1 Overview of BB

"BB" is a package intended for two purposes: (1) for solving a nonlinear system of equations, and (2) for finding a local optimum (can be minimum or maximum) of a scalar, objective function. An attractive feature of the package is that it has minimum memory requirements. Therefore, it is particularly well suited to solving high-dimensional problems with tens of thousands of parameters. However, *BB* can also be used to solve a single nonlinear equation or optimize a function with just one variable. The functions in this package are made available with:

```
> library("BB")
```

You can look at the basic information on the package, including all the available functions wtih

```
> help(package=BB)
```

The three basic functions are: *spg*, *dfsane*, and *sane*. You should *spg* for optimization, and either *dfsane* or *sane* for solving a nonlinear system of equations. We prefer *dfsane*, since it tends to perform slightly better than *sane*. There are also 3 higher level functions: *BBoptim*, *BBsolve*, and *multiStart*. *BBoptim* is a wrapper for *spg* in the sense that it calls *spg* repeatedly with different algorithmic options. It can be used when *spg* fails to find a local optimum, or it can be used in place of *spg*. Similarly, *BBsolve* is a wrapper for *dfsane* in the sense that it calls *dfsane* repeatedly with different algorithmic options. It can be used when *dfsane* (*sane*) fails to find a local optimum, or it can be used in place of *dfsane* (*sane*). The *multiStart* function can accept multiple starting values. It can be used for either solving a nonlinear system or for optimizing. It is useful for exploring sensitivity to starting values, and also for finding multiple solutions.

The package *setRNG* is not necessary, but if you want to exactly reproduce the examples in this guide then do this:

```
> require("setRNG")
> test.rng <- list(kind="Wichmann-Hill", normal.kind="Box-Muller", seed=1236)
> setRNG(test.rng)
```

after which the example need to be run in the order here (or at least the parts that generate random numbers).

2 How to solve a nonlinear system of equations with BB?

The first two examples are from La Cruz and Raydan, Optim Methods and Software 2003, 18 (583-599).

```
> expo3 <- function(p) {
  #  From La Cruz and Raydan, Optim Methods and Software 2003, 18 (583-599)
  n <- length(p)
  f <- rep(NA, n)
  onm1 <- 1:(n-1)
  f[onm1] <- onm1/10 * (1 - p[onm1]^2 - exp(-p[onm1]^2))
  f[n] <- n/10 * (1 - exp(-p[n]^2))
  f
  }
> p0 <- runif(10)
> ans <- dfsane(par=p0, fn=expo3)
```

```
Iteration:  0  ||F(x0)||:  0.2024112
iteration:  10  ||F(xn)|| =   0.07536174
iteration:  20  ||F(xn)|| =   0.08777425
iteration:  30  ||F(xn)|| =   0.005029196
iteration:  40  ||F(xn)|| =   0.001517709
iteration:  50  ||F(xn)|| =   0.001769548
iteration:  60  ||F(xn)|| =   0.007896929
iteration:  70  ||F(xn)|| =   0.0001410588
iteration:  80  ||F(xn)|| =   2.002796e-06
```

```
> ans
```

```
$par
 [1]  3.819663e-02  3.031250e-02  2.647897e-02  2.404688e-02  2.233208e-02
 [6]  2.101498e-02  1.996221e-02  1.909301e-02  1.835779e-02 -7.493381e-06

$residual
[1] 6.645152e-08

$fn.reduction
[1] 0.6400804

$feval
[1] 96

$iter
[1] 85

$convergence
[1] 0

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

Let us look at the output from *dfsane*. It is a list with 7 components. The most important components to focus on are the two named "*par*" and "*conver-*

gence". *ans$par* provides the solution from *dfsane*, but this is a root if and only if *ans$convergence* is equal to *0*, i.e. *ans$message* should say "Successful convergence". Otherwise, the algorithm has failed.

Now, we show an example demonstrating the ability of BB to solve a large system of equations, N = 10000.

```
> trigexp <- function(x) {
  n <- length(x)
  F <- rep(NA, n)
  F[1] <- 3*x[1]^2 + 2*x[2] - 5 + sin(x[1] - x[2]) * sin(x[1] + x[2])
  tn1 <- 2:(n-1)
  F[tn1] <- -x[tn1-1] * exp(x[tn1-1] - x[tn1]) + x[tn1] * ( 4 + 3*x[tn1]^2) +
          2 * x[tn1 + 1] + sin(x[tn1] - x[tn1 + 1]) * sin(x[tn1] + x[tn1 + 1]) - 8
  F[n] <- -x[n-1] * exp(x[n-1] - x[n]) + 4*x[n] - 3
  F
  }
> n <- 10000
> p0 <- runif(n)
> ans <- dfsane(par=p0, fn=trigexp, control=list(trace=FALSE))
> ans$message

[1] "Successful convergence"

> ans$resid

[1] 5.725351e-08
```

The next example is from Freudenstein and Roth function (Broyden, Mathematics of Computation 1965, p. 577-593).

```
> froth <- function(p){
  f <- rep(NA,length(p))
  f[1] <- -13 + p[1] + (p[2]*(5 - p[2]) - 2) * p[2]
  f[2] <- -29 + p[1] + (p[2]*(1 + p[2]) - 14) * p[2]
  f
  }
```

Now, we introduce the function *BBsolve*. For the first starting value, both *dfsane* and *BBsolve* find the zero of the system.

```
> p0 <- c(3,2)
> BBsolve(par=p0, fn=froth)

  Successful convergence.
$par
[1] 5 4

$residual
```

```
[1] 3.659749e-10

$fn.reduction
[1] 0.001827326

$feval
[1] 100

$iter
[1] 10

$convergence
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M     NM
     2     50      1

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

$par
[1] -9.822061 -1.875381

$residual
[1] 11.63811

$fn.reduction
[1] 25.58882

$feval
[1] 137

$iter
[1] 114

$convergence
[1] 5

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

For the next starting value, *BBsolve* finds the zero of the system, but *dfsane* (with defaults) fails.

```
> p0 <- c(1,1)
> BBsolve(par=p0, fn=froth)

  Successful convergence.
$par
[1] 5 4

$residual
[1] 9.579439e-08

$fn.reduction
[1] 6.998875

$feval
[1] 1165

$iter
[1] 247

$convergence
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M     NM
     1     50      1

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

$par
[1] -9.674222 -1.984882

$residual
[1] 12.15994

$fn.reduction
[1] 24.03431

$feval
[1] 138

$iter
[1] 109

$convergence
```

```
[1] 5

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

Try random starting values. Run the following set of code many times. This shows that *BBsolve* is quite robust in finding the zero, whereas *dfsane* (with defaults) is sensitive to starting values. Admittedly, these are poor starting values, but still it would be nice to have a strategy that has a high likelihood of finding a zero of the nonlinear system.

```
> p0 <- rpois(2,10) # two values generated independently from a poisson distribution with me
> BBsolve(par=p0, fn=froth)

  Successful convergence.
$par
[1] 5 4

$residual
[1] 7.330654e-08

$fn.reduction
[1] 0.07273382

$feval
[1] 91

$iter
[1] 41

$convergence
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M     NM
     2     50      1

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

$par
[1] 5 4

$residual
[1] 5.472171e-08
```

```
$fn.reduction
[1] 490.618

$feval
[1] 32

$iter
[1] 31

$convergence
[1] 0

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

Now, we introduce the function *multiStart*. This accepts a matrix of starting values, where each row is a single starting value. *multiStart* calls *BBsolve* for each starting value. Here is a system of 3 non-linear equations, where each equation is a high-degree polynomial. This system has 12 real-valued roots and 126 complex-valued roots. Here we will demonstrate how to identify all the 12 real roots using *multiStart*. Note that we specify the '*action*' argument in the following call to *multiStart* only to highlight that *multiStart* can be used for both solving a system of equations and for optimization. The default is '*action = "solve"*', so it is really not needed in this call.

```
> # Example
> # A high-degree polynomial system (R.B. Kearfoot, ACM 1987)
> # There are 12 real roots (and 126 complex roots to this system!)
> #
> hdp <- function(x) {
    f <- rep(NA, length(x))
    f[1] <- 5 * x[1]^9 - 6 * x[1]^5 * x[2]^2 + x[1] * x[2]^4 + 2 * x[1] * x[3]
    f[2] <- -2 * x[1]^6 * x[2] + 2 * x[1]^2 * x[2]^3 + 2 * x[2] * x[3]
    f[3] <- x[1]^2 + x[2]^2 - 0.265625
    f
    }
```

We generate 200 randomly generated starting values, each a vector of length equal to 3.

```
> set.seed(123)
> p0 <- matrix(runif(600), 200, 3)  # 200 starting values, each of length 3
> ans <- multiStart(par=p0, fn=hdp, action="solve")
> sum(ans$conv)  # number of successful runs = 190
> pmat <- ans$par[ans$conv, ] # selecting only converged solutions
```

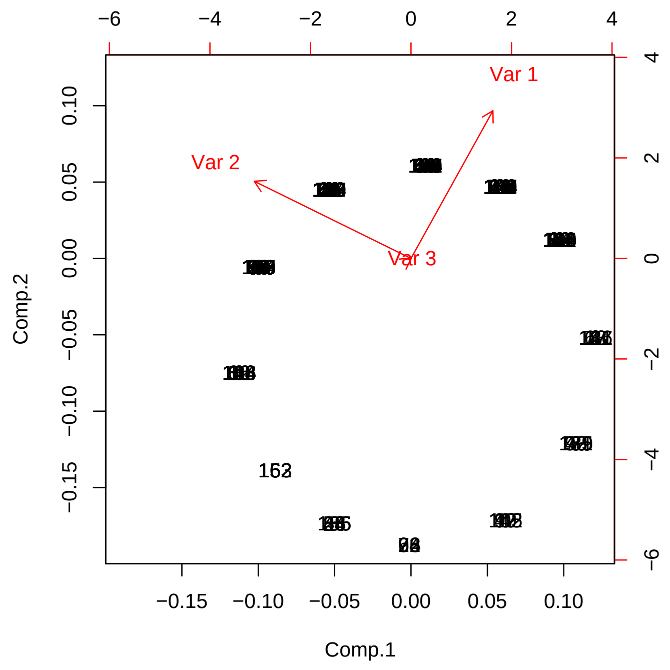
Now, we display the 12 unique real solutions.

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

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

We can also visualize these 12 solutions beautifully using a 'biplot' based on the first 2 principal components of the converged parameter matrix.

```
> pc <- princomp(pmat)
> biplot(pc)  # you can see all 12 solutions beautifully like on a clock!
```



3 How to optimize a nonlinear objective function with BB?

The basic function for optimization is *spg*. It can solve smooth, nonlinear optimization problems with box-constraints, and also other types of constraints using projection. We would like to direct the user to the help page for many examples of how to use *spg*. Here we discuss an example involving estimation of parameters maximizing a log-likelihood function for a binary Poisson mixture distribution.

```
> poissmix.loglik <- function(p,y) {
  # Log-likelihood for a binary Poisson mixture distribution
  i <- 0:(length(y)-1)
  loglik <- y * log(p[1] * exp(-p[2]) * p[2]^i / exp(lgamma(i+1)) +
          (1 - p[1]) * exp(-p[3]) * p[3]^i / exp(lgamma(i+1)))
  return (sum(loglik) )
  }
> # Data from Hasselblad (JASA 1969)
> poissmix.dat <- data.frame(death=0:9, freq=c(162,267,271,185,111,61,27,8,3,1))
```

There are 3 model parameters, which have restricted domains. So, we define these constraints as follows:

```
> lo <- c(0,0,0)  # lower limits for parameters
> hi <- c(1, Inf, Inf) # upper limits for parameters
```

Now, we maximize the log-likelihood function using both *spg* and *BBoptim*, with a randomly generated starting value for the 3 parameters:

```
> p0 <- runif(3,c(0.2,1,1),c(0.8,5,8))  # a randomly generated vector of length 3
> y <- c(162,267,271,185,111,61,27,8,3,1)
> ans1 <- spg(par=p0, fn=poissmix.loglik, y=y, lower=lo, upper=hi,
            control=list(maximize=TRUE, trace=FALSE))
> ans1

$par
[1] 0.3598829 1.2560909 2.6634013

$value
[1] -1989.946

$gradient
[1] 2.273737e-06

$fn.reduction
[1] -929.1606
```

```
$iter
[1] 69

$feval
[1] 78

$convergence
[1] 0

$message
[1] "Successful convergence"

> ans2 <- BBoptim(par=p0, fn=poissmix.loglik, y=y, lower=lo, upper=hi,
            control=list(maximize=TRUE))

  Successful convergence.

> ans2

$par
[1] 0.3598832 1.2560913 2.6634016

$value
[1] -1989.946

$gradient
[1] 2.273737e-06

$fn.reduction
[1] -929.1606

$iter
[1] 55

$feval
[1] 57

$convergence
[1] 0

$message
[1] "Successful convergence"

$cpar
method      M
     2     50
```

Note that we had to specify the '*maximize*' option inside the *control* list to let the algorithm know that we are maximizing the objective function, since the default is to minimize the objective function. Also note how we pass the data vector '*y*' to the log-likelihood function, *possmix.loglik*.

Now, we illustrate how to compute the Hessian of the log-likelihood at the MLE, and then how to use the Hessian to compute the standard errors for the parameters. To compute the Hessian we require the package "*numDeriv*."

```
> require(numDeriv)
> hess <- hessian(x=ans2$par, func=poissmix.loglik, y=y)  # Note that we have to pass data v
> hess

           [,1]       [,2]       [,3]
[1,] -907.1100  270.22864  341.25415
[2,]  270.2286 -113.47947  -61.68193
[3,]  341.2542  -61.68193 -192.78201

> se <- sqrt(diag(solve(-hess)))
> se

[1] 0.1946834 0.3500300 0.2504768
```

Now, we explore the use of multiple starting values to see if we can identify multiple local maxima. We have to make sure that we specify '*action = "optimize"*', because the default option in *multiStart* is *"solve"*.

```
> p0 <- matrix(runif(300, c(0.2,1,1), c(0.8,8,8)), 100, 3, byrow=TRUE)  # 3 randomly generat
> ans <- multiStart(par=p0, fn=poissmix.loglik, action="optimize",
        y=y, lower=lo, upper=hi, control=list(maximize=TRUE))
> pmat <- ans$par[ans$conv, ] # selecting only converged solutions
> ans <- round(pmat, 4)
> ans[!duplicated(ans), ]
```

This seemingly identifies many solutions. However, except for two solutions, the rest are degenerate (i.e. the mixing proportion, which is the first parameter, is either 0 or 1). The two non-degenerate solutions are actually the same, except that the labels for the first and second components are switched.