Working Document for the Improvement to nls() GSOC project

Arkajyoti Bhattacharjee, Indian Institute of Technology, Kanpur
John C. Nash, University of Ottawa Heather Turner, ???

2021-5-18

Abstract

nls() is the primary nonlinear modeling tool in base R. It has a great many features, but it is about two decades old and has a number of weaknesses, as well as some gaps in documentation. This document is an ongoing record of work under the Google Summer of Code 2021 of the first author. As such it is NOT meant to be a finished academic report, but a form of extended diary of activity, issues and results.

2021-5-18

By email, a Google Meet was set up for May 19 at noon Ottawa time. JN downloaded the latest version of `R-devel.tar.gz` and unpacked it into a directory `~/vmshare/R-devel` which is shared with a VirtualBox VM of Kubuntu 20.04 (Focal Fossa), a long-term support version of Linux. Opening a terminal inside this VM, it was possible to build and run this version of R.

```
cd /media/sf_vmshare/R-devel/
./configure
make
sudo make install
[admin password]
R
```

This launched the development version of R correctly.

JN also set up this document.

Agenda for May 19

- check Meet is working
- introductions
- start linux VM install if an iso is available. May want to check that VirtualBox Guest additions is available and that the shared directory works, as these sometimes require some attention to permissions and ownership etc.
- Consider early goals to for possible nls() changes. See below.
- Set objectives for next two weeks
- Set next online meeting

Possible early goals

- Get a VM running under VirtualBox and install build tools. See https://support.rstudio.com/hc/en-us/articles/218004217-Building-R-from-source I did NOT need more than ./configure in my build, as I am not building a server version.
- Try the build. (Cheer loudly when it works!)

- Explore and document the R source for nls-related code. In particular, we want
  - to list the files that have such code or calls to it
  - to note, if possible, what each does
  - to note, in particular, where nls() solves the Gauss-Newton equations, as we will want to modify these sections to augment them to allow a Marquardt stabilization.
  - to note, in particular, where nls() computes the Jacobian and/or Hessian for the nonlinear least squares problem. In package `nlsr`, there are tools that allow an expression for the model to be parsed and processed to compute the Jacobian using symbolic or automatic derivatives. This may or may not be feasible for `nls()`. However, we may be able to improve the approximation used.
- Augment this document to record what has been done and results or problems.
- Add to the bibliography file specified. This is taken from another work, but as Rmarkdown only uses the references it needs, it will serve as a template.
- AB can ask for pointers to references to add to this document or to subsidiary documents we will create as necessary to describe parts of the work if required.

From `nlsr` vignette `nlsr-devdoc.Rmd`

5. Implementation of nonlinear least squares methods

This section is a review of approaches to solving the nonlinear least squares problem that underlies nonlinear regression modelling. In particular, we look at using a QR decomposition for the Levenberg-Marquardt stabilization of the solution of the Gauss-Newton equations.

Gauss-Newton variants

Nonlinear least squares methods are mostly founded on some or other variant of the Gauss-Newton algorithm. The function we wish to minimize is the sum of squares of the (nonlinear) residuals $r(x)$ where there are m observations (elements of r) and n parameters x. Hence the function is

$$f(x) = \sum_i ({r_i}^2)$$

Newton's method starts with an original set of parameters $x[0]$. At a given iteraion, which could be the first, we want to solve

$$x[k+1] = x[k] - H^{-1} g$$

where H is the Hessian and g is the gradient at $x[k]$. We can rewrite this as a solution, at each iteration, of

$$H\delta = -g$$

with

$$x[k+1] = x[k] + \delta$$

For the particular sum of squares, the gradient is

$$g(x) = 2 * r[k]$$

and

$$H(x) = 2(J'J + \sum_i (r_i * Z_i))$$

where J is the Jacobian (first derivatives of r w.r.t. x) and Z_i is the tensor of second derivatives of r_i w.r.t. x). Note that J' is the transpose of J.

The primary simplification of the Gauss-Newton method is to assume that the second term above is negligible. As there is a common factor of 2 on each side of the Newton iteration after the simplification of the Hessian, the Gauss-Newton iteration equation is

$$(J'J)\delta = -J'r$$

This iteration frequently fails. The approximation of the Hessian by the Jacobian inner-product is one reason, but there is also the possibility that the sum of squares function is not "quadratic" enough that the unit step reduces it. Hartley (1961) introduced a line search along delta, while (**Marq63?**) suggested replacing $J'J$ with $(J'J + \lambda * D)$ where D is a diagonal matrix intended to partially approximate the omitted portion of the Hessian.

Marquardt suggested $D = I$ (a unit matrix) or $D =$ (diagonal part of $J'J$). The former approach, when λ is large enough that the iteration is essentially

$$\delta = -g/\lambda$$

gives a version of the steepest descents algorithm. Using the diagonal of $J'J$, we have a scaled version of this (see https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm; (**Levenberg44?**) predated Marquardt, but the latter seems to have done the practical work that brought the approach to general attention.)

Nash (1977) found that on low precision machines, it was common for diagonal elements of $J'J$ to underflow. A very small modification to solve

$$(J'J + lambda * (D + phi * I)) * delta = -g$$

where *phi* is a small number avoids most of these conditions. $phi = 1$ seems to work quite well. We note that this modification would likely not have been recognized had I not been working on machines with mediochre floating-point arithmetic – a little more than 6 decimal digits of precision and no extended precision.

Choices

Both `nlsr::nlxb()` and `minpack.lm::nlsLM` use a Levenberg-Marquardt stabilization of the iteration described above, with `nlsr` using the modification involving the ϕ control parameter. The complexities of the code in `minpack.lm` are such that I have relied largely on the documentation to judge how the iteration is accomplished. `nls()` uses a straightforward Gauss-Newton iteration, but rather than form the sum of squares and cross-products, uses a QR decomposition of the matrix J that has been found by a forward difference approximation.

(Nash (1979)), solving

$$(J^T J + \lambda D)\delta = -J^T r$$

where D is some diagonal matrix and lambda is a number of modest size initially. Clearly for $\lambda = 0$ we have a Gauss-Newton method. Typically, the sum of squares of the residuals calculated at the "new" set of

parameters is used as a criterion for keeping those parameter values. If so, the size of λ is reduced. If not, we increase the size of λ and compute a new δ. Note that a new J, the expensive step in each iteration, is NOT required.

As for Gauss-Newton methods, the details of how to start, adjust and terminate the iteration lead to many variants, increased by different possibilities for specifying D. See Nash (1979).

Using matrix decompositions

In Nash (1979), the iteration equation was solved as stated. However, this involves forming the sum of squares and cross products of J, a process that loses some numerical precision. A better way to solve the linear equations is to apply the QR decomposition to the matrix J itself. However, we still need to incorporate the $lambda * I$ or $lambda * D$ adjustments. This is done by adding rows to J that are the square roots of the "pieces." We add 1 row for each diagonal element of I and each diagonal element of D.

In each iteration, we reduce the *lambda* parameter before solution. If the resulting sum of squares is not reduced, *lambda* is increased, otherwise we move to the next iteration. Various authors (including the present one) have suggested different strategies for this. My current opinion is that a "quick" increase, say a factor of 10, and a "slow" decrease, say a factor of 0.2, work quite well. However, it is important to check that lambda has not got too small or underflowed before applying the increase factor. On the other hand, it is useful to be able to set lambda = 0 in the code so that a pure Gauss-Newton method can be evaluated with the program(s). The current code `nlfb()` uses the line

```
if (lamda<1000*.Machine$double.eps) lamda<-1000*.Machine$double.eps
```

to ensure we get an increase. To force a Gauss-Newton algorithm, the controls `laminc` and `lamdec` are set to 0.

The Levenberg-Marquardt adjustment to the Gauss-Newton approach is the second major improvement of `nlsr` (and also its predecessor `nlmrt` and the package `minpack-lm`) over `nls()`.

`nls()` attempts to minimize a sum of squared residuals by a Gauss-Newton method. If we compute a Jacobian matrix J and a vector of residuals r from a vector of parameters x, then we can define a linearized problem

$$J^T J\delta = -J^T r$$

This leads to an iteration where, from a set of starting parameters x_0, we compute

$$x_{i+1} = x_i + \delta$$

This is commonly modified to use a step factor `step`

$$x_{i+1} = x_i + step * \delta$$

It is in the mechanisms to choose the size of *step* and to decide when to terminate the iteration that Gauss-Newton methods differ. Indeed, though I have tried several times, I find the very convoluted code behind `nls()` very difficult to decipher. Unfortunately, its authors have now (as far as I am aware) all ceased to maintain the code.

We could implement the methods using the equations above. However, the accumulation of inner products in $J^T J$ occasions some numerical error, and it is generally both safer and more efficient to use matrix decompositions. In particular, if we form the QR decomposition of J

$$QR = J$$

where Q is an orthonormal matrix and R is Right or Upper triangular, we can easily solve

$$R\delta = Q^T r$$

for which the solution is also the solution of the Gauss-Newton equations. But how do we get the Marquardt stabilization?

If we augment J with a square matrix $sqrt(\lambda D)$ whose diagonal elements are the square roots of λ times the diagonal elements of D, and augment the vector r with n zeros where n is the column dimension of J and D, we achieve our goal.

Typically we can use $D = 1_n$ (the identity of order n), but (**Marq63?**) showed that using the diagonal elements of $J^T J$ for D results in a useful implicit scaling of the parameters. Nash (1977) pointed out that on computers with limited arithmetic (which now are rare since the IEEE 754 standard appeared in 1985), underflow might cause a problem of very small elements in D and proposed adding $\phi 1_n$ to the diagonals of $J^T J$ before multiplying by λ in the Marquardt stabilization. This avoids some awkward cases with very little extra overhead. It is accomplished with the QR approach by appending $sqrt(\phi * \lambda)$ times a unit matrix I_n to the matrix already augmented matrix. We must also append a further n zeros to the augmented r.

======== End of extract on approaches to solving Gauss Newton equations =========

References

Hartley, H. O. 1961. "The Modified Gauss-Newton Method for Fitting of Nonlinear Regression Functions by Least Squares." *Technometrics* 3: 269–80.

Nash, John C. 1977. "Minimizing a Nonlinear Sum of Squares Function on a Small Computer." *Journal of the Institute for Mathematics and Its Applications* 19: 231–37.

———. 1979. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation.* Book. Hilger: Bristol.