Manuscript JSS2717
"sparseHessianFD: An R Package for Estimating Sparse Hessian Matrices"

Response to reviewers

I would like to express my appreciation for the time and effort that the reviewers invested in their reviews. The comments and suggestions are all well-taken, and I have addressed all of the items, point by point, in this note. The reviewers comments are in italics, and my responses follow.

0. *I had a minor problem checking the package tarball:*

   (error messages related to algorithm.sty removed).

   I don't remember if there is an easy way to distribute "special" sty files, or if you must force the user to install them. If the sty is not really critical it might be better to omit it.

   I suspect that the error is a result of the *algorithm* package not being present in your LaTeX installation. I would like to continue to use it, because it does allow for nicely formatted algorithms. I believe that this is a relatively common package, and one that should be easily installed via your closest CTAN mirror. However, if there is a strong objection, I may be able to find another way to format the algorithms.

1. *I am not familiar with reference classes, but for most users I think it should not be important that the package uses them. However, as a user, I find it disconcerting that sparseHessianFD() is used as a (constructor) function (eg. p12, and in the example in ?sparseHessianFD) but is not documented in the help with 'usage' and 'arguments' as is usual for functions. There seems to be little guidance in the usual R places about how reference classes should be documented. Perhaps you could seek some guidance from the community on this point.*

2. *It is pointed out (p2) that requirements are burdensome and emphasized that the package is not appropriate for all uses. Also, the conditions 1-5 are fairly stringent. The reader is left with the impression that there may be few applications. I think the value of the package could be promoted better. Perhaps a small list of different types or applications of hierarchical problems could be given, or an indication of other classes of problems where the conditions would apply.*

3. *Condition 5 might be relaxed to a local condition in a neighborhood of the evaluation point, rather than a global condition, but I do not know if that would have any practical value.*

4. *Section 2.1 p7 I think could be made more easily readable by adding a few more hints about dimension, and more care about use of the term "coefficients vector", for example:*

   - 'continuous covariates x_i,' -¿ 'continuous covariates $x_i \in R^k$,'

   - 'heterogeneous coefficient vector β_i' -¿ 'heterogeneous coefficient vector $\beta_i \in R^k$'

   - 'The coefficients are distributed' -¿ 'The coefficient vectors β_i are distributed'

5. *p7 following "the cross-partial derivatives" $\mathsf{H}_{\beta_i,\beta_j} = \mathsf{D}^2_{\beta_i,\beta_j} = 0$ for all $i \neq j$.*

   Is this a definition of $\mathsf{H}_{\beta_i,\beta_j}$ in terms of D^2 or a statement of equality?

   Also, I am confused by the single subscript β_i to hess here, but a double subscript just above in "Thus, $\mathsf{H}_{\beta_{ik},\mu_k} \neq 0$".

6. *I think possibly (16) is "banded" and (17) "block arrow" rather than the reverse which is indicated.*

7. *Code in the paper prior to table 4 needs to be copied from the vinettes/sparseHessianFD.Rnw file. For those trying to reproduce results it would be nice if this where mentioned in the file replication.R.*

8. *p9, last line. The R¿ at the beginning of*

   R¿ obj ¡- sparseHessianFD(x, fn, gr, rows, cols, ...)

   suggests that this is code that can be entire at the command line, but ... causes an error when entered (and the arguments have not been defined at this point in the vignette. Instead it should be indicated as the usage syntax.

9. *p10. "where ... represents all other named arguments" I think the usual usage is the ... represents the arguments other than the "named" ones, so it is probably better to just say "other arguments".*

10. *On p12, the function calls all.equal(f, true.f) and all.equal(gr, true.grad) are comparing f and gr calculated with the exact calculation, so the difference is zero:*

```
print(obj$fn(P) - binary.f(P,
      data=binary, priors=priors, order.row=order.row), digits=20)
```

```
[1] 0

print(max(abs(obj$gr(P) - binary.grad(P,
         data=binary, priors=priors,
order.row=order.row))),digits=20)
[1] 0
```

It is not clear to me whether $objfn() and objgr()$ use code as in the true functions or a modified version using sparse techniques. Some further clarification would be helpful.

On the other hand, all.equal(hs, true.hess) is comparing a true analytic calculation with a first order simple difference aproximation using the true gradient function:

```
max(abs(  obj$hessian(P)
     - binary.hess(P, data=binary, priors=priors, order.row=order.row)))
[1] 2.786891e-06
```

which might also be mentioned in the text. (Really just for exposition purposes, after all, it is almost the main purpose of the package.) p10. "where ... represents all other named arguments" I think the usual usage is the ... represents the arguments other than the "named" ones, so it is probably better to just say "other arguments".

11. *I think it would be instructive to add some of the following comparisons with the above on p12. The package numDeriv function hessian by default does a second order Richardson approximation using the true function value approximation. This involves a very large number of function evaluations in an attempt to obtain some accuracy, but the accuracy is limited by being an approximation of a second difference:*

```
max(abs(
     numDeriv::hessian( binary.f, P, method="Richardson",
                    data=binary, priors=priors,
order.row=order.row)
   - binary.hess(P, data=binary, priors=priors, order.row=order.row)))
[1] 0.0001610595
```

Since the hessian is the first difference of the gradient, which is the calculation used by obj$hessian() in sparseHessianFD, one could also use the function numDeriv::jacobian:

```
max(abs(
    numDeriv::jacobian( binary.grad, P, method="Richardson",
                   data=binary, priors=priors,
order.row=order.row)
  - binary.hess(P, data=binary, priors=priors, order.row=order.row)))
[1] 3.268224e-07
```

This is still doing the calculation intensive Richardson approximation. The calculation which would seem to most closely resemble what is done by obj$hessian() is

```
max(abs(
    numDeriv::jacobian( binary.grad, P, method="simple",
                   data=binary, priors=priors,
order.row=order.row)
  - binary.hess(P, data=binary, priors=priors, order.row=order.row)))

[1] 0.0008255852
```

Another very interesting comparison is

```
max(abs(
  numDeriv::jacobian( binary.grad, P, method="complex",
                  data=binary, priors=priors,
order.row=order.row)
 - binary.hess(P, data=binary, priors=priors, order.row=order.row)))

[1] 7.105427e-15
```

The complex step derivative provides extremely accurate approximations with a number of function evaluation similar to the simple method. (This does not seem to be anticipated by footnote 1 in the paper.) However, the method imposes some serious requirements on the function. (Something like complex analytic even though the user may only be interested in the real part.) The code also has to accept complex arguments and return the complex result. Fortunately most R primitive work with complex numbers so the code requirement may happen accidentally, which can be partly verified by

binary.grad(P + 0+1i, data=binary, priors=priors, order.row=order.row)

returning a complex result. (This does not rule out all possible problems.)

As I recall, sums, multiplication, and exponentiation are all complex analytic, so it would not be too surprising if the example in the paper is too, but I have not analyzed that. However, based on the result being very good, it seems highly likely.

12. *A possible extension to the package would be to implement the complex method in the sparse code. The function numDeriv:::jacobian.default implements both simple and complex, so provides a good comparison of the necessary (non-sparse) computation.*

13. *p.13 l. -7 Figure 3b should be Figure 3c*

14. *The file replication.R does not set the RNG seed. This may not be too important if only times are generated, but will be if resulting values are included.*

15. *Table 4 and 5. Some OS, processor, and memory details are helpful to put timing results in context.*

16. *Table 4 is really not the proper comparison. I think a comparison with numDeriv::jacobian(binary.grad, P, method="simple", data=binary, priors=priors, order.row=order.row) really serves to highlight the improvement of the sparse calculation because it is a valid comparison. Even though the results are not as exaggerated, they are still important: (My laptop is a Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz, 4GB RAM, SSD swap, running Mint variant of Ubuntu 14.04.2 LTS.)*

```
run.par <- FALSE

(Note this is using obj$gr() rather than binary.grad(). I am not sure if
that makes a difference.)

run_test_tab4b <- function(Nk, reps=50) {
    ## Replication function like Table 4  with jacobian of grad and
"simple"
    N <- as.numeric(Nk["N"])
    k <- as.numeric(Nk["k"])
    data <- binary_sim(N, k, T=20)
    priors <- priors_sim(k)
    F <- make_funcs(D=data, priors=priors)
    nvars <- N*k+k
    M <-
as(Matrix::kronecker(Matrix::Diagonal(N),Matrix(1,k,k)),"nMatrix") %>%
      rBind(Matrix(TRUE,k,N*k)) %>%
      cBind(Matrix(TRUE, k*(N+1), k)) %>%
      as("nMatrix")
    pat <- Matrix.to.Coord(tril(M))
    X <- rnorm(nvars)
    obj <- sparseHessianFD(X, F$fn, F$gr, pat$rows, pat$cols)

    bench <- microbenchmark(
        numDeriv = numDeriv::jacobian(obj$gr, X, method="simple"),
        df = obj$gr(X),
        sparse = obj$hessian(X))
    vals <- plyr::ddply(data.frame(bench), "expr",
                  function(x)
return(data.frame(expr=x$expr,
time=x$time,

rep=1:length(x$expr))))
    res <- data.frame(N=N, k=k,
                      bench=vals)
    cat("Completed N = ",N,"\tk = " , k ,"\n")
    return(res)
}

cases_tab4b <- expand.grid(k=c(2,3,4),
                     N=c(9, 12, 15))
runs_tab4b <- plyr::adply(cases_tab4b, 1, run_test_tab4b, reps=20,
.parallel=run.par)

tab4b <-  mutate(runs_tab4b, ms=bench.time/1000000) %>%
  select(-bench.time) %>%
  spread(bench.expr, ms) %>%
  gather(method, hessian, c(numDeriv, sparse)) %>%
  mutate(M=N*k+k, hessian.df=hessian/df) %>%
  gather(stat, time, c(hessian, hessian.df)) %>%
  group_by(N, k, method, M, stat)  %>%
  summarize(mean=mean(time), sd=sd(time)) %>%
  gather(stat2, value, mean:sd) %>%
  dcast(N+k+M~stat+method+stat2,value.var="value") %>%
  arrange(M)

tab4b
   N k  M hessian_numDeriv_mean hessian_numDeriv_sd hessian_sparse_mean
1  9 2 20              5.731754
0.6759421            1.713736
2 12 2 26              7.577777
0.9481714            1.756165
3  9 3 30              8.763990
1.0358481            2.348153
4 15 2 32              9.553452
1.1875816            1.844192
5 12 3 39             11.583360
1.3371329            2.402663
6  9 4 40             11.584188
1.2045539            2.978028
7 15 3 48             14.638763
1.5469615            2.547145
8 12 4 52             15.612216
2.5401431            3.086010
9 15 4 64             19.672496
2.7303771            3.055931
  hessian_sparse_sd hessian.df_numDeriv_mean hessian.df_numDeriv_sd
1       0.07532715
```

```
19.02359                3.353629
2        0.07522725
24.31841                4.715941
3        0.15790507
28.47628                4.366788
4        0.30441636
30.52029                4.533416
5        0.08570303
35.98930                6.033750
6        0.55389180
36.30998                6.007112
7        0.61666543
44.46145                6.206399
8        0.69277761
48.73135                9.180011
9        0.12617835
60.27624                9.530503
  hessian.df_sparse_mean hessian.df_sparse_sd
1              5.687322            0.7812959
2              5.619639            0.7445722
3              7.615163            0.7901088
4              5.884283            1.0339672
5              7.455115            0.8976474
6              9.329878            2.0531265
7              7.734390            2.0592415
8              9.637077            2.4535258
9              9.356261            0.6168655
```

17. *It is possible to do a larger example with this comparison:*

    (On my laptop the next took about 30 hrs of which 24 was for the last, N=2500, k=8 comparison.)

```
cases_tab4b5 <- expand.grid(k=c(2,4,8),
                     N=c(10, 100, 1000, 2500))
runs_tab4b5 <- plyr::adply(cases_tab4b5, 1, run_test_tab4b, reps=20,
.parallel=run.par)

tab4b5 <-  mutate(runs_tab4b5, ms=bench.time/1000000) %>%
  select(-bench.time) %>%
  spread(bench.expr, ms) %>%
  gather(method, hessian, c(numDeriv, sparse)) %>%
  mutate(M=N*k+k, hessian.df=hessian/df) %>%
  gather(stat, time, c(hessian, hessian.df)) %>%
  group_by(N, k, method, M, stat)  %>%
  summarize(mean=mean(time), sd=sd(time)) %>%
  gather(stat2, value, mean:sd) %>%
  dcast(N+k+M~stat+method+stat2,value.var="value") %>%
  arrange(M)

tab4b5
tab4b5
      N k     M hessian_numDeriv_mean hessian_numDeriv_sd
hessian_sparse_mean
1    10 2    22          6.363689e+00
6.585771e-01           1.795344
2    10 4    44          1.273715e+01
1.254662e+00           2.934588
3    10 8    88          2.608751e+01
1.803375e+00           5.659668
4   100 2   202          1.021658e+02
3.439251e+00           2.938642
5   100 4   404          2.162431e+02
1.289940e+01           5.622877
6   100 8   808          4.846450e+02
1.946854e+01          12.922397
7  1000 2  2002          5.511146e+03
1.918534e+02          14.694924
8  1000 4  4004          1.194815e+04
1.499518e+02          29.332463
9  2500 2  5002          3.132878e+04
4.934896e+02          33.393927
10 1000 8  8008          2.835954e+04
6.959400e+02          76.881070
11 2500 4 10004          6.874300e+04
2.197321e+02          68.318820
12 2500 8 20008          8.615465e+05
9.316236e+04        1806.766254
   hessian_sparse_sd hessian.df_numDeriv_mean hessian.df_numDeriv_sd
1       6.056685e-01
21.19760              2.433792
2       5.508396e-01
41.16514              5.648203
3       1.265859e+00
79.97503              8.748075
4       7.409767e-02
193.27447             11.790089
5       1.210575e+00
388.74262             31.184602
6       1.168307e+01
758.39623            121.225037
7       9.293258e-01
2042.30264            168.538055
8       1.343366e+00
```

```
4099.05785             221.910206
9       1.377541e+00
5081.52982             473.962083
10      2.408524e+01
8323.91140             516.709041
11      2.646135e+00
10315.16422             832.305746
12      2.272944e+03
77200.22397           42790.360882
   hessian.df_sparse_mean hessian.df_sparse_sd
1                5.984750            2.1093324
2                9.479244            1.9844323
3               17.337516            4.1347369
4                5.556561            0.2851630
5               10.115250            2.3189536
6               20.160163           17.8116929
7                5.448765            0.5700265
8               10.061368            0.6629470
9                5.416287            0.5421972
10              22.563990            7.1995778
11              10.249990            0.8971753
12             151.538778          217.2031623
```

```
I think the complex step takes a similar amount of time, but produces a more
accurate result.
```

18. *p.16 l. -7 "to to compute" -¿ "to compute"*

19. *While that package is useful, and reasonably demonstrated in the paper, I think it would be nice to expand the paper in some ways that might be deemed more "original research". Some possibilities are:*

    - Explain and try to assess how much of the speedup is due to simple sparseness and how much is due to the "sparse patern" (p2) allowing for perturbing multiple elements together. (I think these are related but slightly different?)

    - Try to assess how much of the speedup is due to reduce computing demand and how much is due to different memory demand. (I had the impression in the larger problems with numDeriv that my computer started to use swap space, which resulted in a big slow down.)

    - Consider implementing a complex step method, and do a comparison.

    - Assess the difference when multiple CPUs are used. (run.par == FALSE vs run.par == TRUE)

20.

21.

22.

23.

24.

25.