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

Response to reviewer

I would like to express my sincere appreciation for the time and effort that you invested in the review. It is obvious to me that you went through the paper and code in great detail. In particular, thank you for bringing the complex step method to my attention. I *never* use complex numbers or functions in my own work, so I typically ignore anything with the word "complex" in it. But this review prompted me to learn something new, and that is always fun.

I am happy to write that I was able to incorporate a large majority of your suggestions into the the revised paper and package. For the others, I hope that you, and the editor, will accept my explanations and excuses. Of course, if there are any strong objections to any of my responses, I am happy to consider further changes in advance of publication.

Below are my responses to each item, in the same order as in the review. The original comment is in italics, and is followed by my response in normal font. To save space, I removed the code and output that were included in the review.

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.

   Because *sparseHessianFD* passes CRAN check, I suspect that the error is a result of the *algorithm* package not being present in your LaTeX installation. It is a standard package in the MacTeX distribution, but I cannot speak with authority about other platforms. It should be easily installed through CTAN.

   I would prefer to continue to use the *algorithm* package, but I could try to find another way to format the algorithms if there is a strong objection.

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.*

   The help page for *sparseHessianFD* now includes explicit instructions for the initializer. You are correct that there is little guidance in how to document reference classes, and I agree that there should be a way for the existing mechanism to include documentation for the initializer to an RC. But it's hard to do it using Roxygen2 in way that also passes CRAN checks. So I just "hard coded" a description of the initializer into the details section of the sparseHessianFD-class documentation. I could not find a more elegant way of doing it.

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.

I included this section primarily as a way to avoid overselling the method, but the point that the reader might be scared away by this list of restrictions is well-taken. In the revision, I removed the list, and wove the restrictions into the text. The revision projects a more positive tone that in the original submission.

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.*

The list of conditions is no longer in the paper, but to this point, the condition is global and not local. I added a defintion of a "structural zero" in the third paragraph. The sparsity pattern represents the structure of the Hessian, not its value, so any element that *could* be zero, but does not have to be zero, is not a structural zero.

At the end of Section 2.2, I added a paragraph that explains that if there is uncertainty about whether an element is a structural zero or not, it is better to include it in the sparsity pattern.

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 $x_i \in \mathbb{R}^k$,'*
- *'heterogeneous coefficient vector β_i' -> 'heterogeneous coefficient vector $\beta_i \in \mathbb{R}^k$'*
- *'The coefficients are distributed' -> 'The coefficient vectors β_i are distributed'*

Thank you for the suggestions. Changes have been made throughout the paper.

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, $\mathsf{H}_{\beta_{ik},\mu_k} \neq 0$".

Based on this comment, I rewrote Section 2.2 to distinguish the cross-partial derivatives D^2 from the corresponding elements of the Hessian H. I did this by defining x_i as a subset of the elements of x, and $\text{Ind}(x_i)$ as an "indexing" function. This change required me to redefine the covariate matrix in the example from $\mathbf{X}$ to $\mathbf{Z}$.

The revised version is slightly more formal and precise, and I hope that this is the kind of change you had in mind.

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

The descriptions of the two sparsity patterns are correct. The first index in the subscript of β refers to a household, and the second index refers to a covariate. Thus, $\mathsf{D}^2_{\beta_{11},\beta_{12}} \neq 0$, but $\mathsf{D}^2_{\beta_{11},\beta_{21}} = 0$. When the second index changes faster than the first, the pattern is "block-arrow." When the first index changes faster than the second, the pattern is "banded."

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.*

Assuming that the R package builder works the way I think it does, the `replication.R` file, and the data for Table 4 (`vignette_tab4.Rdata`), will be copied to the `doc\` directory of the package once it is installed. At the end of the second paragraph of Section 4, I now write that "code to replicate Table 4 is available as an online supplement to this paper." I cannot be more specific about the name of the replication file, because I believe it depends on the volume and issue number of the published version.

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.

I removed the `R>` prompt, and now refer to that code as a usage syntax. Thank you for the suggestion.

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".*

Done.

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:*

(Code removed)

It is not clear to me whether obj$fn() and obj$gr() 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:

(Code removed)

which might also be mentioned in the text. (Really just for exposition purposes, after all, it is almost the main purpose of the package.)

Good point. I revised that section to explain that evaluations of the function and gradient, as called from the *sparseHessianFD* object, have to be identical to the true values. I replaced `all.equal` with `identical` to highlight that fact. I then explain that the *sparseHessianFD* calculation of the Hessian will not be identical to the true value, and report the mean relative difference. I chose the mean relative difference over maximum absolute difference to be consistent with what `all.equal` uses to test if two matrices are equal within numeric tolerance.

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:

(Code removed)

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:

(Code removed)

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

(Code removed)

Another very interesting comparison is

(Code removed)

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.

See response to Item 12

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.*

Here I am combining my responses to items 11 and 12 together.

The complex step function is now implemented in the package, and is introduced in Section 2.5 of the paper. You are correct that "most R primitives work with complex numbers," but there are many commonly used functions that do not, including `log1p`, `gamma`, and the probability distribution functions. This limitation, combined with the requirement that the function be holomorphic, and my own inexperience with the complex step method, make me hesitant to promote it too heavily in the paper. But it is a nifty trick, and I am happy that it is now included in the package.

In terms of accuracy, the paper now includes comparisons between the true Hessian and the two *sparseHessianFD* methods (finite differences and complex step). To maintain the focus of the paper on *sparseHessianFD*, I am not including an assessment of the accuracy of *numDeriv*. However, for the timing comparisons in Table 4, I did change the baseline *numDeriv* method from `hessian` with the `Richardson` method, to `jacobian` with both the `simple` and `complex` methods. Removing the second-order approximation saves a lot of time, and lets me compute Table 4 without resorting to parallel computation. That will simplify attempts at replication, and removes another variable that might influence results.

Footnote 1 was expanded to acknowledge the complex step method.

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

Fixed. Thanks.

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.*

I added a `set.seed` statement near the top of `replication.R`, but you are correct that because only times are reported, this does not matter much.

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

This information is now included in the first paragraph of Section 4.

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.)

(Code and output removed)

I agree. See my response to Item 12. Table 4 includes times from the *numDeriv* `complex` and `simple` methods.

To reduce clutter in Table 4, I removed comparisons of the hessian-to-gradient computation time ratio. That was not a particularly interesting metric.

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.)

(Code and output removed)

Because the `numDeriv::jacobian` function runs so much faster than `numDeriv::hessian`, I was able to increase the size of the examples in Table 4. One objective of Table 4 is to compare the relative increases in computation time for different N and k between *numDeriv* and *sparseHessianFD*, and I think that objective is met with $N = (15, 50, 100, 500)$. Please forgive me for not going as high as $N = 2500$, $k = 8$. For the reader who wants to replicate the results, a run time of more than a day is a lot to ask.

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

Fixed. Thanks.

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)*

These are all reasonable suggestions. Let me address them point by point.

- It really is all about the sparsity pattern, and specifically, the how few partitions are needed to group the variables. Of course, it will be hard to keep the number of partitions small if there are too many non-zero elements, so the sparsity itself has some effect. The arrangement of the non-zeros is related to the ordering of the variables, but the permutation handles that during the initialization of the *sparseHessianFD* argument. For example, variables arranged in a way that the Hessian has a block-arrow pattern are permuted to get a Hessian with a banded pattern instead. This is all discussed in Section 3.1.
- I have no idea how to do this in a generalizable way. I think it has a lot to do with the architecture of the computer itself. For example, my computer has 64 GB of RAM, so memory usage is not a binding constraint, and swap space never comes into play.

  Figure 4 does break down the components of the computation into the time it takes to reserve memory (initialization), partition the variables, and compute gradients themselves. Perhaps that is the kind of comparison you had in mind.

  I do not want to ignore or dismiss this suggestion, but I am simply not clear on what is being asked. I am open to more specific recommendations.
- The package now implements the complex step method, and comparisons of accuracy and computation time are included in the paper. Thank you for this suggestion.
- Parallel computing on a shared memory architecture really muddles up timing comparisons. The *only* reason I generated Table 4 in parallel was because of the run time of the *numDeriv* `hessian` function. Now that `hessian` was replaced with `jacobian`, the baseline measures from *numDeriv* run fast enough that I can generate Table 4 in serial.

  Your point that timing comparisons can be affected by whether computation is being run on multiple processors is well taken, but the "serialization" of Table 4 makes the issue moot.

  Data for Figure 4 were generated in parallel because of the large number of required simulations.