BB: An R Package for Solving a Large System of Nonlinear Equations and for Optimizing a High-Dimensional Nonlinear Objective Function

Ravi Varadhan
Johns Hopkins University

Paul D. Gilbert
Bank of Canada

Abstract

*This introduction to the R package **BB** is a (slightly) modified version of Varadhan and Gilbert (2009), published in the Journal of Statistical Software.*

We discuss R package **BB**, in particular, its capabilities for solving a nonlinear system of equations. The function `BBsolve` in **BB** can be used for this purpose. We demonstrate the utility of these functions for solving: (a) large systems of nonlinear equations, (b) smooth, nonlinear estimating equations in statistical modeling, and (c) non-smooth estimating equations arising in rank-based regression modeling of censored failure time data. The function `BBoptim` can be used to solve smooth, box-constrained optimization problems. A main strength of **BB** is that, due to its low memory and storage requirements, it is ideally suited for solving high-dimensional problems with thousands of variables.

Keywords: accelerate failure time model, Barzilai-Borwein, derivative-free, estimating equations, large-scale optimization, non-monotone line search, non-smooth optimization, rank-based regression.

1. Introduction

R (R Development Core Team 2009) package **BB** provides functions for solving large-scale nonlinear problems. **BB** (version 2009.6-1) comprises six functions, which are nested at three levels. At the bottom level are three functions: `sane` and `dfsane` for solving a nonlinear system of equations; and `spg` for optimizing a nonlinear objective function with box constraints. These functions, especially `dfsane` and `spg`, are the workhorses of **BB**. The functions `BBsolve` and `BBoptim` are at the next higher level. `BBsolve` is a wrapper for `dfsane`. It takes a single parameter vector as starting value and calls `dfsane` repeatedly with different algorithm control parameters to try and achieve successful convergence to the solution. Similarly, `BBoptim`, which is a wrapper for `spg`, takes a single parameter vector as starting value and calls `spg` repeatedly with different algorithm control parameters. At the top-most level is the function `multiStart`. This takes a matrix of parameters as multiple starting values and, depending on the value of the argument `action` specified by the user, calls either `BBsolve` or `BBoptim` for each starting value.

The main purposes of this article are: (1) to introduce **BB** to R users, (2) to present background necessary for the appropriate use of the algorithms, and (3) to demonstrate the utility of the algorithms by presenting results on a variety of test problems. In addition

to **BB**, R package **nleqslv** (Hasselman 2009) has recently been added to the Comprehensive R Archive Network (CRAN) https://CRAN.R-project.org/. However, **nleqslv** uses Newton-type methods, and hence it may not be suitable for solving large systems of equations. We will confine this article to the problem of finding a root of simultaneous nonlinear equations, and not discuss `spg` for nonlinear optimization, since that problem can be addressed using existing R functions including `optim`, `nlminb`, and `nlm`. Other optimization packages are summarized in the CRAN task view (Zeileis 2005) on optimization at https://CRAN.R-project.org/view=Optimization. However, we point out that the optimization function `spg` in **BB** is different from the existing functions in that it is well suited to large-scale optimization, since it does not require the Hessian matrix of the objective function. It is based on the Barzilai-Borwein gradient method developed by Raydan (1997). Our R implementation (which is based on the Fortran code of Birgin, Martínez, and Raydan 2001) is competitive with the limited-memory BFGS algorithm (`method = "L-BFGS-B"`) in `optim` for large-scale, box-constrained optimization, and is superior to the conjugate gradient methods (`method = "CG"`) in `optim` for large-scale, unconstrained optimization. Results presented here were obtained with **BB** version 2009-6.1. The most recent version of package **BB** is available from CRAN at https://CRAN.R-project.org/package=BB. A tutorial on **BB** is available and can be viewed from an R session by typing:

```
R> vignette("BB", package = "BB")
```

Also, a version of this paper, augmented with results from the system on which the vignette is compiled, can be viewed by typing:

```
R> vignette("BBvignetteJSS", package = "BB")
```

The JSS paper was compiled with settings for the number of simulations and bootstrap samples as `nsim=1000, nboot=500`, but, in order to maintain a reasonable build time for the package, these values are set very much smaller in the vignette (nsim=10, nboot=50).

```
R> nsim  <- 10 # 1000
R> nboot <- 50 # 500
```

2. Solving nonlinear system of equations

We are interested in solving the nonlinear system of equation

$$F(x) = 0, \tag{1}$$

where $F : \mathbb{R}^p \mapsto \mathbb{R}^p$ is a nonlinear function with continuous partial derivatives. We are interested in situations where p is large, and where the Jacobian of F is either unavailable or requires a prohibitive amount of storage, although the algorithms in **BB** are also applicable when p is small. The best known methods for solving (1) are Newton's method and the quasi-Newton's methods (Ortega and Rheinboldt 1970; Dennis and Schnabel 1983). Newton's method employs a working linear approximation to $F(x)$ around an estimate of the solution, and improves it in an iterative manner:

$$x_{k+1} = x_k \; - \; J(x_k)^{-1} \, F(x_k),$$

where $J : \mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}^p$ is the Jacobian of F evaluated at x_k. Quasi-Newton methods use an approximation of J, which, along with the solution vector, is updated at each iteration. For example, the classical Broyden's ("good") method is given by the equations:

$$\begin{aligned} x_{k+1} &= x_k - {B_k}^{-1} F(x_k); \\ B_{k+1} &= B_k + \frac{F(x_{k+1})\,(x_{k+1} - x_k)^\top}{(x_{k+1} - x_k)^\top (x_{k+1} - x_k)}, \end{aligned}$$

where B_0 is usually the identity matrix. These methods are attractive because they converge rapidly from any good starting value. However, they need to solve a linear system of equations using the Jacobian or an approximation of it at each iteration, which can be prohibitively expensive for large p problems.

An indirect approach to solving Equation 1 is to transform it to a standard optimization problem by defining a merit function $\phi(u)$ where $\phi : \mathbb{R}^p \mapsto \mathbb{R}$ is a functional with a unique global minimum at $u = 0$. Now, any solution of $F(x) = 0$ is also a minimizer of $\phi(F(x))$, but the converse does not always hold. A sufficient condition for the converse to hold is that the Jacobian of F be non-singular at the minimizer of $\phi(u)$ (Ortega and Rheinboldt 1970). A commonly used merit function is the L_2-norm of F: $\phi(F(x)) = \|F(x)\|$, which is also known as the "residual". This approach generally does not work well in practice, and hence is little used as a stand-alone method for solving nonlinear systems. However, as discussed later, we shall use this approach for generating good starting values for the spectral algorithms.

2.1. Spectral method for nonlinear systems

Recently, two efficient algorithms, SANE and DF-SANE, for solving large-scale nonlinear systems of equations have been proposed in the numerical analysis literature by Raydan and his colleagues (SANE: La Cruz and Raydan 2003; DF-SANE: La Cruz, Martínez, and Raydan 2006). These methods are an extension of the Barzilai-Borwein method for finding local minimum (Barzilai and Borwein 1988; Raydan 1997). They use $\pm F(x)$ as search directions in a systematic way, with one of the spectral coefficients as steplength, and a non-monotone line-search technique for global convergence. This provides a robust scheme for solving nonlinear systems. The simplicity of search direction and steplength results in low-cost per iteration.

The spectral approach for nonlinear systems is defined by the following iteration:

$$x_{k+1} = x_k + \alpha_k\, d_k;\ k = 0, 1, 2, \ldots \tag{2}$$

where α_k is the spectral steplength, and d_k is the search direction, which is defined as follows.

$$d_k = \begin{cases} -F(x_k) & : \text{ for DF-SANE,} \\ \pm F(x_k) & : \text{ for SANE} \end{cases}$$

For the SANE algorithm, the sign associated with $F(x_k)$ is that which yields a descent direction with respect to the merit function $\|F(x_k)\|^2$. The only spectral steplength considered in La Cruz and Raydan (2003) and La Cruz *et al.* (2006) is:

$$\alpha_k = \frac{s_{k-1}^\top\, s_{k-1}}{s_{k-1}^\top\, y_{k-1}}; k = 1, 2, \ldots, \tag{3}$$

where $s_{k-1} = x_k - x_{k-1}$, and $y_{k-1} = F(x_k) - F(x_{k-1})$. Below, and in the tables, we denote the SANE and DF-SANE algorithms that use this steplength as *sane-1* and *dfsane-1*, respectively. In addition to Equation 3, Barzilai and Borwein (1988) proposed a second spectral steplength:

$$\alpha_k = \frac{s_{k-1}^\top y_{k-1}}{y_{k-1}^\top y_{k-1}}. \tag{4}$$

We denote the SANE and DF-SANE algorithms that use this steplength as *sane-2* and *dfsane-2*, respectively. We also consider a third spectral steplength, first proposed in Varadhan and Roland (2008) for the acceleration of EM algorithms:

$$\alpha_k = \text{sgn}(s_{k-1}^\top y_{k-1}) \frac{\|s_{k-1}\|}{\|y_{k-1}\|}, \tag{5}$$

where $\text{sgn}(x) = x/|x|$, when $x \neq 0$, and is zero when $x = 0$. For all three steplengths, we define $\alpha_0 = \min(1, 1/\|F(x_0)\|)$. The general effectiveness of spectral steplengths is due to the fact that they can be viewed as a Rayleigh quotient with respect to a secant approximation of the Jacobian. The scalar $|\alpha_k|$ is closely related to the condition number of the Jacobian J_k (Fletcher 2001).

2.2. Globalization using non-monotone line search

To achieve global convergence, the spectral iterative scheme (2) must be combined with a suitable line search technique. For SANE, La Cruz and Raydan (2003) consider a non-monotone line search technique (Grippo, Lampariello, and Lucidi 1986), which can be written as

$$f(x_{k+1}) \leq \max_{0 \leq j \leq M} f(x_{k-j}) \; + \; \gamma \alpha_k \nabla f(x_k)^\top d_k, \tag{6}$$

where the merit function $f(x) = F(x)^\top F(x)$, and γ is a small positive number (we choose $\gamma = 10^{-4}$). In the above condition, denoted here as the GLL condition, M is a positive integer that plays an important role in dictating the allowable degree of non-monotonicity in the value of the merit function, with $M = 0$ yielding a strictly monotone scheme. As pointed out by Fletcher (2001), the Barzilai-Borwein schemes perform poorly when strict monotonicity is imposed, especially in ill-conditioned problems. They perform better when some amount of non-monotonicity is allowed, hence globalization using the GLL condition, with values of M between 5-20. The term $\nabla f(x_k)^\top d_k$ in SANE is equal to $\pm F_k^\top J_k F_k$, where $F_k = F(x_k)$ and J_k is the Jacobian of F at x_k. This can be evaluated without computing the Jacobian as follows:

$$F_k^\top J_k F_k \approx F_k^\top \left[\frac{F(x_k + hF_k) - F_k}{h} \right],$$

where $h = 10^{-7}$.

For DF-SANE (stands for "derivative-free SANE"), La Cruz *et al.* (2006) propose a new, and different globalization line search technique:

$$f(x_{k+1}) \leq \max_{0 \leq j \leq M} f(x_{k-j}) \; + \; \eta_k - \gamma \alpha_k^2 f(x_k), \tag{7}$$

where $\gamma = 10^{-4}$, and $\eta_k > 0$ decreases with k such that $\sum_{k=0}^{\infty} \eta_k = \eta < \infty$. Note that this strategy does not involve any Jacobian computations. Hence the phrase "derivative-free".

This strategy maintains the non-monotonicity of GLL, while avoiding the quadratic product involving the Jacobian, which entails an additional function evaluation at each iteration. Consequently, DF-SANE is generally more economical than SANE in terms of number of evaluations of F. The presence of $\eta_k > 0$ ensures that all the iterations are well-defined, and the forcing term $-\gamma\alpha_k^2 f(x_k)$ provides the theoretical condition sufficient for establishing global convergence (La Cruz *et al.* (2006)).

2.3. Implementations of SANE and DF-SANE in BB

For detailed algorithmic implementation of the iterations and non-monotone line searches for SANE and DF-SANE, the reader is directed to La Cruz and Raydan (2003) and La Cruz *et al.* (2006), respectively. Also see the documentation for the R functions `sane` and `dfsane` in **BB** for more details. Here we only discuss the salient features of our R implementation for SANE and DF-SANE algorithms in the package **BB** that are different from the original Fortran codes (which can be obtained from Raydan 2009). These are:

1. We provide an option for three spectral steplengths, Equations 3, 4 and 5. The `method` argument in `sane` and `dfsane` functions can be used to select between these steplengths. The original SANE and DF-SANE algorithms only allowed one steplength, Equation 3, which can be selected with `method=1`. We have set `method=2`, which corresponds to Equation 4, as the default. In our numerical experiments, this generally outperformed the other two methods. (See Table 1 discussed in the next section for results.)

2. We re-scale the first BB steplength as: $\alpha_0 = \min(1, 1/\|F(x_0)\|)$, whereas in the original implementation $\alpha_0 = 1$.

3. We provide an option for improving on starting values, when the user is unable to generate good starting values. We do this by calling the Nelder-Mead nonlinear simplex algorithm (Nelder and Mead 1965), as implemented in the R function `optim`, with the merit function $f(x)$ as the objective function.

4. We provide an option for improving upon convergence when `sane` or `dfsane` terminates unsuccessfully in some particular manner, i.e. when `convergence = 4` or `5` for `sane` and when `convergence = 2 or 5` for `dfsane`. We do this by calling the limited memory BFGS algorithm (`method="L-BFGS-B"`) in `optim` with the merit function $f(x)$ as the objective function.

5. When we are close to the solution, i.e. when $f(x_k) < 10^{-4}$, we use the dynamic retard strategy proposed in Luengo and Raydan (2003):

$$x_{k+1} = x_k + \alpha_{k-1}\, d_k,$$

   i.e. we use the spectral steplength from two iterations before the current one. This retarded spectral scheme was never worse than the unretarded spectral method (Eq. 2) in our experiments, and in many cases it actually exhibited faster convergence (results not shown).

6. We implement an additional stopping criterion in our R functions. The iterations are terminated when there is no decrease in the merit function $f(x)$ over `noimp` iterations,

where we choose a default value of `noimp = 100`. This is particularly essential when a large M, say, $M \geq 100$ is used.

2.4. What to do when the algorithm fails? – Function `BBsolve`

Algorithm `dfsane` or (`sane`) is said to have failed when a non-zero convergence type is obtained, i.e. when `convergence` > 0. In this case, we have found that the following sequential strategy generally works quite well:

1. Try a different non-monotonicity parameter `M`. Since the default is `M = 10`, try `M=50`.
2. Try a different method. Since the default is `method = 2`, try methods 1 and 3.
3. Try with Nelder-Mead initialization `NM`. Since the default is `NM = FALSE`, the user should try `NM = TRUE`.

We have written an R wrapper function called `BBsolve` to automatically implement this strategy. We have found this function to be successful in problems where `dfsane` and `sane` had failed to converge. Here we give a simple example to illustrate this using the Freudenstein-Roth function.

```
R> require("BB")
R> froth <- function(p){
+    r <- rep(NA, length(p))
+    r[1] <- -13 + p[1] + (p[2] * (5 - p[2]) - 2)  * p[2]
+    r[2] <- -29 + p[1] + (p[2] * (1 + p[2]) - 14) * p[2]
+    r
+    }
R> p0 <- rep(0, 2)
R> dfsane(par = p0, fn = froth, control = list(trace = FALSE))

$par
[1] -4.990589 -1.448398

$residual
[1] 10.42073

$fn.reduction
[1] 17.04337

$feval
[1] 131

$iter
[1] 106

$convergence
```

```
[1] 5

$message
[1] "Lack of improvement in objective function"

R> sane(par = p0, fn = froth, control = list(trace = FALSE))

$par
[1] -5.729871 -1.702569

$residual
[1] 9.592763

$fn.reduction
[1] 18.21428

$feval
[1] 417

$iter
[1] 127

$convergence
[1] 5

$message
[1] "Lack of improvement in objective function"

R> BBsolve(par = p0, fn = froth)

  Successful convergence.
$par
[1] 5 4

$residual
[1] 2.012452e-09

$fn.reduction
[1] 6.998875

$feval
[1] 1109

$iter
[1] 233

$convergence
```

```
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M     NM
     1     50      1
```

Function `dfsane` and `sane` fail to converge, while `BBsolve` converges successfully. A similar wrapper function called `BBoptim` can be used to solve optimization problems when `spg` fails to converge.

3. Numerical Experiments

3.1. Standard test problems

We have tested our algorithms extensively on a number of nonlinear systems considered in La Cruz and Raydan (2003), La Cruz *et al.* (2006), and Luksan and Vlcek (2003). Here we report the results for six problems, whose statements are given in Apppendix A. We tested four methods, `sane` and `dfsane`, each with two steplengths Equations 3 and 4, for 1000 randomly generated initial values for each problem, which are also provided in Appendix A. This approach of using random starting values is uncommon in the numerical analysis literature when testing new methods, and when comparing methods. Rather, a single, reasonably good starting value is used in each test problem. Hence, our tests are much more stringent than those commonly seen in the numerical analysis literature (e.g. La Cruz and Raydan 2003, La Cruz *et al.* 2006). Therefore, it should not come as a surprise that there are substantial number of convergence failures in some problems. We used $\frac{\|F(x_n)\|}{\sqrt{p}} \leq 10^{-7}$, where p is the dimensionality of the problem, as the stopping criterion. We have successful convergence (i.e. `convergence = 0`) when this criterion is satisfied. The algorithm (`sane` or `dfsane`) is said to have failed when `convergence > 0`.

We chose $p = 500$ for all the 6 test problems. Unless otherwise stated explicitly, we used the default control parameter setting for all the parameters of `dfsane` and `sane`. The numerical experiment results presented here were performed using R version 2.9.1 running on a Microsoft Windows Vista operating system, with a 2.2 GHz Intel Dual-core Pentium processor and 4 GB of RAM. The results are presented in Table 1.

In order to reproduce the random numbers used in this paper, the seed and RNG types are set to known values.

```
R> require("setRNG")
R> test.rng <- list(kind = "Mersenne-Twister", normal.kind = "Inversion",
+                   seed = 1234)
R> old.seed <- setRNG(test.rng)
```

Iterative numerical procedures can be sensitive to system math libraries and even hardware floating point calculation, since a very small difference in a search steps will result in slightly

different paths to the solution. This can result in a different number of iterations and/or a slightly different answer. The difference may be especially aggravated in problems where the objective function is nearly "flat" near the solution. We have run the examples here with different versions of R and on different hardware and operating systems and the results are relatively similar, but users replicating the results may see small differences.

We define a "failure" as the inability of an algorithm to achieve the default tolerance of $1.e-07$ under default values for all the control parameters. It might be possible that a different control setting enables successful convergence. In fact, this is one of the main motivations for creating the `BBsolve` function that can automatically try different control settings to achieve successful convergence.

Algorithms *sane-2* and *dfsane-2* performed better (using steplength Equation 4) than *sane-1* and *dfsane-1* (with steplength Equation 3), except for the *extended Rosenbrock function*. *dfsane-2* was the best method overall. We re-ran the tests with `BBsolve` for the two problems: *exponential function 3* and *extended Rosenbrock*, where even the best performing method had a substantial number of convergence failures. Now, `BBsolve` converged successfully in the *extended Rosenbrock* problem for all 1000 starting values, and had only one failure in the *exponential function 3* problem. This demonstrates that `BBsolve` is a reliable tool for solving a nonlinear system of equations.

3.2. Finding multiple roots or multiple local optima – Function `multiStart`

It is not uncommon for a nonlinear system of equations to have multiple roots or for a nonlinear objective function to have multiple local minima (or maxima). In this case, it may be of interest to identify as many, if not all, solutions as possible. To this end, we have provided a function called `multiStart`, which can accept a matrix of parameter values, where each row of the matrix is a starting value. The user needs to define this matrix and pass it as an input to `multiStart`. Two widely used approaches are: (1) generate random numbers according to some probability distribution, and (2) regular grid search. This function has an argument called `action`, which indicates whether the user wants to *solve* a nonlinear system of equations or to *optimize* a nonlinear objective function. For each starting value, `multiStart` calls either `BBsolve` or `BBoptim`.

We now illustrate how to use `multiStart` to find multiple roots. We consider a system of high-degree polynomial equations (Kearfott 1987), comprising 3 equations in 3 variables. It has 12 real roots and 126 complex roots. Here we find all the 12 roots.

We generate 300 random starting values, each a vector of length equal to 3. The system is then solved 300 times and the unique solutions were picked out.

```
R> hdp <- function(x) {
+    r <- rep(NA, length(x))
+    r[1] <- 5*x[1]^9 - 6*x[1]^5 * x[2]^2 + x[1] * x[2]^4 + 2*x[1] * x[3]
+    r[2] <- -2 * x[1]^6 * x[2] + 2 * x[1]^2 * x[2]^3 + 2 * x[2] * x[3]
+    r[3] <- x[1]^2 + x[2]^2 - 0.265625
+    r
+    }
R> old.seed <- setRNG(test.rng)
R> p0 <- matrix(runif(900), 300, 3)
```

Methods	# Iters	# Fevals	CPU (sec)	# Failures
		1. Exponential function 3		
sane-1	147 (50, 92)	630 (131, 275)	0.30 (0.06, 0.14)	427
dfsane-1	231 (152, 271)	551 (283, 490)	0.31 (0.17, 0.28)	428
sane-2	115 (99, 130)	252 (208, 279)	0.13 (0.11, 0.14)	57
dfsane-2	210 (175, 237)	227 (183, 256)	0.15 (0.12, 0.17)	7
`BBsolve`	212 (175, 235)	229 (183, 254)	0.15 (0.12, 0.17)	1
		2. Trigexp function		
sane-1	33 (24, 27)	72 (49, 55)	0.08 (0.05, 0.06)	6
dfsane-1	29 (24, 28)	30 (25, 29)	0.04 (0.03, 0.04)	0
sane-2	37 (24, 28)	76 (49, 57)	0.08 (0.05, 0.07)	0
dfsane-2	31 (24, 28)	32 (25, 29)	0.04 (0.03, 0.05)	0
		3. Broyden's tridiagonal function		
sane-1	19 (19, 19)	39 (39, 39)	0.02 (0.01, 0.02)	0
dfsane-1	19 (19, 19)	20 (20, 20)	0.01 (0.00, 0.02)	0
sane-2	20 (20, 20)	41 (41, 41)	0.02 (0.01, 0.02)	0
dfsane-2	20 (20, 20)	21 (21, 21)	0.01 (0.00, 0.02)	0
		4. Extended Rosenbrock function		
sane-1	41 (35, 41)	91 (73, 86)	0.03 (0.03, 0.03)	30
dfsane-1	43 (35, 42)	61 (39, 50)	0.03 (0.01, 0.03)	39
sane-2	80 (39, 120)	174 (80, 247)	0.07 (0.03, 0.10)	484
dfsane-2	61 (38, 60)	66 (42, 68)	0.04 (0.02, 0.04)	158
`BBsolve`	40 (37, 43)	42 (39, 45)	0.02 (0.02, 0.03)	0
		5. Troesch function		
sane-1	1501 (1501, 1501)	6068 (6026, 6107)	3.29 (3.21, 3.28)	1000
dfsane-1	1481 (1501, 1501)	4005 (3936, 4192)	2.43 (2.37, 2.53)	949
sane-2	803 (673, 904)	2067 (1722, 2338)	1.17 (0.97, 1.33)	1
dfsane-2	907 (763, 1033)	1391 (1169, 1580)	0.93 (0.78, 1.06)	1
		6. Chandrasekhar's H-equation		
sane-1	14 (14, 14)	29 (29, 29)	2.15 (2.08, 2.21)	0
dfsane-1	14 (14, 14)	15 (15, 15)	2.15 (2.08, 2.21)	0
sane-2	13 (13, 13)	27 (27, 27)	2.15 (2.08, 2.21)	0
dfsane-2	13 (13, 13)	14 (14, 14)	2.15 (2.08, 2.21)	0

Table 1: Results of numerical experiments for 6 standard test problems. 1000 randomly generated starting values were used for each problem. Means and inter-quartile ranges (in parentheses) are shown. Default control parameters were used in all the algorithms.

```
R> ans <- multiStart(par = p0, fn = hdp, action = "solve")

R> sum(ans$conv)

[1] 294

R> pmat <- ans$par[ans$conv, ]
R> ord1 <- order(pmat[, 1])
```

table1										
sane-1	358.700	38.750	177.250	1700.300	152.750	672.000	0.056	0.005	0.022	8.000
dfsane-1	226.500	189.750	262.000	418.500	342.500	436.750	0.016	0.014	0.018	5.000
sane-2	120.300	107.250	128.750	259.900	240.500	283.750	0.009	0.008	0.009	1.000
dfsane-2	253.100	185.750	298.750	276.000	206.000	319.750	0.012	0.008	0.014	0.000
BBsolve	230.800	185.750	253.000	250.000	206.000	272.750	0.011	0.009	0.012	0.000
sane-1	27.600	26.250	27.750	56.600	53.500	57.500	0.005	0.003	0.004	0.000
dfsane-1	32.300	26.250	30.000	35.100	27.250	31.750	0.003	0.002	0.003	0.000
sane-2	72.300	27.000	113.000	145.700	55.000	227.250	0.010	0.004	0.015	0.000
dfsane-2	43.700	27.000	59.500	44.700	28.000	60.500	0.004	0.002	0.005	0.000
sane-1	19.100	19.000	19.000	39.200	39.000	39.000	0.003	0.001	0.002	0.000
dfsane-1	19.100	19.000	19.000	20.100	20.000	20.000	0.001	0.001	0.001	0.000
sane-2	20.100	20.000	20.000	41.200	41.000	41.000	0.001	0.001	0.001	0.000
dfsane-2	20.100	20.000	20.000	21.100	21.000	21.000	0.001	0.001	0.001	0.000
sane-1	38.500	37.000	38.750	79.900	77.000	80.750	0.003	0.002	0.003	0.000
dfsane-1	37.200	33.750	38.750	42.600	39.250	46.750	0.002	0.002	0.002	0.000
sane-2	54.600	37.000	44.250	111.900	75.000	91.750	0.003	0.002	0.003	2.000
dfsane-2	39.300	36.250	42.000	44.200	39.500	45.500	0.002	0.002	0.002	0.000
BBsolve	39.300	37.000	41.500	41.300	39.000	43.500	0.002	0.002	0.002	0.000
sane-1	1501.000	1501.000	1501.000	6095.300	6052.750	6133.750	0.274	0.268	0.272	10.000
dfsane-1	1501.000	1501.000	1501.000	3979.200	3840.000	4056.000	0.193	0.187	0.195	10.000
sane-2	809.900	707.500	889.000	2079.900	1782.000	2315.500	0.092	0.079	0.101	0.000
dfsane-2	810.700	746.750	890.750	1235.100	1107.750	1378.250	0.063	0.058	0.070	0.000
sane-1	14.000	14.000	14.000	29.000	29.000	29.000	0.100	0.094	0.099	0.000
dfsane-1	14.000	14.000	14.000	15.000	15.000	15.000	0.100	0.094	0.099	0.000
sane-2	13.000	13.000	13.000	27.000	27.000	27.000	0.100	0.094	0.099	0.000
dfsane-2	13.000	13.000	13.000	14.000	14.000	14.000	0.100	0.094	0.099	0.000

Table 2: Results of numerical experiments for 6 standard test problems. 10 randomly generated starting values were used for each problem. Means and inter-quartile ranges (in parentheses) are shown. Default control parameters were used in all the algorithms.

```
R> ans <- round(pmat[ord1, ], 4)
R> ans[!duplicated(ans), ]
```

```
         [,1]    [,2]    [,3]
 [1,] -0.5154  0.0000 -0.0124
 [2,] -0.4670 -0.2181  0.0000
 [3,] -0.4670  0.2181  0.0000
 [4,] -0.2799  0.4328 -0.0142
 [5,] -0.2799 -0.4328 -0.0142
 [6,]  0.0000 -0.5154  0.0000
 [7,]  0.0000  0.5154  0.0000
 [8,]  0.2799  0.4328 -0.0142
 [9,]  0.2799 -0.4328 -0.0142
[10,]  0.4670  0.2181  0.0000
[11,]  0.4670 -0.2181  0.0000
[12,]  0.5154  0.0000 -0.0124
```

The `sum(ans$conv)` gives the number of successful runs (284 in our experiments). `pmat` are the converged solutions and `ans[!duplicated(ans), ]` displays the 12 unique solutions.

4. Solving nonlinear estimating equations in statistics

Nonlinear system of equations arise commonly in statistics. In some cases, there will be a naturally associated scalar function of parameters, which can be optimized to obtain parameter estimates. For example, maximum likelihood estimates can be obtained by solving the score equations, even though in general it is better to obtain parameter estimates by directly maximizing the log-likelihood. In other cases, there may not be a natural scalar function associated with the nonlinear system, and we need to solve the system of equations to obtain parameter estimates. This includes a broad class of statistical estimation problems under the heading of estimating functions or estimating equations, where a probability distribution for the data generating process is not explicitly postulated, but only weaker conditions such as unbiasedness and information unbiasedness are imposed on the estimating function (Small and Wang 2003). Well known examples are: generalized least squares (Carroll and Ruppert 1988), generalized estimating equations (Diggle, Heagerty, Liang, and Zeger 2002), and semi-parametric accelerated failure time models in survival analysis (Kalbfleisch and Prentice 2002). Here we consider two examples with simulated data, and one with real data. Our goal is to show the utility of **BB** for solving nonlinear estimating equations.

4.1. Poisson regression with offset

Poisson regression is commonly used to model data in the form of counts, i.e. number of times a particular event occurred over some known period of time. We consider data of the form $(Y_i, t_i, X_i) : i = 1, \cdots, n$, where Y_i are the counts over an observation period t_i, and X_i are the corresponding covariates. Estimating equations for poisson regression of count data, with offset, can be written as:

$$\sum_{i=1}^{n} X_i^\top \left\{ Y_i \, - \, t_i \, e^{X_i^\top \beta} \right\} \, = \, 0. \tag{8}$$

We consider a simulation problem with $n = 500$, and $p = 8$. We set $\beta = (-5, 0.04, 0.3, 0.05, 0.3, -0.005, 0.1, -0.4)$, and generate data from a poisson distribution: $Y_i \,|\, t_i, X_i \sim \text{poisson}(t_i X_i^\top \beta)$, where $t_i \sim N(\mu = 100, \sigma = 30)$, and the covariates X_i are generated according to the following R code. This problem can be readily solved using the `glm` function in R, by specifying the `offset` option. However, we show that it can also be directly solved by solving the estimating equations Eq. 8, which are nothing but the score equations of the Poisson likelihood. Parameter estimates from `dfsane` are identical to that from `glm`.

```
R> U.eqn <- function(beta) {
+        Xb <- c(X %*% beta)
+        c(crossprod(X,  Y - (obs.period * exp(Xb))))
+        }
R> poisson.sim <- function(beta, X, obs.period) {
+        Xb <- c(X %*% beta)
+        mean <- exp(Xb) * obs.period
+        rpois(nrow(X), lambda = mean)
+        }
R> old.seed <- setRNG(test.rng)
R> n <- 500
R> X <- matrix(NA, n, 8)
R> X[,1] <- rep(1, n)
R> X[,3] <- rbinom(n, 1, prob=0.5)
R> X[,5] <- rbinom(n, 1, prob=0.4)
R> X[,7] <- rbinom(n, 1, prob=0.4)
R> X[,8] <- rbinom(n, 1, prob=0.2)
R> X[,2] <- rexp(n, rate = 1/10)
R> X[,4] <- rexp(n, rate = 1/10)
R> X[,6] <- rnorm(n, mean = 10, sd = 2)
R> obs.period <- rnorm(n, mean = 100, sd = 30)
R> beta <- c(-5, 0.04, 0.3,  0.05, 0.3, -0.005, 0.1, -0.4)
R> Y <- poisson.sim(beta, X, obs.period)
R> res <- dfsane(par = rep(0,8), fn = U.eqn,
+       control = list(NM = TRUE, M = 100, trace = FALSE))
R> res

$par
[1] -5.015722742  0.042448236  0.308251807  0.049251745  0.318457820
[6] -0.005503549  0.074734709 -0.461351611

$residual
[1] 9.256209e-08

$fn.reduction
[1] 9140.99

$feval
[1] 1055
```

```
$iter
[1] 882

$convergence
[1] 0

$message
[1] "Successful convergence"
```

```
R> glm(Y ~ X[,-1], offset = log(obs.period),
+           family = poisson(link = "log"))

Call:  glm(formula = Y ~ X[, -1], family = poisson(link = "log"), offset = log(obs.period)

Coefficients:
(Intercept)     X[, -1]1     X[, -1]2     X[, -1]3     X[, -1]4     X[, -1]5
  -5.015723     0.042448     0.308252     0.049252     0.318458    -0.005504
   X[, -1]6     X[, -1]7
   0.074735    -0.461352

Degrees of Freedom: 499 Total (i.e. Null);  492 Residual
Null Deviance:            2170
Residual Deviance: 519.5         AIC: 1664
```

The last command shows that `glm` gives the same result.

4.2. Rank-based regression using accelerated failure time model

Accelerated failure time (AFT) model is a useful alternative to the popular Cox relative risk model for the analysis of failure time data subject to censoring. The AFT model relates the logarithm of the failure time to a linear function of the covariates, and hence the model has direct physical interpretation in terms of the failure time. Let T_i be the failure time, and $X_i \in \mathbb{R}^p$ be the corresponding covariates for the ith individual ($i = 1, \ldots, n$). The semi-parametric AFT model may be written as:

$$\log T_i \;=\; X_i^\top \beta \,+\, \epsilon_i; \quad (i = 1, \ldots, n),$$

where $\beta \in \mathbb{R}^p$ is a vector of regression parameters to be estimated from the data, and ϵ_i are independent errors with a common, but unspecified, probability distribution. Let C_i be the censoring time for ith individual. It is usually assumed that C_i is independent of T_i, given X_i. Let $T_i^* = \min(T_i, C_i)$ and $\delta_i = I(T_i \le C_i)$, where $I(.)$ is the usual indicator function. The data then comprises (T_i^*, δ_i, X_i). The regression parameters β are estimated by solving the weighted log-rank estimating function (Jin, Lin, Wei, and Ying 2003):

$$U(\beta) = \sum_{i=1}^{n} \delta_i \, \phi_i \left\{ X_i - \frac{\sum_{j=1}^n X_j \, I(T_j^* - X_j^\top \beta \ge T_i^* - X_i^\top \beta)}{\sum_{j=1}^n I(T_j^* - X_j^\top \beta \ge T_i^* - X_i^\top \beta)} \right\} \;=\; 0, \tag{9}$$

where ϕ_i is a possibly data-dependent weight function. The choice of $\phi_i = 1$ yields the log-rank estimator, and $\phi_i = n^{-1} \sum_{j=1}^{n} I(T_j^* - X_j^\top \beta \geq T_i^* - X_i^\top \beta)$ yields the Gehan estimator.

In spite of the theoretical advances, semiparametric methods for the AFT model have been seldom used in practice, mainly because of the lack of efficient and reliable computational methods (Jin *et al.* 2003). One main difficulty is that the system of semiparametric estimating functions, (9), involves step functions of the regression parameters. Therefore, conventional numerical techniques, which depend essentially on the smoothness of the functions, cannot be used. Lin and Geyer (1992) proposed simulated annealing, but their algorithm is not guaranteed to find the true minimum. Jin *et al.* (2003) proposed an iterative estimator that converts the solution of (9) into a sequence of minimization problems, which can be solved using linear programming techniques. Here we take a more direct approach by directly solving (9) using the DF-SANE algorithm, which does not involve any derivatives.

We first consider a simulation problem with $n = 1000$, and $p = 8$. We randomly generated a 1000×8 matrix of binary and continuous covariates (see the code below for details of simulation). We set $\beta = (0.5, -0.4, 0.3, -0.2, -0.1, 0.4, 0.1, -0.6)$. We generated independent errors ϵ_i from a log-normal distribution with mean = 1 and variance = 1. Censoring times C_i were generated from a uniform distribution so as to obtain close to 20% censoring. We ran 1000 simulations, with a fixed covariate matrix X, but generating new T^* and δ in each simulation. For each simulated data set, we used the same starting value $\beta_0 =$ `rep(0, 8)` in `dfsane` to find a root of (9). The function `aft.eqn` computes (9).

```
R> aft.eqn <- function (beta, X, Y, delta, weights = "logrank") {
+        deltaF <- delta == 1
+        Y.zeta <- Y - c(X %*% beta)
+        ind <- order(Y.zeta, decreasing = TRUE)
+        dd <- deltaF[ind]
+        n <- length(Y.zeta)
+        tmp <- apply(X[ind, ], 2, function (x) cumsum(x))
+
+        if (weights == "logrank") {
+           c1 <- colSums(X[deltaF, ])
+           r <- (c1 - colSums(tmp[dd, ] / (1:n)[dd])) / sqrt(n)
+           }
+
+        if (weights == "gehan") {
+           c1 <- colSums(X[deltaF, ]* ((1:n)[order(ind)][deltaF]))
+           r <- (c1 - colSums(tmp[dd, ])) / ( n * sqrt(n))
+           }
+    r
+    }
R> old.seed <- setRNG(test.rng)
R> n <- 1000
R> X <- matrix(NA, n, 8)
R> X[,1] <- rbinom(n, 1, prob=0.5)
R> X[,2] <- rbinom(n, 1, prob=0.4)
R> X[,3] <- rbinom(n, 1, prob=0.4)
```

```
R> X[,4] <- rbinom(n, 1, prob=0.3)
R> temp <- as.factor(sample(c("0", "1", "2"), size=n, rep=T,
+                       prob=c(1/3,1/3,1/3)))
R> X[,5] <- temp == "1"
R> X[,6] <- temp == "2"
R> X[,7] <- rexp(n, rate=1/10)
R> X[,8] <- rnorm(n)
R> eta.true <- c(0.5, -0.4, 0.3, -0.2, -0.1, 0.4, 0.1, -0.6)
R> Xb <- drop(X %*% eta.true)
R> old.seed <- setRNG(test.rng)
R> par.lr <- par.gh <- matrix(NA, nsim, 8)
R> stats.lr <- stats.gh <- matrix(NA, nsim, 5)
R> sumDelta <- rep(NA, nsim)
R> t1 <- t2 <-0
```

The `sum(Delta)` indicates that 81.8 percent of the individuals experienced failure. The results are shown in Table 3 for both log-rank and Gehan estimators.

		Log-rank			Gehan		
Parameter	Truth	Mean	Bias	Std. Dev	Mean	Bias	Std. Dev
X_1	0.5	0.498	−0.002	0.233	0.501	0.001	0.139
X_2	−0.4	−0.386	0.014	0.228	−0.397	0.003	0.136
X_3	0.3	0.297	−0.003	0.226	0.298	−0.002	0.135
X_4	−0.2	−0.196	0.004	0.256	−0.195	0.005	0.152
X_5	−0.1	−0.102	−0.002	0.270	−0.104	−0.004	0.166
X_6	0.4	0.405	0.005	0.277	0.400	0.000	0.168
X_7	0.1	0.100	0.000	0.011	0.100	0.000	0.007
X_8	−0.6	−0.601	−0.001	0.114	−0.601	−0.001	0.068

Table 3: Simulation results for the rank-based regression in accelerated failure time model (1000 simulations). Estimates were obtained using the `dfsane` algorithm with `M=100`.

```
R> cat("Simulation for Table 2: ")
R> for (i in 1:nsim) {
+     cat( i, " ")
+     err <- rlnorm(n, mean=1)
+     Y.orig <- Xb + err
+     cutoff <- floor(quantile(Y.orig, prob=0.5))
+     cens <- runif(n, cutoff, quantile(Y.orig, prob=0.95))
+     Y <- pmin(cens, Y.orig)
+     delta <- 1 * (Y.orig <= cens)
+     sumDelta[i] <- sum(delta)
+
+     t1 <- t1 + system.time(ans.eta <-
+        dfsane(par=rep(0,8), fn=aft.eqn,
+            control = list(NM = TRUE,  trace = FALSE),
+            X=X, Y=Y, delta = delta, weights = "logrank"))[1]
```

```
+     par.lr[i,] <- ans.eta$par
+     stats.lr[i, ] <- c(ans.eta$iter, ans.eta$feval, as.numeric(t1),
+                              ans.eta$conv, ans.eta$resid)
+
+     t2 <- t2 + system.time(ans.eta <-
+        dfsane(par=rep(0,8), fn=aft.eqn,
+           control = list(NM = TRUE,  trace = FALSE),
+           X=X, Y=Y, delta = delta, weights="gehan"))[1]
+     par.gh[i,] <- ans.eta$par
+     stats.gh[i, ] <- c(ans.eta$iter, ans.eta$feval, as.numeric(t2),
+                              ans.eta$conv, ans.eta$resid)
+     invisible({gc(); gc()})
+     }
R> cat("\n")

R> print(t1/nsim)

user.self
    0.405

R> print(t2/nsim)

user.self
    0.751

R> print(mean(sumDelta))

[1] 820.9

R> mean.lr <- signif(colMeans(par.lr),3)
R> bias.lr <- mean.lr - eta.true
R> sd.lr <- signif(apply(par.lr, 2, sd),3)
R> mean.gh <- signif(colMeans(par.gh),3)
R> bias.gh <- mean.gh - eta.true
R> sd.gh <- signif(apply(par.gh, 2, sd),3)
R> signif(colMeans(stats.lr),3)

[1] 4.81e+02 1.24e+03 2.39e+00 5.00e+00 1.03e-02

R> signif(colMeans(stats.gh),3)

[1] 6.79e+02 2.15e+03 4.19e+00 5.00e+00 1.33e-03
```

We conducted another test of the ability of `dfsane` for solving the semi-parametric AFT equations (9) on a real data set that has been widely used in survival analysis: Mayo Clinic's primary biliary cirrhosis (PBC) data (see the appendix of Dickson, Grambsch, Fleming, Fisher,

table2		Log-rank	Gehan	table2		Log-rank	Gehan
	Truth	Mean	Bias	Std. Dev.	Mean	Bias	Std. Dev.
X_1	0.500	0.506	0.006	0.164	0.526	0.026	0.095
X_2	−0.400	−0.341	0.059	0.274	−0.365	0.035	0.114
X_3	0.300	0.339	0.039	0.167	0.377	0.077	0.120
X_4	−0.200	−0.249	−0.049	0.230	−0.191	0.009	0.125
X_5	−0.100	−0.043	0.057	0.326	−0.039	0.061	0.210
X_6	0.400	0.351	−0.049	0.203	0.376	−0.024	0.117
X_7	0.100	0.100	0.000	0.009	0.100	0.000	0.005
X_8	−0.600	−0.513	0.087	0.078	−0.554	0.046	0.052

Table 4: Simulation results for the rank-based regression in accelerated failure time model (10 simulations). Estimates were obtained using the `dfsane` algorithm with `M=100`.

and Langworthy 1989). A corrected version of this data is available at the Mayo Clinic's website and in the R package **survival** (Therneau and original R port by Thomas Lumley 2009). We computed the regression coefficients for an AFT model with 5 covariates, age, log(albumin), log (bilirubin), edema, and log(protime), with log-rank and Gehan weights. We also estimated standard errors for them using 500 bootstrap samples. Results are provided in Table 5.

	Gehan				Log-rank			
Covariate	`dfsane`		Jin *et al.* (2003)		`dfsane`		Jin *et al.* (2003)	
`age`	−0.026	(0.006)	−0.025	(0.006)	−0.027	(0.006)	−0.026	(0.005)
`log(albumin)`	1.456	(0.518)	1.499	(0.523)	1.094	(0.504)	1.633	(0.449)
`log(bili)`	−0.574	(0.069)	−0.558	(0.063)	−0.596	(0.065)	−0.572	(0.056)
`edema`	−0.996	(0.291)	−0.924	(0.284)	−0.842	(0.310)	−0.762	(0.246)
`log(protime)`	−2.124	(0.918)	−2.776	(0.776)	−0.941	(0.695)	−1.918	(0.548)
Residual norm $\frac{\|F(x_n)\|}{\sqrt{p}}$	0.002		0.005		0.040		0.173	

Table 5: Rank-based regression of the accelerated failure time (AFT) model for the primary biliary cirrhosis (PBC) data set. Point estimates and standard errors (in parentheses) are provided. Standard errors for `dfsane` are obtained from 500 bootstrap samples.

```
R> require("survival")
R> attach(pbc)

R> Y <- log(time)
R> delta <- status == 2
R> X <- cbind(age,  log(albumin), log(bili), edema, log(protime))
R> missing <- apply(X, 1, function(x) any(is.na(x)))
R> Y <- Y[!missing]
R> X <- X[!missing, ]
R> delta <- delta[!missing]
R> ####### Log-rank estimator #######
R> t1 <- system.time(ans.lr <-
```

```
+           dfsane(par=rep(0, ncol(X)), fn = aft.eqn,
+                control=list(NM = TRUE, M = 100, noimp = 500, trace = FALSE),
+                X=X, Y=Y, delta=delta))[1]
R> # With maxit=5000 this fails with "Lack of improvement in objective function"
R> #  not with "Maximum limit for iterations exceeded"
R>
R> t1

user.self
    0.316

R> ans.lr

$par
        age                                edema
-0.02604586  1.47049360 -0.58095618 -0.71477055 -1.35834955

$residual
[1] 0.0397396

$fn.reduction
[1] 7.15225

$feval
[1] 2088

$iter
[1] 1501

$convergence
[1] 1

$message
[1] "Maximum limit for iterations exceeded"

R> ####### Gehan estimator #######
R> t2 <- system.time(ans.gh <-
+         dfsane(par = rep(0, ncol(X)), fn = aft.eqn,
+         control = list(NM = TRUE, M = 100, noimp = 500, trace = FALSE),
+         X=X, Y=Y, delta=delta, weights = "gehan"))[1]
R> t2

user.self
    0.405

R> ans.gh
```

```
$par
        age                                    edema
-0.02548359  1.51373621 -0.56088393 -0.93627892 -2.64109642

$residual
[1] 0.001853857

$fn.reduction
[1] 3.529347

$feval
[1] 2384

$iter
[1] 1501

$convergence
[1] 1

$message
[1] "Maximum limit for iterations exceeded"
```

The sections which do estimates with code from Jin's web site are not executed in the vignette because they takes too long. You can change this by indicating `eval=TRUE` *for the Scode sections in the vignette.*

```
R> # This source defines functions l1fit and aft.fun
R> source("https://www.columbia.edu/~zj7/aftsp.R")
R> # N.B. aft.fun resets the RNG seed by default to a fixed value,
R> #    and does not reset it. Beware.
R>
R>
R> require("quantreg")
R> t3 <- system.time(ans.jin <-
+          aft.fun(x=X, y=Y, delta=delta, mcsize=1))[1]
R> t3
R> ans.jin$beta

R> #  without Jin's results
R> U <- function(x, func, ...)  sqrt(mean(func(x, ...)^2))
R> # result from Jin et al. (2003) gives higher residuals
R> table3.ResidualNorm <- c(
+     U(ans.gh$par,       func=aft.eqn, X=X, Y=Y, delta=delta,
+         weights="gehan"),
+     U(ans.lr$par,       func=aft.eqn, X=X, Y=Y, delta=delta))

R> #  with Jin's results
R> U <- function(x, func, ...)  sqrt(mean(func(x, ...)^2))
```

```
R> # result from Jin et al. (2003) gives higher residuals
R> table3.ResidualNorm <- c(
+     U(ans.gh$par,       func=aft.eqn, X=X, Y=Y, delta=delta,
+         weights="gehan"),
+     U(ans.jin$beta[1,], func=aft.eqn, X=X, Y=Y, delta=delta,
+         weights="gehan"),
+     U(ans.lr$par,       func=aft.eqn, X=X, Y=Y, delta=delta),
+     U(ans.jin$beta[2,], func=aft.eqn, X=X, Y=Y, delta=delta))
R>

R> # Bootstrap to obtain standard errors
R>
R> Y <- log(time)
R> delta <- status==2
R> X <- cbind(age,  log(albumin), log(bili), edema, log(protime))
R> missing <- apply(X, 1, function(x) any(is.na(x)))
R> Y.orig <- Y[!missing]
R> X.orig <- X[!missing, ]
R> delta.orig <- delta[!missing]
R> old.seed <- setRNG(test.rng)
R> lr.boot <- gh.boot <- matrix(NA, nboot, ncol(X))
R> time1 <- time2 <- 0

R> cat("Bootstrap sample: ")
R> for (i in 1:nboot) {
+     cat(i, " ")
+     select <- sample(1:nrow(X.orig), size=nrow(X.orig), rep=TRUE)
+     Y <- Y.orig[select]
+     X <- X.orig[select, ]
+     delta <- delta.orig[select]
+     time1 <- time1 + system.time(ans.lr <-
+           dfsane(par = rep(0, ncol(X)), fn = aft.eqn,
+             control = list(NM = TRUE, M = 100, noimp = 500, trace = FALSE),
+             X=X, Y=Y, delta=delta))[1]
+     time2 <- time2 + system.time(ans.gh <-
+           dfsane(par = rep(0, ncol(X)), fn = aft.eqn,
+             control = list(NM = TRUE, M = 100, noimp = 500, trace = FALSE),
+             X=X, Y=Y, delta=delta, weights = "gehan"))[1]
+     lr.boot[i,] <- ans.lr$par
+     gh.boot[i,] <- ans.gh$par
+     }
R> cat("\n")

R> time3 <- system.time( ans.jin.boot <-
+        aft.fun(x = X.orig, y = Y.orig, delta = delta.orig,
+           mcsize = nboot))[1]
R> time1
```

```
R> time2
R> time3
R> colMeans(lr.boot)
R> # Results on different systems and versions of R:
R> # [1] -0.02744423  1.09871350 -0.59597720 -0.84169498 -0.95067376
R> # [1] -0.02718006  1.01484050 -0.60553894 -0.83216296 -0.82671339
R> # [1] -0.02746916  1.09371431 -0.59630955 -0.84170621 -0.94147407
R>
R> sd(lr.boot) * (499/500)
R> # Results on different systems and versions of R:
R> # [1] 0.005778319 0.497075716 0.064839483 0.306026261 0.690452468
R> # [1] 0.006005054 0.579962922 0.068367668 0.307980986 0.665742686
R> # [1] 0.005777676 0.504362828 0.064742446 0.309687062 0.695128194
R>
R> colMeans(gh.boot)
R> # Results on different systems and versions of R:
R> # [1] -0.0263899  1.4477801 -0.5756074 -0.9990443 -2.0961280
R> # [1] -0.02616728  1.41126364 -0.58311902 -1.00953045 -2.01724976
R> # [1] -0.02633854  1.45577255 -0.57439183 -0.99630007 -2.12363711
R>
R> sd(gh.boot) * (499/500)
R> # Results on different systems and versions of R:
R> # [1] 0.006248941 0.519016144 0.068759981 0.294145730 0.919565487
R> # [1] 0.005599693 0.571631837 0.075018323 0.304463597 1.043196254
R> # [1] 0.006183826 0.518332233 0.068672881 0.291036025 0.917733660
R>
R>
R> ans.jin.boot$beta
R> sqrt(diag(ans.jin.boot$betacov[,,2]))  # log-rank
R> # Results on different systems and versions of R:
R> # [1] 0.005304614 0.470080732 0.053191766 0.224331718 0.545344403
R> # [1] 0.00517431 0.44904332 0.05632078 0.24613883 0.54826652
R> # [1] 0.00517431 0.44904332 0.05632078 0.24613883 0.54826652
R>
R> sqrt(diag(ans.jin.boot$betacov[,,1]))  # Gehan
R> # Results on different systems and versions of R:
R> # [1] 0.005553049 0.522259799 0.061634483 0.270337048 0.803683570
R> # [1] 0.005659013 0.522871858 0.062670939 0.283731999 0.775959845
R> # [1] 0.005659013 0.522871858 0.062670939 0.283731999 0.775959845
```

The table is generated here without the results from running Jin's code.

We also estimated the semiparametric AFT model using the algorithm of Jin *et al.* (2003). (The R code was obtained from Dr. Jin's website https://www.columbia.edu/~zj7/index_files/Page382.htm). Comparing our results with theirs (see Table 5), we observe some differences in both the point estimates and standard errors. The point estimates for the Gehan estimator seem to agree reasonably well. For the logrank estimator, the point estimates of `log(albumin)` and `log(protime)` are considerably smaller (in absolute magnitude) for

table3.part1				
age	−0.027	1.204	−0.028	0.860
log(albumin)	1.350	1.204	1.070	0.860
log(bili)	−0.585	1.204	−0.603	0.860
edema	−1.033	1.204	−0.844	0.860
log(protime)	−1.957	1.204	−0.973	0.860

table3.ResidualNorm			
Residual norm	$\frac{\|F(x_n)\|}{\sqrt{p}}$	0.002	0.040

Table 6: Rank-based regression of the accelerated failure time (AFT) model for the primary biliary cirrhosis (PBC) data set. Point estimates and standard errors (in parentheses) are provided. Standard errors for `dfsane` are obtained from 50 bootstrap samples.

`dfsane` than those obtained using the method of Jin *et al.* (2003). The residual norm from `dfsane` is 2 to 4 times smaller than that of Jin *et al.* (2003), indicating that our solutions to (9) are better than those in Jin *et al.* (2003). More accurate solutions for the log-rank estimator can be obtained from Jin's algorithm by using a larger number of iterations. For example, when we used 6 iterations (the default is 3), the residual error was almost as small as that from `dfsane`, and the point estimates were in better agreement. Another noteworthy difference, especially for the log-rank estimator, is that our bootstrapped standard error estimates are higher than the standard error estimates of Jin *et al.* (2003), which were obtained using a perturbed estimating equation approach.

We also note that our AFT model estimation using `dfsane` is substantially faster than the algorithm proposed in Jin *et al.* (2003). For example, for the PBC data, the total CPU time for Gehan and log-rank estimates using `dfsane` is around 6.5 seconds, whereas it is around 99 seconds for Jin's algorithm (for 3 iterations). For standard error estimation, the `dfsane` algorithm took 1 hour, and Jin's algorithm took 6 hours for 500 Monte-Carlo samples. A major limitation of Jin's R function is that it can only handle small data sets. It runs into memory limits for even moderate size data sets, for example, it crashed when we tried it on one of the simulated data sets discussed previously with n=1000 and 8 covariates.

It should also be noted that some problems are intrinsically hard and cannot be solved to within a small error tolerance (e.g. default tolerance $= 1.e-07$). The AFT model problem is an example of this. This is a non-smooth problem. We cannot always achieve a tolerance of $1.0e-07$ in these problems. With the PBC data, there may not even be an "exact" solution that will yield a residual of $1.0e-07$. However, we can obtain a solution that is accurate enough. It might be possible to improve upon the solution given by `dfsane` by changing the control parameters (e.g. `M, noimp, maxit`) or by using `BBsolve`, but it may not be worth the added effort for this problem.

5. Conclusions

The package **BB** provides functions which improve the capabilities of R for solving nonlinear systems of equations and for optimizing smooth, nonlinear functions in the following ways:

1. The function `BBsolve` offers a reliable, low-cost method to solving large-scale nonlinear systems of equations.

2. The function `BBoptim` offers a reliable, low-cost method to optimizing smooth, large-scale nonlinear problems.

3. The function `multiStart` can be used to find multiple roots or multiple local optima.

4. `dfsane` appears to be promising for solving non-smooth estimating equations, since it does not involve any derivatives (see condition 7).

5. Rank-based regression estimation in the accelerated failure time models can be performed effectively in R using the `dfsane` function in `BB`.

Acknowledgements

The work of first author (R.V.) was supported by the funding from NIH grant DA023879-01. The authors would like to thank Drs. Marcos Raydan, Jose-Mario Martinez, Dimitris Rizopoulos, Constantine Frangakis, and Daniel Scharfstein for the many valuable discussions pertaining to this research. They would also like to thank the two anonymous referees, the associate editor, and Achim Zeileis for their penetrating comments which improved the quality of the manuscript and the software package.

References

Barzilai J, Borwein JM (1988). "Two-Point Step Size Gradient Methods." *IMA Journal of Numerical Analysis*, **8**(1), 141–148.

Birgin EG, Martínez JM, Raydan M (2001). "Algorithm 813: SPG—Software for Convex-Constrained Optimization." *ACM Transactions on Mathematical Software*, **27**(3), 340–349.

Carroll RJ, Ruppert D (1988). *Transformation and Weighting in Regression*. Chapman & Hall/CRC Press, London, UK.

Dennis JE, Schnabel RB (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, New Jersey.

Dickson ER, Grambsch PM, Fleming TR, Fisher LD, Langworthy A (1989). "Prognosis in primary biliary cirrhosis: Model for decision making." *Hepatology*, **10**, 1–7.

Diggle P, Heagerty P, Liang KY, Zeger SL (2002). *The Analysis of Longitudinal Data*. Oxford University Press, New York.

Fletcher R (2001). "On the Barzilai-Borwein Method." *Technical Report NA/207*, University of Dundee, Dundee, Scotland.

Grippo L, Lampariello F, Lucidi S (1986). "A Nonmonotone Line Search Technique for Newton's Method." *SIAM Journal on Numerical Analysis*, **23**, 707–16.

Hasselman B (2009). *nleqslv: Solve systems of non linear equations*. R package version 1.4, URL https://CRAN.R-project.org/package=nleqslv.

Jin Z, Lin DY, Wei LJ, Ying Z (2003). "Rank-Based Inference for the Accelerated Failure Time Model." *Biometrika*, **90**, 341–353.

Kalbfleisch JD, Prentice RL (2002). *The Statistical Analysis of Failure Time Data*. John Wiley & Sons, Hoboken, New Jersey.

Kearfott R (1987). "Some Tests of Generalized Bisection." *ACM Transactions on Mathematical Software*, **13**(3), 197–220.

La Cruz W, Martínez JM, Raydan M (2006). "Spectral Residual Method Without Gradient Information for Solving Large-Scale Nonlinear Systems of Equations." *Mathematics of Computation*, **75**(255), 1429.

La Cruz W, Raydan M (2003). "Nonmonotone Spectral Methods for Large-Scale Nonlinear Systems." *Optimization Methods and Software*, **18**(5), 583–599.

Lin DY, Geyer CJ (1992). "Computational Methods for Semiparametric Linear Regression with Censored Data." *Journal of Computational and Graphical Statistics*, **1**(1), 77–90.

Luengo F, Raydan M (2003). "Gradient Method with Dynamical Retards for Large-Scale Optimization Problems." *Electronic Transactions on Numerical Analysis*, **16**, 186–193.

Luksan L, Vlcek J (2003). "Test Problems for Unconstrained Optimization." *Technical report*, Academy of Sciences of the Czech Republic, Institute of Computer Science.

Nelder JA, Mead R (1965). "A Simplex Method for Function Minimization." *Computer Journal*, **7**, 308.

Ortega JM, Rheinboldt WC (1970). *Iterative Solution of Non-Linear Equations in Several Variables*. Academic Press, New York.

Raydan M (1997). "The Barzilai and Borwein Gradient Method for the Large Scale Unconstrained Minimization Problem." *SIAM Journal of Optimization*, **7**, 26–33.

Raydan M (2009). "Marcos Raydan's Home Page."

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL https://www.R-project.org/.

Small CG, Wang J (2003). *Numerical Methods for Nonlinear Estimating Equations*. Oxford University Press, New York.

Therneau T, original R port by Thomas Lumley (2009). *survival: Survival analysis, including penalised likelihood*. R package version 2.35-4, URL https://CRAN.R-project.org/package=survival.

Varadhan R, Gilbert P (2009). "**BB**: An R Package for Solving a Large System of Nonlinear Equations and for Optimizing a High-Dimensional Nonlinear Objective Function." *Journal of Statistical Software*, **32**(4). URL https://www.jstatsoft.org/v32/i04/.

Varadhan R, Roland C (2008). "Simple and Globally Convergent Methods for Accelerating the Convergence of Any EM Algorithm." *Scandinavian Journal of Statistics*, **35**(2), 335–353.

Zeileis A (2005). "CRAN Task Views." *R News*, **5**(1), 39–40. URL https://CRAN.R-project.org/doc/Rnews/.

A. Appendix: Test Functions

1. *Exponential function 3:* $F(x) = (F_1(x), \cdots, F_p(x))^\top$, where:

$$\begin{aligned} F_1(x) &= e^{x_1} - 1 \\ F_i(x) &= (i/10)\,(e^{x_i} + x_{i-1} - 1), \quad i = 2, 3, \cdots, p \end{aligned}$$

   Initial value: $x_0 = \text{rnorm}(p)$

2. *Trigexp function:* $F(x) = (F_1(x), \cdots, F_p(x))^\top$, where:

$$\begin{aligned} F_1(x) &= 3x_1^3 + 2x_2 - 5 + \sin(x_1 - x_2)\sin(x_1 + x_2) \\ F_i(x) &= -x_{i-1}e^{(x_{i-1}-x_i)} + x_i(4 + 3x_i^2) + 2x_{i+1} + \sin(x_i - x_{i+1})\sin(x_i + x_{i+1}) - 8, \\ &\quad i = 2, 3, \cdots, p-1 \\ F_p(x) &= -x_{p-1}e^{(x_{p-1}-x_p)} + 4x_p - 3. \end{aligned}$$

   Initial value: $x_0 = \text{rnorm}(p)$

3. *Broyden tridiagonal function:* $F(x) = (F_1(x), \cdots, F_p(x))^\top$, where:

$$\begin{aligned} F_1(x) &= x_1(3 - 0.5x_1) - 2x_2 + 1, \\ F_i(x) &= x_i(3 - 0.5x_i) - x_{i-1} - 2x_{i+1} + 1, \quad i = 2, 3, \cdots, p-1 \\ F_p(x) &= x_p(3 - 0.5x_p) - x_{p-1} + 1. \end{aligned}$$

   Initial value: $x_0 = -\text{ runif}(p)$

4. *Extended-Rosenbrock function:* $F(x) = (F_1(x), \cdots, F_p(x))^\top$, where,
   for $i = 1, 2, \cdots, p/2$,

$$\begin{aligned} F_{2i-1}(x) &= 10(x_{2i} - x_{2i-1}^2) \\ F_{2i}(x) &= 1 - x_{2i-1}. \end{aligned}$$

   Initial value: $x_0 = \text{runif}(p)$

5. *Troesch function:* $F(x) = (F_1(x), \cdots, F_p(x))^\top$, where:

$$\begin{aligned} F_1(x) &= 2x_1 + \rho h^2 \sinh(\rho x_1) - x_2, \\ F_i(x) &= 2x_i + \rho h^2 \sinh(\rho x_i) - x_{i-1} - x_{i+1}, \quad i = 2, 3, \cdots, p-1 \\ F_p(x) &= 2x_p + \rho h^2 \sinh(\rho x_p) - x_{p-1} - 1, \end{aligned}$$

   where $\rho = 10$, $h = 1/(p+1)$.
   Initial value: $x_0 = \text{sort}(\text{runif}(p))$

6. *Discretized version of Chandrasekhar's H-equation:* $F(x) = (F_1(x), \cdots, F_p(x))^\top$, where:

$$F_i(x) \;=\; x_i \;-\; \left(1 \;-\; \frac{c}{2p} \sum_{j=1}^{p} \frac{y_i\, x_j}{y_i + y_j}\right)^{-1}, \quad i = 1, 2, \cdots, p$$

Initial value: $x_0 = \text{runif}(p)$

Affiliation:

Ravi Varadhan
The Center on Aging and Health & School of Medicine
Johns Hopkins University
Baltimore, USA
E-mail: ravi.varadhan@jhu.edu

Paul D. Gilbert
Canadian Economic Analysis Department
Bank of Canada
Ottawa, Canada
E-mail: pgilbert@bank-banque-canada.ca
URL: https://www.bank-banque-canada.ca/index.html