917e-08	5.06
6	18.58	0.08	9.38	0	0
8	−12569.49	−12569.49	−12569.49	−12569.49	−12554.5
9	2.786	0	0.9353	0	4.6e-2
10	2.849	3.997e-15	2.199	3.997e-15	1.8e-2
11	7.994e-17	7.105e-17	9.548e-17	6.439e-17	1.6e-2
12	5.371e-19	0.004147	0.002073	1.178e-19	9.2e-6
13	0.02095	0.006543	0.006629	0.003011	1.6e-4
14	0.998	0.998	0.998	0.998	1.22
15	0.0003441	0.0004746	0.0003807	0.0003807	5.0e-4
16	−1.0316285	−1.0316285	−1.0316285	−1.0316285	−1.03
17	0.3979	0.3979	0.3979	0.3979	0.398
18	3	3	3	3	3.02
19	−3.863	−3.863	−3.863	−3.863	−3.86
20	−3.274	−3.277	−3.279	−3.286	−3.27
21	−9.85	−9.444	−9.95	−10.05	−5.52
22	−9.771	−10.09	−10.19	−10.3	−5.52
23	−10.1	−9.997	−10.32	−10.21	−6.57

Table 3: Mean values of 22 optimized functions. Average minimum function values (over 50 replications) obtained using `genoud`. Mean best function values (over 50 replications) reported for the "fast" evolutionary programming (FEP) algorithm, from Yao *et al.* (1999, p. 85 and 88, Tables II–IV).

for all the functions except function 13. One can reasonably argue that the average solutions for function 5 are not as close to zero as might be desired: these averages are close to 10^{-7}, while the averages for other functions that have a true minimum of zero are 10^{-15} or smaller. And the averages for functions 6, 12 and 15 in the `pop.size = 5000` case are off. The effect of increasing `pop.size` is apparent with respect to both those three functions and also functions 13 and 20–23: the average minima are smaller with `pop.size = 10000` than with `pop.size = 5000`. Except for functions 6 and 12 in the `pop.size = 5000` case and function 13, all the `genoud` solution averages for `max.gen = 100` are either slightly or substantially better than the corresponding FEP solution averages.

The results in Table 3 clearly illustrate the potential consequences of not allowing `genoud` to run for a sufficient number of generations. While some of the `genoud` solutions for `max.gen = 30` have competitive means, several of the means are not good at all.

The effect of increasing `pop.size` are even more clearly apparent in Table 4, which reports the standard deviations of the respective minima across the 50 replications. With the exceptions

	genoud				
	pop.size = 5000		pop.size = 10000		
No.	max = 30	max = 100	max = 30	max = 100	FEP
1	9.997e-32	7.059e-32	9.562e-32	2.1e-31	1.3e-4
2	1.668e-15	1.621e-15	2.116e-15	2.102e-15	7.7e-4
3	4.568e-18	3.342e-18	4.38e-18	5.136e-18	1.4e-2
4	1.793e-16	1.758e-16	2.055e-16	2.002e-16	0.5
5	0.5638	4.921e-08	5.573e-08	4.955e-08	5.87
6	5.65	0.274	3.528	0	0
8	3.749e-10	1.071e-12	8.948e-09	6.365e-13	52.6
9	1.864	0	1.179	0	1.2e-2
10	0.7146	0	0.702	0	2.1e-3
11	1.209e-16	1.582e-16	1.289e-16	8.713e-17	2.2e-2
12	2.336e-18	0.02052	0.01466	7.423e-19	3.6e-6
13	0.03427	0.006867	0.006903	0.001508	7.3e-5
14	5.638e-12	8.894e-11	1.029e-12	4.35e-12	0.56
15	0.0001813	0.0003546	0.000251	0.0002509	3.2e-4
16	1.315e-14	9.751e-15	1.233e-14	1.054e-14	4.9e-7
17	5.422e-15	5.51e-15	4.925e-15	1.392e-14	1.5e-7
18	1.509e-13	3.477e-14	6.18e-14	2.907e-14	0.11
19	7.349e-15	1.521e-15	1.344e-14	7.255e-15	1.4e-5
20	0.05884	0.0583	0.05765	0.05504	5.9e-2
21	1.212	1.776	1.005	0.7145	1.59
22	1.937	1.269	1.052	0.7459	2.12
23	1.479	1.636	1.066	1.29	3.14

Table 4: Standard deviations of values of 22 optimized functions. Standard deviation of the minimum function values (over 50 replications) obtained using `genoud`. Standard deviation of the best function values (over 50 replications) reported for the "fast" evolutionary programming (FEP) algorithm, from Yao *et al.* (1999, p. 85 and 88, Tables II–IV).

of functions 6, 12, 13, 15 and 21 with `pop.size = 5000`, the `genoud` solutions for `max.gen = 100` vary much less than the corresponding FEP solutions. For those functions and also for functions 20, 22 and 23, the `max.gen = 100` solutions with `pop.size = 10000` vary noticeably less than the solutions with `pop.size = 5000`.

3.3. A logistic least quartile difference estimator

Our third example is a version of the LQD estimator used in `multinomRob`. Using the R function `IQR` to compute the interquartile range, the function to be minimized may be defined as follows.[8]

[8]The LQD problem solved in `multinomRob` is somewhat different. There the problem is to minimize the $\binom{h_K}{2}$ order statistic of the set of absolute differences among the standardized residuals, where h_K is a function of the sample size and the number of unknown regression model coefficients (Mebane and Sekhon 2004). The problem considered in the current example is simpler but exhibits similar estimation difficulties.

```
R> LQDxmpl <- function(b) {
+    logistic <- function(x) 1/(1 + exp(-x))
+    sIQR <- function(y, yhat, n) IQR((y - yhat)/sqrt(yhat * (n - yhat)),
+      na.rm = TRUE)
+    sIQR(y, m * logistic(x %*% b), m)
+  }
```

For this example we define `LQDxmpl` after we compute the simulated data, so the data vector `y`, matrix `x` and scalar `m` are in scope to evaluate to have the values we simulate:

```
R> m <- 100
R> x <- cbind(1, rnorm(1000), rnorm(1000))
R> b1 <- c(0.5, 1, -1)
R> b2 <- c(0, -1, 1)
R> logistic <- function(x) 1/(1 + exp(-x))
R> y <- rbinom(1000, m, logistic(c(x[1:900, ] %*% b1, x[901:1000, ] %*% b2)))
```

The data simulate substantial contamination. The first 900 observations are generated by one binomial regression model while the last 100 observations come from a very different model.

Presuming we are interested in the model that generated the bulk of the data, ignoring the contamination in a generalized linear model with the binomial family produces very poor results:

```
R> summary(glm1 <- glm(cbind(y, m - y) ~ x[, 2] + x[, 3], family = binomial))

Call:
glm(formula = cbind(y, m - y) ~ x[, 2] + x[, 3], family = binomial)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-22.9168   -1.1693    0.3975    1.5895   24.6439

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.439492   0.007097   61.93   <2e-16 ***
x[, 2]       0.679847   0.007985   85.14   <2e-16 ***
x[, 3]      -0.716963   0.007852  -91.31   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Of course, if we knew which observations to omit the results would be much better:

```
R> suby <- y[1:900]
R> subx <- x[1:900, ]
R> summary(glm2 <- glm(cbind(suby, m - suby) ~ subx[, 2] + subx[, 3],
+    family = binomial))
```

```
Call:
glm(formula = cbind(suby, m - suby) ~ subx[, 2] + subx[, 3],
    family = binomial)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-3.21478  -0.71699   0.03528   0.67867   2.88314

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.501880   0.008036   62.46   <2e-16 ***
subx[, 2]    1.003592   0.009779  102.63   <2e-16 ***
subx[, 3]   -0.984295   0.009437 -104.30   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

But in practical applications it is unknown apriori which observations should be treated as outliers.

As the definition of `LQDxmpl` indicates, the LQD is based on minimizing the interquartile range (IQR) of the standardized residuals. Because the quartiles correspond to different data points for different values of the regression coefficients, the fitness function is not smooth, which is to say it is not everywhere differentiable. In general, at every point in the parameter space where the identity of the first or third quartile point changes, the function is not differentiable. Figure 2 illustrates this. The IQR clearly has a minimum in the vicinity of the coefficient values used to generate most of the data. But contour plots for the numerically evaluated partial derivative with respect to the second coefficient parameter testify to the function's local irregularity. The function we use to evaluate this numerical derivative is defined as follows.

```
R> dLQDxmpl <- function(b) {
+    eps <- 1e-10
+    logistic <- function(x) 1/(1 + exp(-x))
+    sIQR <- function(y, yhat, n) IQR((y - yhat)/sqrt(yhat * (n - yhat)),
+      na.rm = TRUE)
+    dsIQR <- vector()
+    for (i in 1:length(b)) {
+      beps <- b
+      beps[i] <- b[i] + eps
+      dsIQR <- c(dsIQR, (sIQR(y, m * logistic(x %*% beps), m) -
+        sIQR(y, m * logistic(x %*% b), m))/eps)
+    }
+    return(dsIQR)
+  }
```

Setting the intercept equal to 0.5, the code to generate the plotted values is

```
R> blen <- 3
R> lenbseq <- length(bseq <- seq(-2, 2, length = 200))
```

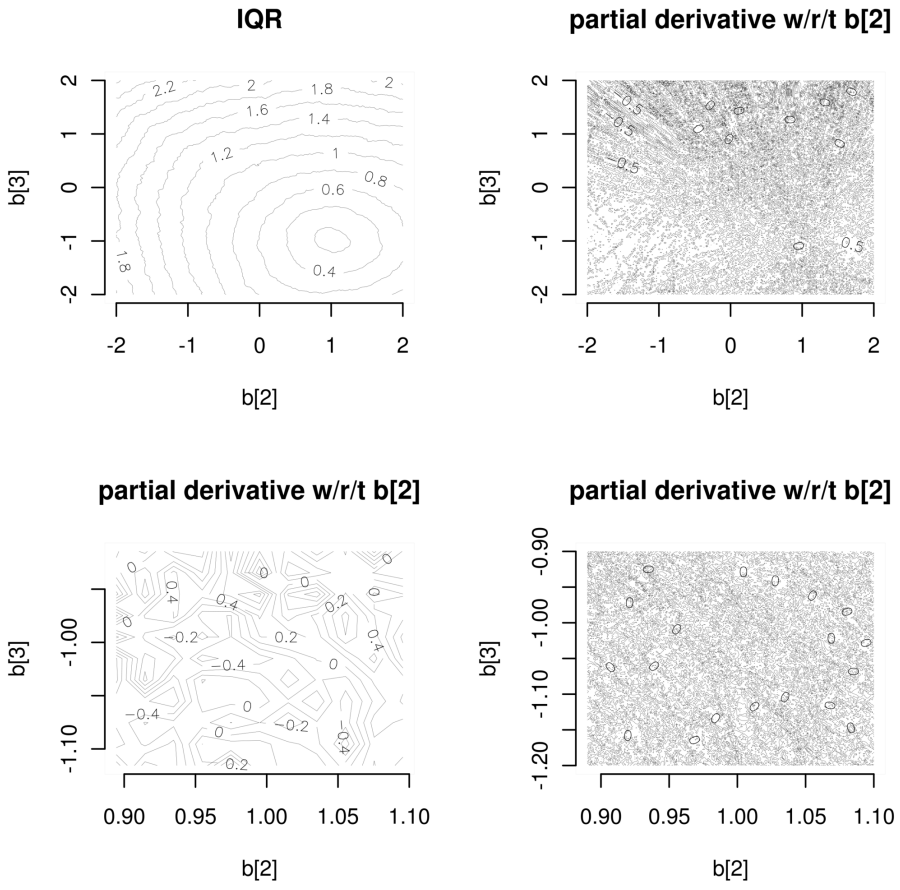

Figure 2: Contour plots of the LQD fitness function and of its partial derivatives.

```
R> bseq3 <- seq(-1.2, -0.9, length = 200)
R> bseq2 <- seq(0.89, 1.1, length = 200)
R> IQRarr <- IQRarrA <- array(NA, dim = c((1 + blen), lenbseq, lenbseq))
R> dimnames(IQRarrA) <- list(NULL, as.character(bseq), as.character(bseq))
R> dimnames(IQRarr) <- list(NULL, as.character(bseq2), as.character(bseq3))
R> for (i in 1:lenbseq) {
+    for (j in 1:lenbseq) {
+      IQRarrA[1, i, j] <- LQDxmpl(c(0.5, bseq[i], bseq[j]))
+      IQRarrA[-1, i, j] <- dLQDxmpl(c(0.5, bseq[i], bseq[j]))
+      IQRarr[1, i, j] <- LQDxmpl(c(0.5, bseq2[i], bseq3[j]))
+      IQRarr[-1, i, j] <- dLQDxmpl(c(0.5, bseq2[i], bseq3[j]))
+    }
+  }
```

The following code produces the plots:

```
R> par(mfrow = c(2, 2), lwd = 0.1)
```

```
R> contour(bseq, bseq, IQRarrA[1,,], main = "IQR", xlab = "b[2]",
+    ylab = "b[3]")
R> contour(bseq, bseq, IQRarrA[3,,], main = "partial derivative w/r/t b[2]",
+    xlab = "b[2]", ylab = "b[3]")
R> loc2 <- (150:160) - 5
R> loc3 <- (40:50) + 5
R> contour(bseq[loc2], bseq[loc3], IQRarrA[3, loc2, loc3],
+    main = "partial derivative w/r/t b[2]", xlab = "b[2]", ylab = "b[3]")
R> contour(bseq2, bseq3, IQRarr[3,,], main = "partial derivative w/r/t b[2]",
+    xlab = "b[2]", ylab = "b[3]")
```

If the IQR function were smooth, we would see clearly separated, slightly curved contour lines, reflecting the nonlinearity of the `logistic` function, but there is nothing like that. Instead, looking over the domain $[-2, 2]^2$ for the second and third regression coefficient parameters, with 200 points evaluated along each axis (the upper right plot), there is a splotchy cloud. This reflects the fact that the derivative changes sign very frequently over the domain: of the 40,000 points at which the derivative is evaluated, it is positive at 12,460 points and negative at 27,540 points.

The LQD fitness function is not appreciably smoother close to the true values. The bottom two plots show the partial derivatives with respect to the second coefficient parameter evaluated over the domain $[.89, 1.1] \times [-1.2, -.9]$. The bottom left plot evaluates the derivatives at 11 points along each axis while the bottom right plot uses 200 points along each axis. In the left plot it is easier to see the intricacy of the partitioning of the parameter space as the identity of the first or third quartile point changes. The bottom right plot shows the intricacy in fact replicates at the finer grain of the more detailed evaluations (to see this it may be necessary to magnify the plot while viewing the online version of this article). Within this smaller domain the sign of the derivative changes even more frequently than it does over the domain $[-2, 2]^2$: the derivative is positive at 18,098 points and negative at 21,902 points.

While the LQD fitness function may be differentiable in a neighborhood of the global solution, that neighborhood, if it exists, is clearly not very big. As likely is that the global solution is located at a point where the function is not differentiable. Hence a numerically evaluated gradient is not meaningful for evaluating whether the solution has been found. At the true solution, numerical gradient values may differ substantially from zero.

To use the LQD to estimate the binomial regression model parameters we use the following call to `genoud`. Because gradient information is of questionable relevance for this problem, we turn off the termination condition that the gradient at the solution be smaller than the value specified for the `solution.tolerance` argument. We retain the default setting `BFGS = TRUE` because, in principle, gradient-driven optimization may help in each of the many differentiable neighborhoods, even if it is useless across the nondifferentiable boundaries. Our experience optimizing the LQD (Mebane and Sekhon 2004) shows that using the BFGS in this way improves performance, even though the gradient is not useful for evaluating whether the solution has been found.

```
R> LQD1 <- genoud(LQDxmpl, nvars = 3, max = FALSE, pop.size = 2000,
+    max.generations = 300, wait.generations = 100, gradient.check = FALSE,
+    print = 1)
```

This invocation of the `LQDxmpl` function matches the behavior of `multinomRob` in that it produces an estimate for the intercept parameter along with the other coefficients. In a linear regression context, the interquartile range statistic contains no information about the intercept, so the LQD is not an estimator for that parameter. With a binomial regression model there is some information about the intercept due to the nonlinearity of the `logistic` function. The LQD estimate for the intercept should nonetheless be expected not to be very good.

Results from the preceding estimation are as follows.

```
Tue Aug  7 02:27:08 2007
Domains:
 -1.000000e+01   <=  X1   <=    1.000000e+01
 -1.000000e+01   <=  X2   <=    1.000000e+01
 -1.000000e+01   <=  X3   <=    1.000000e+01

[...]

HARD Maximum Number of Generations: 300
Maximum Nonchanging Generations: 100
Population size       : 2000
Convergence Tolerance: 1.000000e-06

Using the BFGS Derivative Based Optimizer on the Best Individual Each
Generation.
Not Checking Gradients before Stopping.
Using Out of Bounds Individuals.

Minimization Problem.


Generation#            Solution Value

      0         4.951849e-01
     56         1.298922e-01
     57         1.298891e-01
     59         1.298848e-01
     60         1.298820e-01
     61         1.298793e-01
     62         1.298768e-01
     63         1.298744e-01

'wait.generations' limit reached.
No significant improvement in 100 generations.

Solution Fitness Value: 1.298743e-01

Parameters at the Solution (parameter, gradient):
```

```
 X[ 1] :        8.130357e-02    G[ 1] : 9.616492e-03
 X[ 2] :        8.889485e-01    G[ 2] : -1.167897e-01
 X[ 3] :        -9.327966e-01   G[ 3] : -3.090130e-02

Solution Found Generation 63
Number of Generations Run 164

Tue Aug  7 02:31:09 2007
Total run time : 0 hours 4 minutes and 1 seconds
```

Recall that the gradient is not reliably informative at the solution. To check whether this solution is believable, we might try reestimating the model using a larger population and larger specified number of generations:

```
R> LQD1 <- genoud(LQDxmpl, nvars = 3, max = FALSE, pop.size = 10000,
+    max.generations = 1000, wait.generations = 300, gradient.check = FALSE,
+    print = 1)
```

At the price of a greatly increased running time (from four minutes up to one hour 53 minutes), the results are better than the first run (even though the summary measure of fit is slightly worse):

```
Minimization Problem.


Generation#            Solution Value

      0         2.238865e-01
      2         1.301149e-01
      3         1.300544e-01
      4         1.300482e-01
      6         1.300375e-01
      7         1.300343e-01
      8         1.300323e-01
    134         1.299662e-01
    135         1.299099e-01
    136         1.298867e-01
    137         1.298843e-01
    138         1.298822e-01
    139         1.298791e-01
    141         1.298774e-01

'wait.generations' limit reached.
No significant improvement in 300 generations.

Solution Fitness Value: 1.298770e-01
```

```
Parameters at the Solution (parameter, gradient):

 X[ 1] :        2.013748e-01    G[ 1] : -7.394125e-02
 X[ 2] :        9.526390e-01    G[ 2] : 7.807607e-02
 X[ 3] :        -9.642458e-01   G[ 3] : 3.834052e-02

Solution Found Generation 141
Number of Generations Run 442

Tue Aug  7 05:16:37 2007
Total run time : 1 hours 53 minutes and 45 seconds
```

This example demonstrates a key difficulty that arises when optimizing irregular functions in the absence of gradients. It is difficult to assess when or whether an optimum has been found. The estimated coefficient values are close to the ones used to generate most of the data, except as expected the estimate for the intercept is not good. The estimates are better than if we had ignored the possibility of contamination. But whether these are the best possible estimates is not clear. If we were to use an even larger population and specify that an even greater number of generations be run, perhaps a better solution would be found.

Even for less irregular problems convergence is difficult to determine. Nonlinear optimizers often report false convergence, and users should not simply trust whatever convergence criteria an optimizer uses. McCullough and Vinod (2003) offer four criteria for verifying the solution of a nonlinear solver. These criteria are meaningful only for problems that meet regularity conditions at the solution, notably differentiability, and as such are not useful for the LQD example offered above. The four criteria are: (1) making sure the gradients are zero; (2) inspecting the solution path (i.e., the trace) to make sure it follows the expected rate of convergence; (3) evaluating the Hessian to make sure it is well-conditioned;[9] and (4) profiling the likelihood to ascertain if a quadratic approximation is adequate. One may need to take the results from `genoud` and pass them to `optim` to conduct some of these diagnostic tests such as to profile the likelihood. It is also good practice to use more than one optimizer to verify the solution (Stokes 2004).

Note that `genoud` uses its own random number seeds and internal pseudorandom number generators to insure backward compatibility with the original C version of the software and to make cluster behavior more predictable. These seeds are set by the `unif.seed` and `int.seed` options. The R `set.seed` command is ignored by `genoud`.

4. Conclusion

The `genoud` function provides many more options than can be reviewed in this brief paper. These options are described in the R help file. The most important option influencing how

[9]Following an exchange with Drukker and Wiggins (2004), McCullough and Vinod (2004a) modify their third suggestion to note that determining if the Hessian is well-conditioned in part depends on how the data are scaled. That is, a Hessian that appears to be ill-conditioned may be made well-conditioned by rescaling. So if an Hessian appears to be ill-conditioned, McCullough and Vinod (2004a) recommend that the analyst attempt to determine if rescaling the data can result in a well-conditioned Hessian.

well the evolutionary algorithm works is the `pop.size` argument. This argument controls the population size—i.e., it is the number of individuals `genoud` uses to solve the optimization problem. As noted above, the theorems proving that genetic algorithms find good solutions are asymptotic in both population size and the number of generations. It is therefore important that the `pop.size` value not be small. On the other hand, computational time is finite so obvious trade-offs must be made. As the LQD example illustrates, a larger population size is not necessarily demonstrably better. The most important options to ensure that a good solution is found, aside from `pop.size`, are `wait.generations`, `max.generations` and `hard.generation.limit`.

Many statistical models have objective functions that are nonlinear functions of the parameters, and optimizing such functions is tricky business (Altman, Gill, and McDonald 2003). Optimization difficulties often arise even for problems that are generally considered to be simple. For a cautionary tale on how optimization can be a difficult task even for such problems see Stokes' (2004) effort to replicate a probit model estimated by Maddala (1992, p. 335). A recent controversy over estimating a nonlinear model estimated by maximum likelihood offers another cautionary tale (Drukker and Wiggins 2004; McCullough and Vinod 2003, 2004b,a; Shachar and Nalebuff 2004). End users are generally surprised to learn that such optimization issues can arise, and that results can substantially vary across optimizers and software implementations.

The **rgenoud** package provides a powerful and flexible global optimizer. When compared with traditional derivative-based optimization methods, **rgenoud** performs well (Sekhon and Mebane 1998). Optimization of irregular functions is, however, as much of an art as science. And an optimizer cannot be used without thought even for simple surfaces, let alone spaces that require a genetic algorithm. We hope that the availability of a scalable global optimizer will allow users to work with difficult functions more effectively.

References

Abadie A, Diamond A, Hainmueller J (2011). "**Synth**: An R Package for Synthetic Control Methods in Comparative Case Studies." *Journal of Statistical Software*, **42**(13), 1–17. URL http://www.jstatsoft.org/v42/i13/.

Abadie A, Gardeazabal J (2003). "The Economic Costs of Conflict: A Case-Control Study for the Basque Country." *American Economic Review*, **92**(1).

Altman M, Gill J, McDonald M (2003). *Numerical Issues in Statistical Computing for the Social Scientist*. John Wiley & Sons, New York.

Altman M, McDonald MP (2011). "**BARD**: Better Automated Redistricting." *Journal of Statistical Software*, **42**(4), 1–28. URL http://www.jstatsoft.org/v42/i04/.

Billingsley P (1986). *Probability and Measure*. John Wiley & Sons, New York.

Braumoeller BF (2003). "Causal Complexity and the Study of Politics." *Political Analysis*, **11**(3), 209–233.

Braumoeller BF, Goodrich B, Kline J (2010). ***boolean**: Boolean Binary Response Models*. R package version 2.0-2, URL http://CRAN.R-project.org/package=boolean.

Davis L (ed.) (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York.

De Jong KA (1993). "Genetic Algorithms Are Not Function Optimizers." In LD Whitley (ed.), *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Mateo, CA.

Diamond A, Hainmueller J (2008). ***Synth**: Synthetic Control Group Method for Comparative Case Studies*. R package version 0.1-6, URL http://CRAN.R-project.org/package=Synth.

Drukker DM, Wiggins V (2004). "Verifying the Solution from a Nonlinear Solver: A Case Study: Comment." *American Economic Review*, **94**(1), 397–399.

Efron B, Tibshirani RJ (1994). *An Introduction to the Bootstrap*. Chapman & Hall, New York.

Feller W (1970). *An Introduction to Probability Theory and Its Applications*. 3rd edition. John Wiley & Sons, New York. Volume 1, revised printing.

Filho Jose L Ribeiro PCT, Alippi C (1994). "Genetic Algorithm Programming Environments." *Computer*, **27**, 28–43.

Gill PE, Murray W, Wright MH (1981). *Practical Optimization*. Academic Press, San Diego.

Ginsbourger D, Roustant O (2010). ***DiceOptim**: Kriging-Based Optimization for Computer Experiments*. R package version 1.0, URL http://cran.r-project.org/package=DiceOptim.

Goldberg DE (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

Goodrich B (2011). ***FAiR**: Factor Analysis in R*. R package version 0.4-7, URL http://cran.r-project.org/package=FAiR.

Grefenstette JJ (1993). "Deception Considered Harmful." In LD Whitley (ed.), *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Mateo, CA.

Grefenstette JJ, Baker JE (1989). "How Genetic Algorithms Work: A Critical Look at Implicit Parallelism." In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 20–27. Morgan Kaufmann, San Mateo, CA.

Holland JH (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

King G, Wand J (2007). "Comparing Incomparable Survey Responses: Evaluating and Selecting Anchoring Vignettes." *Political Analysis*, **15**, 46–66.

Kuhnt S, Rudak N (2011). ***JOP**: Joint Optimization Plot*. R package version 2.0.1, URL http://cran.r-project.org/package=JOP.

Lee CY, Lee YJ (2009a). ***PKfit**: A Tool for Data Analysis in Pharmacokinetics*. R package version 1.1.8, URL http://CRAN.R-project.org/package=PKfit.

Lee HY, Lee YJ (2009b). ***ivivc**: In Vitro-In Vivo Correlation (IVIVC) Modeling*. R package version 0.1.5, URL http://CRAN.R-project.org/package=ivivc.

Maddala G (1992). *Introduction to Econometrics*. 2nd edition. MacMillan, New York.

McCullough BD, Vinod HD (2003). "Verifying the Solution from a Nonlinear Solver: A Case Study." *American Economic Review*, **93**(3), 873–892.

McCullough BD, Vinod HD (2004a). "Verifying the Solution from a Nonlinear Solver: A Case Study: Reply." *American Economic Review*, **94**(1), 400–403.

McCullough BD, Vinod HD (2004b). "Verifying the Solution from a Nonlinear Solver: A Case Study: Reply." *American Economic Review*, **94**(1), 391–396.

Mebane Jr WR, Sekhon JS (1997). *GENetic Optimization Using Derivatives (GENOUD)*. Computer program available upon request.

Mebane Jr WR, Sekhon JS (2004). "Robust Estimation and Outlier Detection for Overdispersed Multinomial Models of Count Data." *American Journal of Political Science*, **48**(2), 391–410.

Mebane Jr WR, Sekhon JS (2009). ***multinomRob**: Robust Estimation of Overdispersed Multinomial Regression Models*. R package version 1.8-4, URL http://CRAN.R-project.org/package=multinomRob.

Michalewicz Z (1992). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York.

Michalewicz Z, Swaminathan S, Logan TD (1993). "GENOCOP Version 2.0." C language computer program source code, URL http://www.cs.adelaide.edu.au/~zbyszek/EvolSyst/genocop2.0.tar.Z.

Nix AE, Vose MD (1992). "Modeling Genetic Algorithms with Markov Chains." *Annals of Mathematics and Artificial Intelligence*, **5**, 79–88.

Picheny V, Ginsbourger D (2011). ***KrigInv**: Kriging-Based Inversion for Deterministic and Noisy Computer Experiments*. R package version 1.1, URL http://CRAN.R-project.org/package=KrigInv.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Sekhon JS (2011a). "Multivariate and Propensity Score Matching Software with Automated Balance Optimization: The **Matching** Package for R." *Journal of Statistical Software*, **42**(7), 1–52. URL http://www.jstatsoft.org/v42/i07/.

Sekhon JS (2011b). ***Matching**: Multivariate and Propensity Score Matching with Automated Balance Search*. R package version 4.7-12, URL http://CRAN.R-project.org/package=Matching.

Sekhon JS, Mebane Jr WR (1998). "Genetic Optimization Using Derivatives: Theory and Application to Nonlinear Models." *Political Analysis*, **7**, 189–203.

Shachar R, Nalebuff B (2004). "Verifying the Solution from a Nonlinear Solver: A Case Study: Comment." *American Economic Review*, **94**(1), 382–390.

Stokes H (2004). "On the Advantage of Using Two or More Econometric Software Systems to Solve the Same Problem." *Journal of Economic and Social Measurement*, **29**(1-3), 307–320.

Vose MD (1993). "Modeling Simple Genetic Algorithms." In LD Whitley (ed.), *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Mateo, CA.

Wand J, King G, Lau O (2008). ***anchors**: Software for Anchoring Vignette Data*. R package version 2.0, URL http://wand.stanford.edu/anchors/.

Wand J, King G, Lau O (2011). "**anchors**: Software for Anchoring Vignette Data." *Journal of Statistical Software*, **42**(3), 1–25. URL http://www.jstatsoft.org/v42/i03/.

Yao X, Liu Y, Lin G (1999). "Evolutionary Programming Made Faster." *IEEE Transactions on Evolutionary Computation*, **3**(2), 82–102.

Affiliation:

Walter R. Mebane, Jr.
Department of Political Science
Department of Statistics
University of Michigan
Ann Arbor, MI 48109-1045, United States of America
E-mail: wmebane@umich.edu
URL: http://www.umich.edu/~wmebane

Jasjeet S. Sekhon
Department of Political Science
210 Barrows Hall #1950
UC Berkeley
Berkeley, CA 94720-1950, United States of America
E-mail: sekhon@berkeley.edu
URL: http://sekhon.berkeley.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 42, Issue 11 *Submitted:* 2007-02-11
June 2011 *Accepted:* 2007-11-21