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 John C. Nash (1979), the author realized that this variety could be an instructive subject for software designers.

Underlying algorithms

Gauss Newton

https://en.wikipedia.org/wiki/Gauss%E2%80%93Newton_algorithm

In calculus we learn that a stationary point (local maximum, minimum or saddle point) of a function f occurs where its gradient (or first derivative) is zero. In multiple dimensions, this is the same as having the gradient g stationary. The so-called Newton method uses the second derivatives in the Hessian matrix H defined as

$$H_{i,j} = \partial^2 f / \partial x_i \partial x_j$$

where f is a function of several variables x_i, where $i = 1, ..., n$, written collectively as x. The Newton equations are defined as

$$H\delta = -g$$

where H and g are evaluated at a guess or estimate of x. x is then updated to

$$x \leftarrow x + \delta$$

and we iterate until there is no change in x.

There are some adjustments When our function $f(x)$ can be written as a sum of squares

$$f(x) = \sum_{i-1}^{n} r_i(x)^2 = r'r$$

where the final vector product form is the one I favour because it is the least fussy to write. In particular, when we try to evaluate the Hessian of this sum of squares function, we see that it is

$$H_{j,k} = 2\sum_{i-1}^{n} [(\partial r_i/\partial x_j)(\partial r_i/\partial x_k) + r_i(\partial^2 r_i/\partial x_j \partial x_k)]$$

If we define the **Jacobian** matrix

$$J_{i,j} = \partial r_i/\partial x_j$$

then the gradient is

$$g = 2J'r$$

and the first part of the Hessian is

$$J'J$$

Arguing that the residuals should be "small," Gauss proposed that the Newton equations could be approximated by ignoring the second term (using elements of the residual times second derivatives of the residual). This gives the **Gauss-Newton** equations (cancelling the factor 2)

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

We can use this in an iteration similar to the Newton iteration.

Hartley's method

Conditions for convergence of the Newton iteration and the Gauss-Newton iteration are rather a nuisance to verify, and in any case we want a solution. Hartley (1961) proposed that rather than use an iteration

$$x \leftarrow x + \delta$$

one could do a search along the direction δ. Ideally we would want to minimize

$$f(x + \alpha\delta)$$

with respect to the (scalar) parameter δ. However, typically a value of α that permits a reduction in the sum of squares $f(x)$ is accepted and the iteration repeated. Clearly there are many tactical choices that give rise to a variety of particular algorithms. One concern is that the direction δ gives no lower value of the sum of squares, since there is an approximation involved in using $J'J$ rather than H.

Marquardt

Marquardt (1963) is perhaps one of the most important developments in nonlinear least squares apart from the Gauss-Newton method. There are several ways to view the method.

First, considering that the $J'J$ may be effectively singular in the computational environment at hand, the Gauss-Newton method may be unable to compute a search step that reduces the sum of squared residuals. The right hand side of the normal equations, that is, $g = -J'r$ is the gradient for the sum of squares. Thus a solution of

$$1_n\delta = -g$$

is clearly a downhill version of the gradient. And if we solve

$$\lambda 1_n\delta = -g$$

various values of λ will produce steps along the **steepest descents** direction. Solutions of the Levenberg-Marquardt equations

$$(J'J + \lambda 1_n)\delta = -J'r$$

can be thought of as yielding a step δ that merges the Gauss-Newton and steepest-descents direction. This approach was actually first suggested by Levenberg (1944), but it is my opinion that while the idea is sound, Levenberg very likely never tested them in practice. Before computers were commonly available, this was not unusual, and many papers were written with computational ideas that were not tested or were given only cursory trial.

Practical Marquardt methods require us to specify an initial λ and ways to adjust its value. For example, one could set an initial λ of 0.0001 and reduce it by multiplying it by 0.4 whenever a trial x has a lower value of $f = r'r$ and is accepted to evaluate J and g for the next iteration.

When a trial \$x5 is not "lower," we do NOT accept it, but keep J and g but increase λ by multiplying it by 10 before again computing a new trial x.

The increase in λ is repeated until there is either a "success" or the trial x is unchanged from the current one, which implies the algorithm has converged.

In this scheme, an algorithm is defined by the initial value, decrease factor and increase factor for λ, but there are a number of annoying computational details concerning underflow or overflow and how we measure equivalence of iterates.

Others

There have been many proposed approaches to nonlinear least squares.

Spiral

Jones (1970) is a method ...

Sources of implementation variety

The sources of variety in implementation include:

- programming language
- possible 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, and R, with Python pending. There may be dialects of these programming languages also, giving rise to other variations.

Operating environment

Generally, for modern computers, the operating system and its localization do not have much influence on the way in which we set up nonlinear least squares computations. I will comment below about some issues where speed and size of data storage may favour some approaches over others. However, such issues are less prominent today.

Solver for the least squares or linear equations sub-problems

Solution of the linear normal equations

The Gauss-Newton or Marquardt eqations are a set of linear equations. Moreover, the coefficient matrix is non-negative definite and symmetric. Thus it permits of both general and specialized methods for linear equations. Furthermore, one can also set up the solution without forming the $J'J$ matrix by using several matrix decomposition methods. Thus there are many possible procedures.

- Gauss elimination with partial pivoting
- Gauss elimination with complete pivoting
- Variants of Gauss elimination that build matrix decompositions
- Gauss-Jordan inversion
- Choleski Decomposition and back substitution
- Eigendecompositions of the SSCP

Solution of the least squares sub-problem by matrix decomposition

- Householder
- Givens
- pivoting options
- Marquardt and Marquardt Nash options
- SVD approaches

Avoiding duplication of effort when increasing the λ parameter

How to do this??

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 John C. 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 John C. Nash (1990) and in John C. Nash and Walker-Smith (1987). However, in the now defunct John C. Nash (2016) and successor John C. 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:

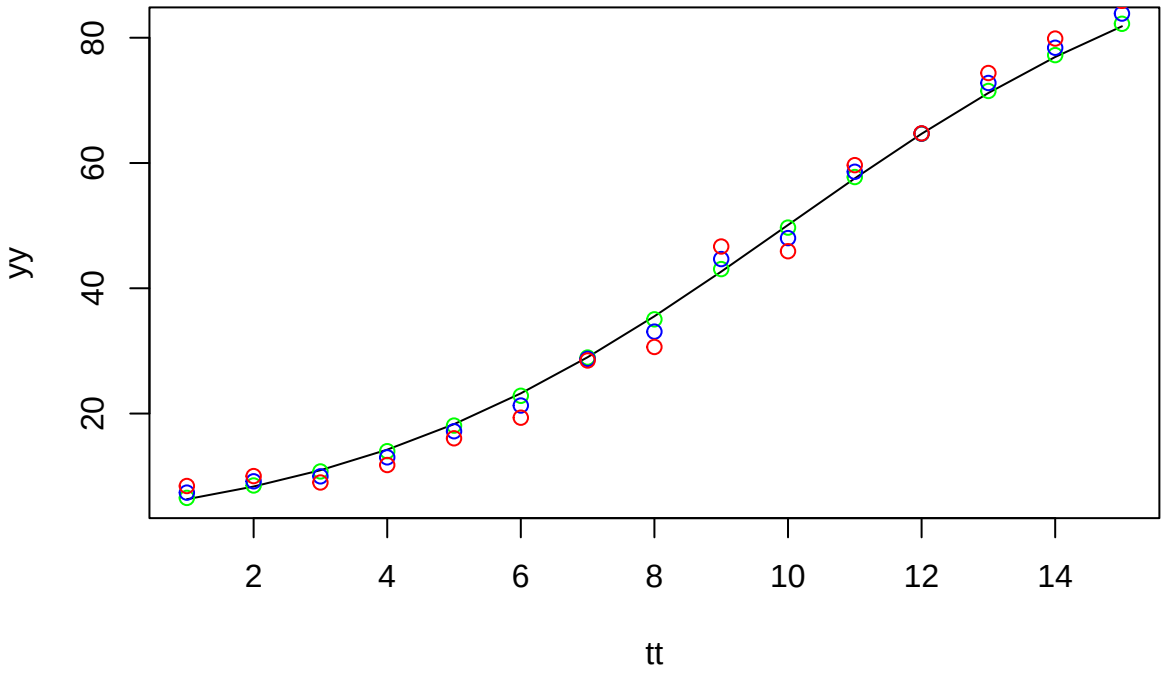
- use a row-wise generation of the Jacobian in either a Givens' QR or SSCP approach. This saves space for the Jacobian as well as as well as the working matrices of the Gauss-Newton or Marquardt iterations;

- 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
set.seed(123456)
a <- 1
b <- 2
c <- 3
np <- 15
tt <-1:np
yy <- 100*a/(1+10*b*exp(-0.1*c*tt))
plot.new()
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, start=list(a0=1, b0=1, c0=1))
```

```
## vn:[1] "yy" "a0" "b0" "c0" "tt"
## no weights
```

```
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, start=list(a1=1, b1=1, c1=1))
```

```
## vn:[1] "y1" "a1" "b1" "c1" "tt"
## no weights
```

```
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, start=list(a2=1, b2=1, c2=1))
```

```
## vn:[1] "y2" "a2" "b2" "c2" "tt"
## no weights
```

```
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, start=list(a3=1, b3=1, c3=1))
```

```
## vn:[1] "y3" "a3" "b3" "c3" "tt"
## no weights
```

```
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
```

The following is a larger dataset version of this test.

```
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

Here want to explore the ideas.

Linear least squares and storage considerations

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

$$Ax \tilde{=} b$$

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.

Let us set up a simple problem in R:

```
# simple linear least squares examples
v <- 1:6
v2 <- v^2
vx <- v+5
one <- rep(1,6)
```

```
Ad <- data.frame(one, v, v2)
A <- as.matrix(Ad)
print(A)
```

```
##      one v v2
## [1,]   1 1  1
## [2,]   1 2  4
## [3,]   1 3  9
## [4,]   1 4 16
## [5,]   1 5 25
## [6,]   1 6 36
```

```
Ax <- as.matrix(data.frame(one, v, vx, v2))
print(Ax)
```

```
##      one v vx v2
## [1,]   1 1  6  1
## [2,]   1 2  7  4
## [3,]   1 3  8  9
## [4,]   1 4  9 16
## [5,]   1 5 10 25
## [6,]   1 6 11 36
```

```
y <- -3 + v + v2
print(y)
```

```
## [1] -1  3  9 17 27 39
```

```
set.seed(12345)
ee <- rnorm(6)
ee <- ee - mean(ee)
ye <- y + 0.5*ee
print(ye)
```

```
## [1] -0.66725  3.39472  8.98534 16.81324 27.34293 38.13101
```

```
sol1 <- lm.fit(A, y)
print(sol1)
```

```
## $coefficients
## one   v  v2
##  -3   1   1
##
## $residuals
## [1] -1.4580e-16 -3.7027e-16  1.5848e-15 -1.0748e-15 -3.9479e-16  4.0082e-16
##
## $effects
##         one           v          v2
## -3.8375e+01  3.3466e+01  6.1101e+00 -1.7764e-15 -8.8818e-16  4.4409e-16
##
## $rank
## [1] 3
##
## $fitted.values
## [1] -1  3  9 17 27 39
##
## $assign
```

```
## NULL
##
## $qr
## $qr
##           one         v        v2
## [1,] -2.44949 -8.573214 -37.15059
## [2,]  0.40825  4.183300  29.28310
## [3,]  0.40825 -0.053724   6.11010
## [4,]  0.40825 -0.292770   0.66060
## [5,]  0.40825 -0.531816   0.38717
## [6,]  0.40825 -0.770861  -0.21358
##
## $qraux
## [1] 1.4082 1.1853 1.6067
##
## $pivot
## [1] 1 2 3
##
## $tol
## [1] 1e-07
##
## $rank
## [1] 3
##
## attr(,"class")
## [1] "qr"
##
## $df.residual
## [1] 3
```

```
cat("Residual SS=",as.numeric(crossprod(sol1$residuals)),"\n")
```

```
## Residual SS= 4.1415e-30
```

```
sol1e <- lm.fit(A,ye)
print(sol1e)
```

```
## $coefficients
##      one        v       v2
## -2.80191  1.14561  0.95334
##
## $residuals
## [1]  0.035714  0.092063 -0.229614 -0.220678  0.583375 -0.260860
##
## $effects
##       one         v        v2
## -38.37534  32.70908   5.82498  -0.11371   0.66662  -0.24948
##
## $rank
## [1] 3
##
## $fitted.values
## [1] -0.70296  3.30266  9.21495 17.03392 26.75956 38.39187
##
## $assign
```

```
## NULL
##
## $qr
## $qr
##           one         v        v2
## [1,] -2.44949 -8.573214 -37.15059
## [2,]  0.40825  4.183300  29.28310
## [3,]  0.40825 -0.053724   6.11010
## [4,]  0.40825 -0.292770   0.66060
## [5,]  0.40825 -0.531816   0.38717
## [6,]  0.40825 -0.770861  -0.21358
##
## $qraux
## [1] 1.4082 1.1853 1.6067
##
## $pivot
## [1] 1 2 3
##
## $tol
## [1] 1e-07
##
## $rank
## [1] 3
##
## attr(,"class")
## [1] "qr"
##
## $df.residual
## [1] 3
```

```
crossprod(sol1e$residuals)
```

```
##         [,1]
## [1,] 0.51955
```

```
sol2<-lm.fit(Ax,y)
# Note the NA in the coefficients -- Ax is effectively singular
print(sol2)
```

```
## $coefficients
## one   v  vx  v2
##  -3   1  NA   1
##
## $residuals
## [1] -1.4580e-16 -3.7027e-16  1.5848e-15 -1.0748e-15 -3.9479e-16  4.0082e-16
##
## $effects
##         one           v          v2
## -3.8375e+01  3.3466e+01  6.1101e+00 -1.7764e-15 -8.8818e-16  4.4409e-16
##
## $rank
## [1] 3
##
## $fitted.values
## [1] -1  3  9 17 27 39
```

```
##
## $assign
## NULL
##
## $qr
## $qr
##           one         v        v2          vx
## [1,] -2.44949 -8.573214 -37.15059 -2.0821e+01
## [2,]  0.40825  4.183300  29.28310  4.1833e+00
## [3,]  0.40825 -0.053724   6.11010  2.1554e-15
## [4,]  0.40825 -0.292770   0.66060 -1.9080e-15
## [5,]  0.40825 -0.531816   0.38717 -2.0870e-01
## [6,]  0.40825 -0.770861  -0.21358 -9.6833e-01
##
## $qraux
## [1] 1.4082 1.1853 1.6067 1.1370
##
## $pivot
## [1] 1 2 4 3
##
## $tol
## [1] 1e-07
##
## $rank
## [1] 3
##
## attr(,"class")
## [1] "qr"
##
## $df.residual
## [1] 3
```

```
S2 <- sol2$coefficients
J <- which(is.na(S2))
S2[J]<-0
crossprod(S2)
```

```
##      [,1]
## [1,]   11
```

The above uses the intermediate code in function `lm.fit()`. This uses a QR solver, but it is written as a wrapper in C calling a Fortran routine (in the Fortran 77 dialect).

?? put in the structure and where the code is located

```
x <- qr.solve(A,y)
 x
one   v  v2
 -3   1   1
 xe<-qr.solve(A, ye)
> xe
       one          v         v2
-2.8019063  1.1456093  0.9533367
```

```
Z<-svd(Ax)
D1 <- 1/Z$d
print(D1)
```

```
D1[4]<-0 # to remove linear dependency (small singval)
# minimum length LS solution
minsol2 <- Z$v %*% (diag(D1) %*% (t(Z$u) %*% y))
print(minsol2)
cat("Residual SS=",as.numeric(crossprod(minsol2)),"\n")
```

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 John C. Nash (1979) and John C. 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

$$Q'A = Q'QR = R \tilde{=} Q'b$$

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

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

Jones, A. 1970. "Spiral—A new algorithm for non-linear parameter estimation using least squares." *The Computer Journal* 13 (3): 301–8.

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.

Levenberg, Kenneth. 1944. "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics* 2: 164--168.

Marquardt, Donald W. 1963. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics* 11 (2): 431–41.

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.

———. 2016. *Nlmrt: Functions for Nonlinear Least Squares Solutions*. https://CRAN.R-project.org/package=nlmrt.

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

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