rHanso

Project Description

Hybrid Algorithm for Non Smooth Algorithm is an algorithm for general unconstrained optimization for convex or non convex functions. This idea is proposed by Michael Overton over a series of papers, and finally accompanied by a Matlab library. rHanso is an implementation of this algorithm in R.

Installation

Currently rHanso latest version is on R-forge. To install use

```
install.packages("rHanso", repos="http://R-Forge.R-project.org")
```

rHanso functions

The most important function of rHanso is hanso(), which can be used for minimization of functions based on Hanso algorithm. Also, there are several other useful functions provided by rHanso. Strong or weak wolfe line search has been implemented in linesch_sw(), and linesch_ww() functions. BFGS algorithm for minimization of functions has been added in bfgs(). gradsamp() is a function for running just the gradient sampling algorithm by its own. The main function hanso() uses all of the other functions, so results of minimization found by hanso() should be atleast as good as bfgs() or gradsamp(). But bfgs() and gradsamp() provides a easy interface for testing out those algorithms. Along with hanso algorithm, rHanso also provides two other optimization routines that are different from bfgs based methods. nlcg() provides a Non Linear Conjugate Gradient solver and shor() is an implementation of Shor's r algorithm. Both of these are alternate ways of optimizing functions and can be thought of as competitor of Hanso. We provide them all, so that users can test and compare them with their own functions.

hanso():

Below we show Shor's piecewise continuous function as the test function for hanso. The function and its exact gradient is available in the package itself. shor_f() is the function, and shor_g() is the gradient for that function.

```
> shor_f
function (x)
{
    stopifnot(is.numeric(x), length(x) == 5)
    x <- as.matrix(c(x))
    A <- matrix(c(0, 2, 1, 1, 3, 0, 1, 1, 0, 1, 0, 1, 2, 4, 2,
        2, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 0, 1, 1,
        2, 0, 0, 1, 2, 1, 0, 0, 3, 2, 2, 1, 1, 1, 1, 0, 0), 5,
        10, byrow = TRUE)
    b <- as.matrix(c(1, 5, 10, 2, 4, 3, 1.7, 2.5, 6, 4.5))
    f <- 0
    for (i in 1:10) {
        d <- b[i] * sum((x - A[, i])^2)
        if (d > f)
            f <- d
    }
    f
}
> shor_g
function (x)
{
    stopifnot(is.numeric(x), length(x) == 5)
    x <- as.matrix(c(x))
    A <- matrix(c(0, 2, 1, 1, 3, 0, 1, 1, 0, 1, 0, 1, 2, 4, 2,
        2, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 0, 1, 1,
        2, 0, 0, 1, 2, 1, 0, 0, 3, 2, 2, 1, 1, 1, 1, 0, 0), 5,
        10, byrow = TRUE)
    b <- as.matrix(c(1, 5, 10, 2, 4, 3, 1.7, 2.5, 6, 4.5))
    f <- 0
    for (i in 1:10) {
        d <- b[i] * sum((x - A[, i])^2)
        if (d > f) {
            f <- d
```

```
            k <- i
        }
    }
    2 * b[k] * (x - A[, k])
}
> res <- hanso(shor_f, shor_g, nvar = 5)
Hanso: Best value found by BFGS =  25.04897
gradsamp: completed iteration= 100
gradsamp: not descent direction, quit at iter= 23
gradsamp: not descent direction, quit at iter= 9
Best value found by Gradient Sampling =  22.60016
> res$x
          [,1]
[1,] 1.1243549
[2,] 0.9794637
[3,] 1.4777262
[4,] 0.9202338
[5,] 1.1242945
> res$f
[1] 22.60016
```

shor()

We show the result of the shor() with the same function. The function uses 10 initial values randomly chosen using uniform distribution in [0, 1]. As output, it produces the 10 points corresponding to each of the starting points. So, we take the point which gives the minimum value of the function among those ten values.

```
> res <- shor(shor_f, shor_g, nvar = 5)
shor: linesearch failed, at iter= 92
shor: linesearch failed, at iter= 91
shor: linesearch failed, at iter= 89
shor: linesearch failed, at iter= 95
shor: linesearch failed, at iter= 87
shor: linesearch failed, at iter= 88
shor: linesearch failed, at iter= 88
shor: linesearch failed, at iter= 94
```

```
shor: linesearch failed, at iter= 89
shor: linesearch failed, at iter= 88
> min(res$f)
[1] 22.60016
> as.matrix(res$x[, which.min(res$f)])
          [,1]
[1,] 1.1243510
[2,] 0.9794616
[3,] 1.4777078
[4,] 0.9202335
[5,] 1.1242916
```

The message : "line search failed" is a stopping condition that is used to stop the loop, so this is not unusual. We see the result is quite consistent with result of hanso().

nlcg()

Now, we try using the CG solver method. The syntax is exactly the same. This also starts from ten random points and gives ten points as output. We select the best among those.

```
> res <- nlcg(shor_f, shor_g, nvar = 5)
> min(res$f)
[1] 24.08442
> as.matrix(res$x[, which.min(res$f)])
          [,1]
[1,] 1.2067266
[2,] 0.9235229
[3,] 1.2796095
[4,] 0.7464705
[5,] 1.0591253
```

Notably, hanso() also works in the same way, starting with ten randomly selected points, or the set of user supplied points, whichever may be the case. But in case of hanso(), the best point selection happens internally, when gradsamp() is applied on the result of bfgs().

bfgs()

Now, we can move on to supporting functions. BFGS algorithm is widely used in optimization domain and several optimization routines in R uses BFGS. But, to our knowledge, no native BFGS implementation has existed for the users in R. So, this function can optimize a function with a known gradient by BFGS algorithm. The syntax is quite similar.

```
> res <- bfgs(shor_f, shor_g, nvar = 5)
> min(res$f)
[1] 24.09951
> as.matrix(res$x[, which.min(res$f)])
          [,1]
[1,] 1.1331839
[2,] 1.0840907
[3,] 1.7475980
[4,] 0.8479449
[5,] 1.2126000
```

Like other functions in this package, bfgs() also uses ten randomly selected initial points to start the iteration. We chose the best result. We have shown the application in the same function. The performance of BFGS is not as good as hanso.

gradsamp()

gradsamp() implements gradient sampling. The hanso algorithm calls gradsamp() after the first stage of using BFGS. This is computationally more expensive than bfgs(), so this is not run on all ten points given by bfgs. Hanso choses the best point and then tries to improve that using gradsamp. On the standalone example however, we do get ten point output, and as usual, we choose the best point.

```
> res <- gradsamp(shor_f, shor_g, nvar = 5)
gradsamp: not descent direction, quit at iter= 163
gradsamp: not descent direction, quit at iter= 27
gradsamp: not descent direction, quit at iter= 8
> min(res$f)
[1] 22.60016
> as.matrix(res$x[, which.min(res$f)])
```

```
          [,1]
[1,] 1.1243477
[2,] 0.9794649
[3,] 1.4777178
[4,] 0.9202243
[5,] 1.1242961
```

linesch_ww() / linesch_sw()

Wolfe's weak and strong line search has been implemented in these two functions. They are used by all the optimization routines in this package. Mostly the choice strong vs wolfe is done using an indicator variable supplied by the user. The default is to use weak line search. Here we show using them standalone.

```
> res <- linesch_ww(shor_f, shor_g, rnorm(5), rep(1,5))
> res$xalpha
[1] 2.8453845 0.1559522 1.3523237 0.5732644 1.4648086
> res$falpha
[1] 73.98621
> res1 <- linesch_sw(shor_f, shor_g, rnorm(5), rep(1, 5))
> res1$x
[1] 0.53379665 0.70740545 0.03491795 2.19378892 2.68227675
> res1$f
[1] 79.5999
```

We have used search direction $(1, 1, 1, 1, 1)$ as input here. The output is the new x satisfying wolfe condition in the given direction, starting from initial point. Also, it gives the function evaluated at that point.

Issues

One of the problems we noticed is Hanso does not perform well with inaccurate derivatives. Although it is intended for optimizations of non smooth functions, but the method is heavily dependent on calculating the derivatives accurately. We initially planned for adding support of numerical differentiation in the package by default. But, it was found that the functions with numerical differentiation, the results were not that accurate. So, finally, we removed it. If the user wants, he still can use

some numerical differentiation methods for supplying the gradient of the function to hanso(), but the accuracy of the results can not be guaranteed in that case.

Another issue that has to deal with gradient dependence, is the algorithm converges quite slowly when there is little change in the function. We tested with a function that goes through a long flat region of relatively low change and then achieving the minimum. The algorithm did find the minimum but only with increased iteration counts.

rImfil

Project Description

Implicit filtering is a steepest descent algorithm for noisy optimization problems with bound constraints. This method can work for functions with many local minimum. So, the problem is:

$$\min_{x \in \Omega} f(x)$$
$$\Omega = \{x \in \mathbb{R}^n | L_i \leq x_i \leq U_i\}$$

Where the set of L and U define the constraints on the domain of x. There could be budget constraint also, as the function can be expensive to compute. So, it would take certain cost to evaluate the function, and the optimization routine would be stopped once it reaches its budget.

Implicit filtering is a sampling method. The optimization is controlled only by evaluating f in a cluster of points in Ω. That evaluation determines the next cluster. The other examples of sampling methods are: co-ordinate search method, Nelder Maed method, Hook Jeevs algorithm.

Installation

The latest version of the code is at R forge. The library can be installed by

```
install.packages("rImfil", repos="http://R-Forge.R-project.org")
```

functions

imfil() is the only function in this library. There is one test function provided.

```
> f_easy
function(x, h, core_data){
  fv <- sum(x*x)
  fv <- fv*(1 + 0.1*sin(10*(x[1]+x[2])))
  ifail <- 0
  icount <- 1
  list(fv = fv, ifail = ifail, icount = icount)
}
```

There is also a default options initialization function provided. imfil_optset() sets the optional arguments to imfil to their default values.

imfil()

imfil() works on a sampling method. The samples are arranged on a stencil. The current iterate is x_c and the value of function is $f(x_c)$. The default algorithm is to sample 2N points : $x_c \pm hv_i, i \leq N$, where $v_i = (U_i - l_i)e_i$. The default sizes are given in imfil_optset().

The budget is specified by the user. The objective function provides the cost as one of its output fields. The routine stops when the budget is reached.

The constraints are specified by the user also. In the driver_easy() we specify external bounds. The general form of the objective function is

```
[fout, ifail, icount] = f(x)
```

Results from partial iterations

The program was not able to run error free when we tested it. But it was able to complete several steps of iteration with out any problem. After that point the values given by several functions were confusing and the only way to make them work was to do a side by side comparison with the original code. Given the size of the original matlab code, this did not seem possible in the remaining time of the project. So we

present the result from partial iterations.

The script 'check.R' was written for unit testing the main iteration with out the loop. So, this was run several times to simulate real iterations until any problems occurred.

```
library(rImfil)
source("check.R")
#some output ommitted
#run repeatedly until any problems surfaced
> x
     [,1]
[1,]  0.5
[2,]  0.5
> histout
         [,1]
[1,] 1.000000
[2,] 1.666667
[3,] 0.000000
[4,] 0.000000
[5,] 0.000000
[6,] 0.500000
[7,] 0.500000
```

The program is designed to produce the minimizer (only x) and the history of function evaluations. It does not produce the minimized value of the function at the best point because of budget constraints.

Problems faced

The biggest problem with this coding project was that the matlab code was a huge code, all written in a single file. So, the original code had been able to use global variables, which we could not use nor was it common in any R library. There was many instance of use of matlab's special handling of matrices when their size is changing in each iteration. These operations are not common in R. The sheer size of the code made it extremely difficult and time consuming to debug. The problem had gotten worse in situations where the logic of the program is not clear from the

code itself and there is no manual. So, all in all this was a poor choice of coding project.

Future plans

rHanso is going to be submitted to CRAN. We are currently looking for some more examples of objective functions where the gain in using rHanso is clear over any other methods of optimizations. Other than that, it is a self sufficient library with completed documentation. About rImfil, the plan is to integrate it with 'dfoptim' package. It would depend on the performance of the program in real functions.