Variety in the Implementation of Nonlinear Least Squares Program Codes

John C Nash, retired professor, University of Ottawa

16/02/2021

Contents

Abstract

There are many ways to structure a Gauss-Newton style nonlinear least squares program code. In organizing and documenting the nearly half-century of programs in the Nashlib collection associated with Nash (1979), the author realized that this variety could be an instructive subject for software designers.

Underlying algorithms

Gauss Newton

Hartley

Marquardt (Levenberg)

Spiral (Jones??)

Sources of implementation variety

The sources of variety in implementation include:

- programming language
- possibly operating environment features
- solver for the least squares or linear equations sub-problems
- stucture of storage for the solver, that is, compact or full
- sequential or full creation of the Jacobian and residual, since it may be done in parts
- how the Jacobian is computed or approximated
- higher level presentation of the problem to the computer, as in R's `nls` or packages `minpack.lm` and `nlsr`.

Programming language

We have a versions of the Nashlib Algorithm 23 in BASIC, Fortran, Pascal, R, ...

There may be dialects of these programming languages also.

Operating environment

?? issues of how data is provided

Solver for the least squares or linear equations sub-problems

Solution of the linear normal equations

Solution of the least squares sub-problem by matrix decomposition

- Householder
- Givens
- pivoting options
- Marquardt and Marquardt Nash options
- Matrix updating??

Storage stucture

J, J'J, If J'J, then vector form of lower triangle. ??

If the choice of approach to Gauss-Newton or Marquardt is to build the normal equations and hence the sum of squares and cross products (SSCP) matrix, we know by construnction that this is a symmetric matrix and also positive definite. In this case, we can use algorithms that specifically take advantage of both these properties, namely Algorithms 7, 8 and 9 of Nashlib. Algorithms 7 and 8 are the Cholesky decomposition and back-solution using a vector of length `n*(n+1)/2` to store just the lower triangle of the SSCP matrix. Algorithm 9 inverts this matrix *in situ*.

The original Nash (1979) Algorithm 23 (Marquardt nonlinear least squares solution) computes the SSCP matrix $J'J$ and solves the Marquardt-Nash augmented normal equations with the Cholesky approach. This was continued in the Second Edition Nash (1990) and in Nash and Walker-Smith (1987). However, in the now defunct Nash (2012) and successor Nash and Murdoch (2019), the choice has been to use a QR decomposition as described below in ???. The particular QR calculations are in these packages internal to R-base, complicating comparisons of storage, complexity and performance.

Other storage approaches.

Sequential or full Jacobian computation

We could compute a row of the Jacobian plus the corresponding residual element and process this before computing the next row etc. This means the full Jacobian does not have to be stored. In Nashlib, Algorithms 3 and 4, we used row-wise data entry in linear least squares via Givens' triangularization (QR decompostition), with the possibility of extending the QR to a singular value decomposition. Forming the SSCP matrix can also be generated row-wise as well.

Analytic or approximate Jacobian

Use of finite difference approximations??

Problem interfacing

R allows the nonlinear least squares problem to be presented via a formula for the model.

Saving storage

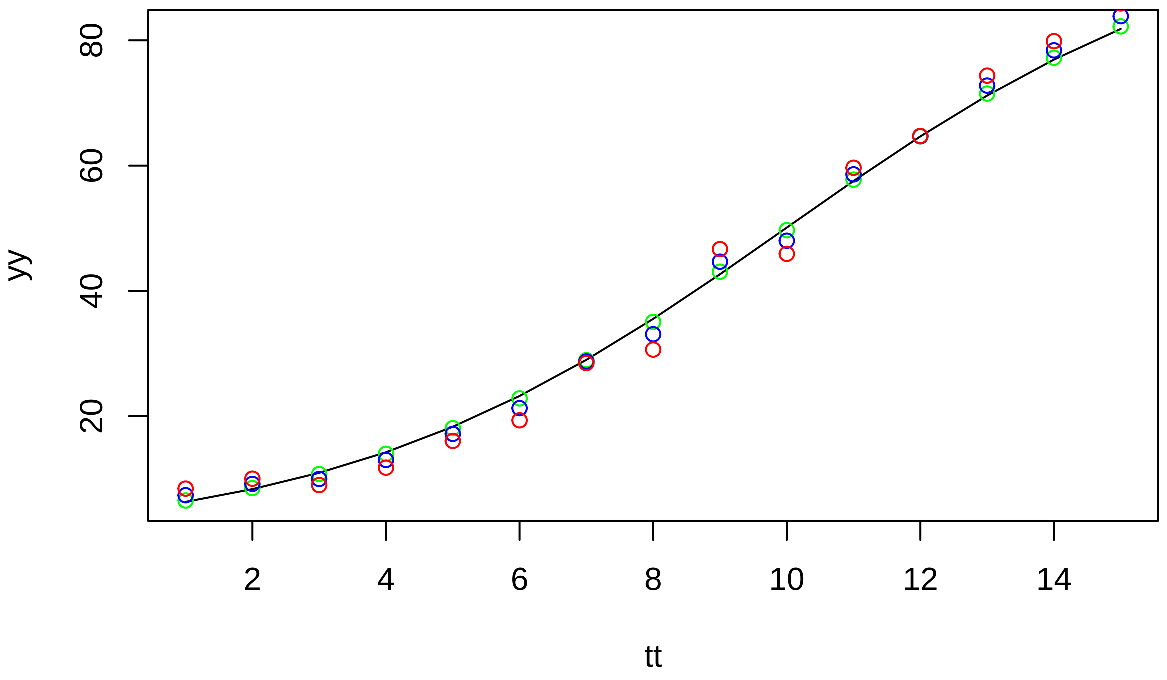
The obvious ways to reduce storage are:

- use a row-wise generation of the Jacobian in either a Givens' QR or SSCP approach. This saves space for the Jacobian as well as as well as the working matrices of the Gauss-Newton or Marquardt iterations;
- if the number of parameters to estimate is large enough, then a normal equations approach using a compact storage of the lower triangle of the SSCP matrix. However, the scale of the saving is really very small in comparison to the size of most programs.

Measuring performance

Test problems

```
# set parameters
a <- 1
b <- 2
c <- 3
np <- 15
tt <-1:np
yy <- 100*a/(1+10*b*exp(-0.1*c*tt))
plot(tt, yy, type='l')
set.seed(123456)
ev <- runif(np)
ev <- ev - mean(ev)
y1 <- yy + ev
points(tt,y1,type='p', col="green")
y2 <- yy + 5*ev
points(tt,y2,type='p', col="blue")
y3 <- yy + 10*ev
lg3d15 <- data.frame(tt, yy, y1, y2, y3)
points(tt,y3,type='p', col="red")
```



```
library(nlsr)
sol0 <- nlxb(yy ~ a0/(1+b0*exp(-c0*tt)), data=lg3d15)
```

```
## Warning in nlxb(yy ~ a0/(1 + b0 * exp(-c0 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a0', 'b0', 'c0' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol0)
```

```
## nlsr object: x
## residual sumsquares =  1.2654e-24  on  15 observations
##     after  18    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a0                  100     8.982e-13  1.113e+14  1.856e-163  -1.153e-13       718.3
## b0                   20     3.186e-13  6.277e+13  1.801e-160   7.489e-14       1.124
## c0                  0.3     3.171e-15  9.459e+13  1.312e-162   7.997e-11      0.3576
```

```
sol1 <- nlxb(y1 ~ a1/(1+b1*exp(-c1*tt)), data=lg3d15)
```

```
## Warning in nlxb(y1 ~ a1/(1 + b1 * exp(-c1 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a1', 'b1', 'c1' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol1)
```

```
## nlsr object: x
## residual sumsquares =  0.80566  on  15 observations
##     after  18    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a1              100.951        0.7311      138.1  1.397e-20  -1.814e-10       727.9
## b1              20.4393        0.2594       78.8  1.162e-17   1.692e-09       1.101
## c1             0.299971      0.002523      118.9  8.397e-20   1.715e-07      0.3505
```

```
sol2 <- nlxb(y2 ~ a2/(2+b2*exp(-c2*tt)), data=lg3d15)
```

```
## Warning in nlxb(y2 ~ a2/(2 + b2 * exp(-c2 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a2', 'b2', 'c2' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol2)
```

```
## nlsr object: x
## residual sumsquares =  20.173  on  15 observations
##     after  18    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a2               209.333         7.862      26.63  4.832e-12  -4.541e-11      764.2
## b2               44.7099         2.832      15.79  2.158e-09   4.515e-11      0.505
## c2              0.300719        0.0125      24.06  1.595e-11   3.166e-08      0.163
```

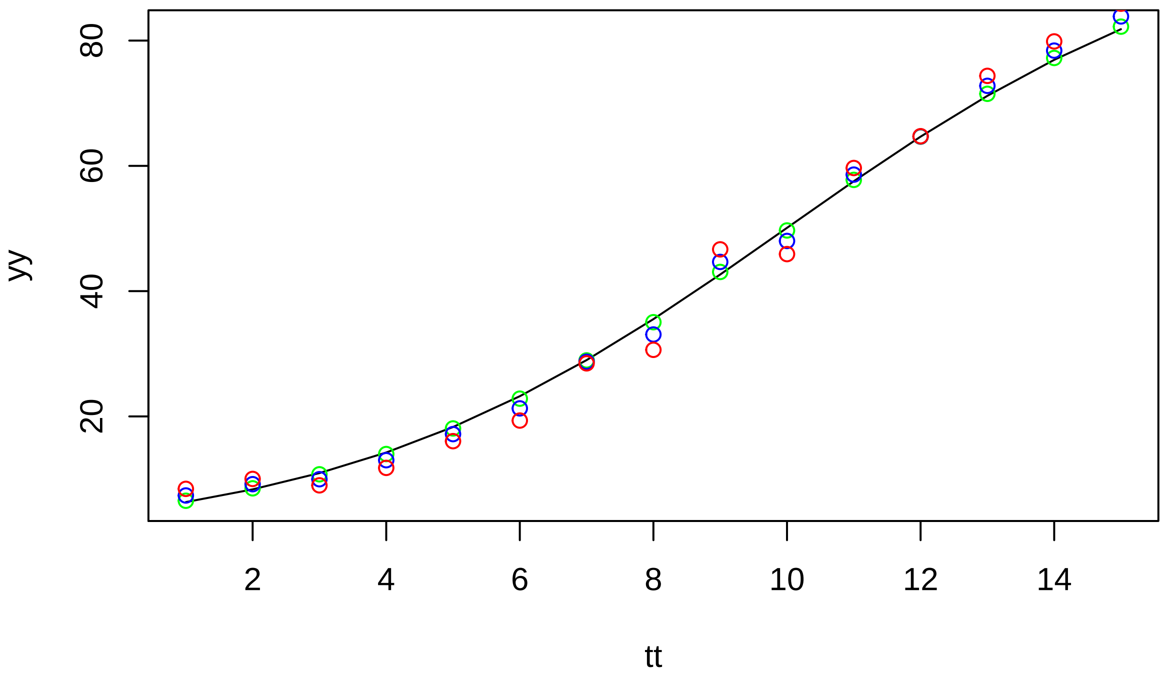
```
sol3 <- nlxb(y3 ~ a3/(3+b3*exp(-c3*tt)), data=lg3d15)
```

```
## Warning in nlxb(y3 ~ a3/(3 + b3 * exp(-c3 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a3', 'b3', 'c3' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol3)
```

```
## nlsr object: x
## residual sumsquares =  80.805  on  15 observations
##     after  19    Jacobian and  26 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a3               327.092         25.36       12.9  2.155e-08  -2.898e-11      804.1
## b3               75.4499         9.629      7.836  4.646e-06   2.191e-11     0.2989
## c3              0.303528       0.02481      12.23  3.905e-08   1.837e-08      0.101
```

```
yy <- 100*a/(1+10*b*exp(-0.1*c*tt))
plot(tt, yy, type='l')
set.seed(123456)
ev <- runif(np)
ev <- ev - mean(ev)
y1 <- yy + ev
points(tt,y1,type='p', col="green")
y2 <- yy + 5*ev
points(tt,y2,type='p', col="blue")
y3 <- yy + 10*ev
lg3d15 <- data.frame(tt, yy, y1, y2, y3)
points(tt,y3,type='p', col="red")
```



```
library(nlsr)
sol0 <- nlxb(yy ~ a0/(1+b0*exp(-c0*tt)), data=lg3d15)
```

```
## Warning in nlxb(yy ~ a0/(1 + b0 * exp(-c0 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a0', 'b0', 'c0' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol0)
```

```
## nlsr object: x
## residual sumsquares =  1.2654e-24  on  15 observations
##     after  18    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a0                  100     8.982e-13  1.113e+14  1.856e-163  -1.153e-13       718.3
## b0                   20     3.186e-13  6.277e+13  1.801e-160   7.489e-14       1.124
## c0                  0.3     3.171e-15  9.459e+13  1.312e-162   7.997e-11      0.3576
```

```
sol1 <- nlxb(y1 ~ a1/(1+b1*exp(-c1*tt)), data=lg3d15)
```

```
## Warning in nlxb(y1 ~ a1/(1 + b1 * exp(-c1 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a1', 'b1', 'c1' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol1)
```

```
## nlsr object: x
## residual sumsquares =  0.80566  on  15 observations
##     after  18    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a1              100.951        0.7311      138.1  1.397e-20  -1.814e-10       727.9
## b1              20.4393        0.2594       78.8  1.162e-17   1.692e-09       1.101
## c1             0.299971      0.002523      118.9  8.397e-20   1.715e-07      0.3505
```

```
sol2 <- nlxb(y2 ~ a2/(2+b2*exp(-c2*tt)), data=lg3d15)
```

```
## Warning in nlxb(y2 ~ a2/(2 + b2 * exp(-c2 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a2', 'b2', 'c2' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol2)
```

```
## nlsr object: x
## residual sumsquares =  20.173  on  15 observations
##     after  18    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a2               209.333         7.862      26.63  4.832e-12  -4.541e-11       764.2
## b2               44.7099         2.832      15.79  2.158e-09   4.515e-11       0.505
## c2              0.300719        0.0125      24.06  1.595e-11   3.166e-08       0.163
```

```
sol3 <- nlxb(y3 ~ a3/(3+b3*exp(-c3*tt)), data=lg3d15)
```

```
## Warning in nlxb(y3 ~ a3/(3 + b3 * exp(-c3 * tt)), data = lg3d15): No starting values specified for s
## Initializing 'a3', 'b3', 'c3' to '1.'.
## Consider specifying 'start' or using a selfStart model
```

```
print(sol3)
```

```
## nlsr object: x
## residual sumsquares =  80.805  on  15 observations
##     after  19    Jacobian and  26 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a3               327.092         25.36       12.9  2.155e-08  -2.898e-11       804.1
## b3               75.4499         9.629      7.836  4.646e-06   2.191e-11      0.2989
## c3              0.303528       0.02481      12.23  3.905e-08   1.837e-08       0.101
```

```
np <- 150
tt <- (1:np)/10
yy <- 100*a/(1+10*b*exp(-0.1*c*tt))
set.seed(123456)
ev <- runif(np)
ev <- ev - mean(ev)
y1 <- yy + ev
y2 <- yy + 5*ev
y3 <- yy + 10*ev
lg3d150 <- data.frame(tt, yy, y1, y2, y3)
np <- 1500
tt <- (1:np)/100
yy <- 100*a/(1+10*b*exp(-0.1*c*tt))
set.seed(123456)
ev <- runif(np)
ev <- ev - mean(ev)
y1 <- yy + ev
y2 <- yy + 5*ev
y3 <- yy + 10*ev
lg3d1500 <- data.frame(tt, yy, y1, y2, y3)
f0 <- yy ~ a0/(1+b0*exp(-c0*tt))
f1 <- y1 ~ a1/(1+b1*exp(-c1*tt))
f2 <- y2 ~ a2/(2+b2*exp(-c2*tt))
f3 <- y3 ~ a3/(3+b3*exp(-c3*tt))
```

Implementation comparisons

Linear least squares and storage considerations

Without going into too many details, we will present the linear least squares problem as

$$Ax \simeq b$$

In this case A is an m by n matrix with $m >= n$ and b a vector of lenght m. We write **residuals** as

$$r = Ax - b$$

or as

$$r_1 = b - Ax$$

Then we wish to minimize the sum of squares $r'r$. This problem does not necessarily have a unique solution, but the **minimal length least squares solution** which is the x that has the smallest $x'x$ that also minimizes $r'r$ is unique.

The historically traditional method for solving the linear least squares problem was to form the **normal equations**

$$A'Ax = A'b$$

This was attractive to early computational workers, since while A is m by n, $A'A$ is only n by n. Unfortunately, this **sum of squares and cross-products** (SSCP) matrix can make the solution less reliable, and this is discussed with examples in Nash (1979) and Nash (1990).

Another approach is to form a QR decomposition of A, for example with Givens rotations.

$$A = QR$$

where Q is orthogonal (by construction for plane rotations) and R is upper triangular. We can rewrite our original form of the least squares problem as

$$Q'A = Q'QR = R \simeq Q'b$$

R is an upper triangular matrix R_n stacked on an $m - n$ by n matrix of zeros. But $z = Q'b$ can be thought of as n-vector z_1 stacked on $(m - n)$-vector z_2. It can easily be shown (we won't do so here) that a least squares solution is the rather easily found (by back-substitution) solution of

$$R_n x = z_1$$

and the minimal sum of squares turns out to be the cross-product $z_2' z_2$. Sometimes the elements of z_2 are called **uncorrelated residuals**. The solution for x can actually be formed in the space used to store z_1 as a further storage saving, since back-substitution forms the elements of x in reverse order.

All this is very nice, but how can we use the ideas to both avoid forming the SSCP matrix and keep our storage requirements low?

Let us think of the row-wise application of the Givens transformations, and use a working array that is $n + 1$ by $n + 1$. (We can actually add more columns if we have more than one b vector.)

Suppose we put the first $n + 1$ rows of a merged $A|b$ working matrix into this storage and apply the row-wise Givens transformations until we have an n by n upper triangular matrix in the first n rows and columns of our working array. We further want row $n+1$ to have n zeros (which is possible by simple transformations) and a single number in the $n+1$, $n+1$ position. This is the first element of z_2. We can write it out to external storage if was want to have it available, or else we can begin to accumulate the sum of squares.

We then put row $n+2$ of $[A|b]$ into the bottom row of our working storage and eliminate the first n columns of this row with Givens transformations. This gives us another element of z_2. Repeat until all the data has been processed.

We can at this point solve for x. Algorithm 4, however, applies the one-sided Jacobi method to get a singular value decomposition of A allowing of a minimal length least squares solution as well as some useful diagnostic information about the condition of our problem. This was also published as Lefkovitch and Nash (1976).

References

Lefkovitch, L. P., and John C. Nash. 1976. "Principal Components and Regression by Singular Value Decomposition on a Small Computer." *Applied Statistics* 25 (3): 210–16.

Nash, John C. 1979. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation.* Book. Hilger: Bristol.

———. 1990. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation, Second Edition.* Book. Institute of Physics : Bristol.

———. 2012. *Nlmrt: Functions for Nonlinear Least Squares Solutions.*

Nash, John C, and Duncan Murdoch. 2019. *Nlsr: Functions for Nonlinear Least Squares Solutions.*

Nash, John C., and Mary Walker-Smith. 1987. *Nonlinear Parameter Estimation: An Integrated System in Basic.* New York: Marcel Dekker.