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

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

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

$$g = 2J^t r$$

and the first part of the Hessian is

$$J^t J$$

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)

$$(J^t J)\delta = -J^t r$$

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 $J^t J$ rather than H.

In `nls.c`, the Gauss Newton direction is computed (`incr` – see nls-flowchart.txt) and a step-halving "line-search" is used. This is a form of the method of Hartley (1961), though Hartley recommends a quadratic (parabolic) line search where two "new" points are evaluated and we then try a third at the suggested minimum of the parabola fitted to the existing and two new sum of squares functions.

Marquardt stabilized Gauss-Newton

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 $J^t J$ 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, $g = -J^t r$ is the gradient for the sum of squares. Thus a solution of

$$\mathbf{1}_n \delta = -g$$

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

$$\lambda \mathbf{1}_n \delta = -g$$

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

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

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 ideas are 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, if

$$SS(x) = r^t r = f(x)$$

is the function to minimize to "fit" our nonlinear model:

- set an initial λ of 0.0001

- **A** Compute J and g, set up Marquardt equations and find δ
- **B** compute new set of parameters

$$x_{new} = x + \delta$$

- if $x_{new}! = x$, compute $SS(x_{new})$, else quit (**termination**)
- if \$$SS(x_{new}) < SS(x)$, replace x with x_{new}, replace λ with $0.4 * \lambda$, then retry from **A**
- ELSE increase λ to $10 * \lambda$ and retry from **B** (J and g do NOT need to be recomputed)

In this scheme, an algorithm is defined by the initial value, decrease factor (e.g., 0.4) and increase factor (e.g., 10) for λ, but there are a number of annoying computational details concerning underflow or overflow and how we measure equivalence of iterates. That is, we need to ensure λ is not so "small" that multiplying by 10 is meaningless, and we need to know what "not equal" means for floating point vectors in the arithmetic in use.

An example of choices to deal with these details:

- if λ is smaller than some pre-set tolerance, set it to the tolerance. (Otherwise do not worry if it is tiny, as long as the sum of squares still gets reduced.)
- when comparing x_{new} and $x5, we can use a modest number, e.g, 100.0 which we will call OFFSET. In the arithmetic of the system at hand, if the floating-point numbers $x_{new} + OFFSET} matches bit for bit $x + OFFSET$, then they are equal.

Many practitioners dislike using IF statements (comparisons) of floating point numbers and I have had a number of exchanges with other workers over the years. However, I have never seen any issues. When x is tiny, we compare OFFSET to itself and declare equality. When x is very large, OFFSET does not change it and we compare x with a value that is identical to it. Moreover, we use very little extra code.

Automatic scaling

In Marquardt (1963), it is shown that the use of the diagonal elements of JtJ instead of 1_n in the Marquardt stabilized Gauss-Newton equations is equivalent to having a similar scale for all the parameters. Call this diagonal matrix D. Generally this is the preferred approach to Levenberg-Marquardt stabilizations. However, in low-precision arithmetic, elements of the diagonal may underflow to zero and cause issues of singularity in the equations to be solved. A simple workaround is to slightly modify the equations to

$$(J^t J + \lambda(D + \phi 1_n)\delta = -J^t r$$

where phi is a modest number such as 1.

Note that we form JTJ in `nlfb()` to provide the scaled stabilization, which is quite a lot of computational work. The experimental package `nlsralt` has been built to add functions `nlxbx()` and `nlfbx()` that use ONLY the fixed element (i.e., unit matrix) stabilizations. A very preliminary timing ([2021-6-26]) using the Hobbs unscaled problem shows a slight improvement for the simplified version in timing with equivalent solution.

Others

There have been many proposed approaches to nonlinear least squares.

Spiral

Jones (1970) is a method …

Approaches to Newton's method

Newton (and his sometime associated worker Raphson) used the algorithm we attribute to them applied to a very special case and in a manner unrecognizable as that described above. (See https://en.wikipedia.org/wiki/Newton%27s_method).

The main attraction of the Gauss-Newton method is that the Jacobian only needs first derivatives, which are generally much, much easier to compute than the full second derivatives of the Hessian. However, there have been suggestions Newton-like approaches, possibly for special types of models where some short cut to the Hessian is available. These may or may not make sense in the context of a given user or group of users. I regard these approaches as part of the computational infrastructure of the special cases rather than as general nonlinear least squares methods.

Hybrid methods

The central idea of Hartley's method is a line search and that of Marquardt a parametrized stabilization that can be shown to reduce a sum of squares for large enough stabilization parameter. There is nothing to prevent use of a line search additional to the Marquardt stabilization, and there are, of course, several reasonable choices for line search. Also we could solve the (possibly stabilized) Gauss-Newton equations by several quite different numerical linear algebra techniques, or even approximations. In certain situations, some of these choices may be important, but here I will not pursue them further.

Sources of implementation variety

The sources of variety in implementation include:

- structuring of the algorithm, that is, how we set up and sequence the parts of the overall computation
- 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 Rˆts `nls` or packages `minpack.lm` and `nlsr`.

Algorithm structure

The R function `nls()` and also the package `minpack.lm` both set up the nonlinear least squares computation for estimating a model specified as a **formula** (an R object class) by building an `nlsModel` object, which we will label `m`. This object is a set of functions that provide the information needed to compute the residuals, jacobian, the search direction, and many other objects required for the nonlinear least squares solution as well as ancillary information used in the post-solution analysis and reporting.

In the package `nslr` (John C. Nash and Murdoch (2019)), the function `nlxb()` calls `model2rjfun()` to create functions `res()` and `jac()` that are used in `nlfb()` (the function `nls.lm()` in `minpack.lm` is very similar) to find a nonlinear least squares solution. However, post-solution information is built explicitly AFTER a solution is found.

The essential difference is that `nls()` and `minpack.lm` build a toolbox, `m`, while `nlsr` creates its required objects as they are needed for the Marquardt computations.

Issues with this structure

A major criticism of this approach is that the collection of information needed at any one time – the "environment" in which residuals and jacobians are calculated, as well as the solution methods – is not at all well-documented. If the software is to be updated or maintained, the location and purpose of the many elements that reside in different environments is difficult to determine without much effort. In my opinion, this renders the structure prone to collapse if there are infrastructure or other changes. Documentation would go a long way to correcting this problem, but even better would be clearer structuring. Separation of the problem from the method would also be useful. At the time of writing of this document, the Gauss-Newton METHOD is embedded in the functions created by `nlsModel()` for a given PROBLEM.

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.

One aspect of the computational environment that has proved important to me has been that of low-precision arithmetic. In the 1970s, it was common to have only the equivalent of 6 o 7 (decimal) digits of precision. In such environments,

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 $J^t J$ 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

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 non-negative definite. (Generally, unless we have a strict linear dependence between the columns of the Jacobian, we can say "positive definite.") If we use an SSCP form, we can apply algorithms that specifically take advantage of both these properties, including programs that store the working matrix in special formats. Some of these are Algorithms 7, 8 and 9 of Nashlib (John C. Nash (1979)). 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 $J^t J$ 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 (1987a). 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 of an augmented Jacobian. `nls()` uses a QR decomposition of the Jacobian without augmentation (referred to as the "gradient," however).

The particular QR calculations in these packages and in R-base are quite well-hidden in internal code, complicating comparisons of storage, complexity and performance. The present exposition is an attempt to provide some documentation.

Storage in Nash/Walker-Smith (1987) BASIC code

In John C. Nash and Walker-Smith (1987b), each major iteration generates a $A = J^t J$ matrix and the right hand side $-g = J^t r$ at parameters x using functions for the residuals and Jacobian. The lower triangle of A is stored as a vector of $n(n+1)/2$ elements (row-wise ordering of C). The diagonal elements are then multiplied by $(1+\lambda)$ and $\lambda * \phi$ is added to each, where λ is our Marquardt parameter and ϕ is the adjustment suggested in John C. Nash (1977) to overcome potential underflow to zero of the accumulation of the diagonal elements in low-precision arithmetic. Let us call the augmented SSCP matrix C.

Solving $C\delta = -g$, we compute the new sum of squares at $x + \delta$. If this value is greater than or equal to the current best (i.e., lowest) sum of squares, we increase λ and generate a new C matrix. Note that we do not, however, need to build a new SSCP matrix or gradient g, just make the diagonal adjustment.

The code in John C. Nash and Walker-Smith (1987b) is much more complicated than this description suggests because of

- bounds and masks on the parameters
- tests for "non-computability," where we allow the user to provide an explicit flag that the function cannot be computed at the trial parameters. This was included to avoid some of the weaknesses in evaluation of special functions, e.g., `log()` for small arguments or `exp()` for large ones, or for problem specific situations where some parameter combinations might be considered inadmissible.
- situations where rounding and truncation errors make the Cholesky decomposition return with an indication that C is "singular," at least in the particular computational arithmetic.

Approach in package `nlsr`

In John C. Nash and Murdoch (2019), the approach to the augmented Gauss-Newton equations is nominally equivalent to that in John C. Nash and Walker-Smith (1987b), except that we perform the solution using a QR decomposition, in fact a modification of the DQRDC routine from Anderson et al. (1999), which uses Householder transformations.

One of the issues with creating the SSCP matrix A is that we lose information in finite arithmetic (See John C. Nash (1979), Chapter 5). A QR approach to the traditional Gauss-Newton codes is to solve

$$J\delta = -r$$

That is, we assume a generalized inverse of J^t can be applied to both sides of the Gauss-Newton equations.

To move to the Marquardt variant of the Gauss-Newton, we need to augment J. Let us call this augmented matrix K. We will also need to augment r by an appropriate number of zeros. For the traditional Marquardt method, we need to use the square roots of the diagonal elements of $J^t J$, that is, an n by n $Z1$ where

$$Z1_{k,k} = \sqrt(\lambda)\sqrt(J^t J)_{k,k}$$

and r is extended by n zeros. For the Nash modification, we add a further n zeros to r and augment the K matrix with another n by n block $Z2$ which is a diagonal matrix where the diagonal elements are

$$Z2_{k,k} = \sqrt(\lambda)\sqrt(\phi)$$

Unfortunately, there is a need for each major iteration to compute at least the diagonal elements of the SSCP matrix. It may, in fact, be more efficient to omit the $Z1$ augmentation. Package `nlsralt`, as mentioned, does this with functions `nlxbx()` (for formula-specified models) and `nlfbx()` (for problems specified with `res()` and `jac()` functions). This gives the following preliminary example using the Hobbs weed problem.

```
source("HobbsTiming.R", echo=TRUE)
```

```
##
## > traceval <- FALSE
##
## > ydat <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192,
## +     31.443, 38.558, 50.156, 62.948, 75.995, 91.972)
##
## > tdat <- seq_along(ydat)
##
## > start1 <- c(b1 = 1, b2 = 1, b3 = 1)
##
## > eunsc <- y ~ b1/(1 + b2 * exp(-b3 * tt))
##
## > weeddata1 <- data.frame(y = ydat, tt = tdat)
##
## > library(microbenchmark)
##
## > library(nlsralt)
##
## > library(nlsr)

## Registered S3 methods overwritten by 'nlsr':
##   method          from
##   print.rjfundoc nlsralt
##   coef.nlsr      nlsralt
##   print.nlsr     nlsralt
##   summary.nlsr   nlsralt
##   predict.nlsr   nlsralt

##
## Attaching package: 'nlsr'
## The following objects are masked from 'package:nlsralt':
##
##     codeDeriv, coef.nlsr, dex, findSubexprs, fnDeriv, isCALL,
##     isMINUSONE, isONE, isZERO, model2rjfun, model2ssgrfun, modelexpr,
##     newDeriv, newSimplification, nlfb, nlsDeriv, nlsSimplify, nlxb,
##     predict.nlsr, print.nlsr, res, resgr, resss, rjfundoc,
##     summary.nlsr, sysDerivs, sysSimplifications, wrapnlsr

##
## > tnlsrh1 <- microbenchmark(nlsrh1 <- nlxb(eunsc, data = weeddata1,
## +     start = start1))
##
## > tnlsrxh1 <- microbenchmark(nlsrxh1 <- nlxbx(eunsc,
## +     data = weeddata1, start = start1))
##
## > nlsrh1
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  20    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              196.186         11.31      17.35  3.167e-08  -1.636e-09        1011
## b2              49.0916         1.688      29.08  3.284e-10  -1.656e-08      0.4605
## b3              0.31357      0.006863      45.69  5.768e-12    1.79e-06     0.04714
##
## > nlsrxh1
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  19    Jacobian and  26 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              196.186         11.31      17.35  3.167e-08  -2.899e-11        1011
## b2              49.0916         1.688      29.08  3.284e-10  -3.891e-12      0.4605
## b3              0.31357      0.006863      45.69  5.768e-12  -4.542e-11     0.04714
##
## > tnlsrh1
## Unit: milliseconds
##                                                      expr    min     lq   mean
##  nlsrh1 <- nlxb(eunsc, data = weeddata1, start = start1) 5.0218 5.1348 5.4116
##  median     uq    max neval
##  5.2026 5.3274 7.7781   100
##
## > tnlsrxh1
## Unit: milliseconds
##                                                        expr    min     lq   mean
##  nlsrxh1 <- nlxbx(eunsc, data = weeddata1, start = start1) 4.2513 4.3667 4.5636
##  median     uq    max neval
##  4.4264 4.5035 6.3988   100
```

Other storage approaches.

?? do we want to report any other approaches here? In particular, should we look at minpack.lm to see how the Marquardt stabilization is applied? This is a very complicated program, since it calls C to call Fortran, and the Fortran is layered in a way that is well-documented but non-obvious.

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. We do not anticipate that any of these approaches will offer advantages that are so great that we would choose them.

Analytic or approximate Jacobian

`nls()` and `minpack.lm` use finite difference approximations to compute the Jacobian. In the case of `minpack.lm` it seems that the standard `numericDeriv()` (from the set of routines in file `nls.R`) is used initially for the "gradient," but within the iteration, the Fortran routine lmdif.f computes its own approximations for the Jacobian, adding another layer of potential confusion.

The `nlsr` package, however, attempts to compute analytic derivatives using symbolic and analytic derivative possibilities from other parts of R. A great advantage of these is that they are for a given point, and do not suffer the issue that a step must be taken, thereby chancing the violation of a constraint or else an inadmissible set of arguments for special functions. In my experience, the use of analytic derivatives is not much faster than the finite difference approximations, and generally finite difference methods iterate as quickly as the "exact" derivative methods to the neighbourhood of a solution. It is in allowing good estimates of convergence criteria that the analytic derivatives seem to show their advantage i.e., knowing we are at a solution rather than speed in getting there.

Problem interfacing

R allows the nonlinear least squares problem to be presented via a formula for the model. This is generally very attractive to users, since they can specify their problem in ways that often accord with how they think about them. Many (most?) researchers do not write program code, so preparing residual and jacobian functions is foreign to them.

From the point of view of package or system developers, specifications as formulas pose some particular issues:

- translating a formula to a function for its actual evaluation may be tricky, since we must essentially parse the formula. R has some tools for this that we use.
- even if we can translate a formula to a function, we need to check that we have all the data and parameter inputs to enable the evaluation. Moreover, we need to check if the inputs are valid or admissible.
- checking for correctness of inputs and translations can be considered to take up most of the code in a reliable and robust package.

Saving storage

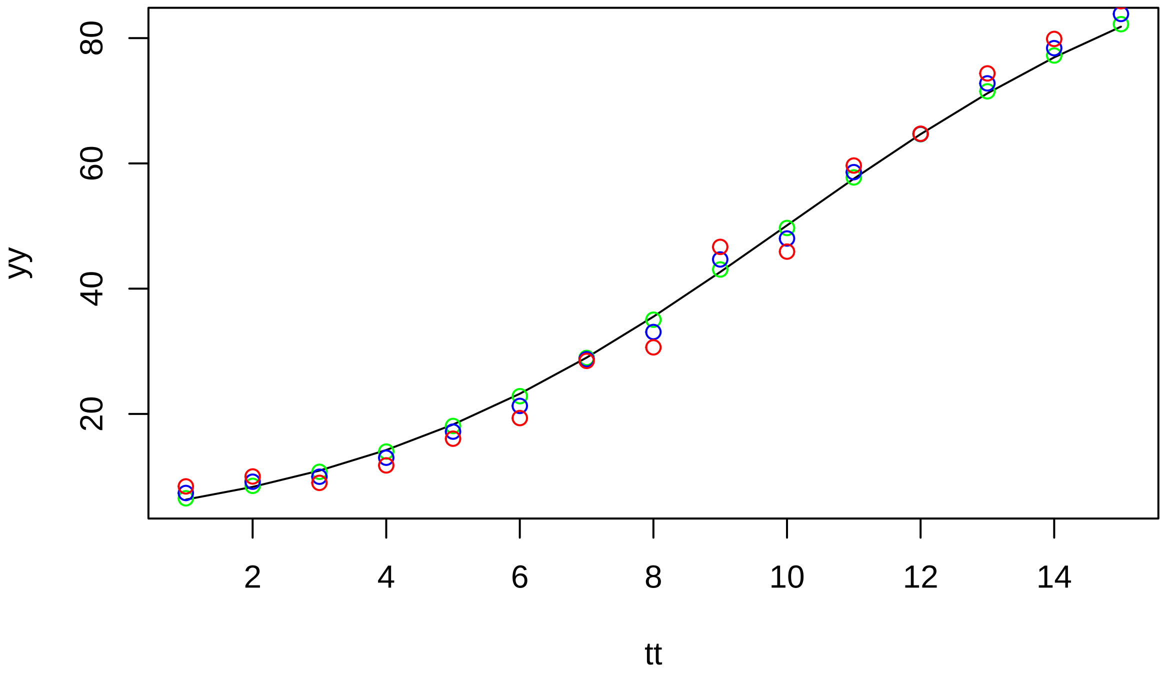
The obvious ways to reduce storage are:

- 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))
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))
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))
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  -4.543e-11      764.2
## b2              44.7099       2.832       15.79  2.158e-09   4.518e-11      0.505
## c2             0.300719      0.0125       24.06  1.595e-11   3.165e-08      0.163
```

```
sol3 <- nlxb(y3 ~ a3/(3+b3*exp(-c3*tt)), data=lg3d15, start=list(a3=1, b3=1, c3=1))
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    2.19e-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. First we will examine the sub-problem of solving the Gauss-Newton equations or their Marquardt variants.

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 $r^t r$. 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.

Example setup and run in lm()

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  1.4576e-16
## [4,]  0.40825 -0.292770   0.66060  1.0074e-15
## [5,]  0.40825 -0.531816   0.38717  3.9931e-01
## [6,]  0.40825 -0.770861  -0.21358  9.0453e-01
##
## $qraux
## [1] 1.4082 1.1853 1.6067 1.1496
##
## $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).

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

I believe that the structure of lm() predates the availability of the family of `qr.xxx()` functions for R. These allow us to access the QR approach directly.

```
x <- qr.solve(A,y)
print(x)
```

```
## one   v  v2
##  -3   1   1
```

```
xe<-qr.solve(A, ye)
print(xe)
```

```
##      one        v       v2
## -2.80191  1.14561  0.95334
```

```
# But then we get an error when we try to solve the singular system
# This was NOT caught above.
xx <- try(qr.solve(Ax,y))
```

```
## Error in qr.solve(Ax, y) : singular matrix 'a' in solve
```

```
print(xx)
```

```
## [1] "Error in qr.solve(Ax, y) : singular matrix 'a' in solve\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in qr.solve(Ax, y): singular matrix 'a' in solve>
```

```
xxe<-try(qr.solve(Ax, ye))
```

```
## Error in qr.solve(Ax, ye) : singular matrix 'a' in solve
```

```
print(xxe)
```

```
## [1] "Error in qr.solve(Ax, ye) : singular matrix 'a' in solve\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in qr.solve(Ax, ye): singular matrix 'a' in solve>
```

Traditional Normal equations approach

The historically traditional method for solving the **linear** least squares problem was to apply calculus to set the partial derivatives of the sum of squares with respect to each parameter to zero. This forms the **normal equations**

$$A^t A x = A^t b$$

This was attractive to early computational workers, since while A is m by n, $A^t A$ 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).

```
AtA <- t(A)%*%A
print(AtA)
```

```
##     one   v   v2
## one   6  21   91
## v    21  91  441
## v2   91 441 2275
```

```
Aty <- t(A)%*%y
print(t(Aty))
```

```
##      one   v   v2
## [1,]  94 469 2443
```

```
x<-solve(AtA,Aty)
print(t(x))
```

```
##      one v v2
## [1,]  -3 1  1
```

Let us try this with the extended matrix `Ax`

```
AxA <- t(Ax)%*%Ax
print(AxA)
```

```
##     one   v  vx   v2
## one   6  21  51   91
## v    21  91 196  441
## vx   51 196 451  896
## v2   91 441 896 2275
```

```
Axy <- t(Ax)%*%y
print(t(Axy))
```

```
##      one   v  vx   v2
## [1,]  94 469 939 2443
```

```
xx<-try(solve(AxA,Axy))
```

```
## Error in solve.default(AxA, Axy) :
##   Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```

```
print(t(xx))
```

```
##      [,1]
## [1,] "Error in solve.default(AxA, Axy) : \n  Lapack routine dgesv: system is exactly singular: U[3,3
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in solve.default(AxA, Axy): Lapack routine dgesv: system is exactly singular: U[3,3] = 
```

Dealing with singularity

The `lm.fit()` function has a parameter `singular.ok` which defaults to TRUE. By setting this to FALSE, we get.

```
sol2b<-try(lm.fit(Ax,y, singular.ok=FALSE))
```

```
## Error in lm.fit(Ax, y, singular.ok = FALSE) : singular fit encountered
```

```
print(sol2b)
```

```
## [1] "Error in lm.fit(Ax, y, singular.ok = FALSE) : singular fit encountered\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in lm.fit(Ax, y, singular.ok = FALSE): singular fit encountered>
```

We've already seen above that the solution `sol2` has a size of

```
eval(as.numeric(crossprod(S2)))
```

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

The solution `sol2` is, it turns out, not unique, since the variables `v` and `vx` and `one` are related, namely

```
print(vx - (v+5*one))
```

```
## [1] 0 0 0 0 0 0
```

Thus we can find a "new" solution as follows and show (essentially) the same residuals

```
res2<-Ax%*%S2-y
print(t(res2))
```

```
##            [,1]       [,2] [,3]        [,4]        [,5]        [,6]
## [1,] 4.4409e-16 4.4409e-16    0 -3.5527e-15 -3.5527e-15 -7.1054e-15
```

```
print(S2)
```

```
## one   v  vx  v2
##  -3   1   0   1
```

```
S2b<-S2+c(10,2,-2,0)
res2b<-Ax%*%S2b-y
print(t(res2b))
```

```
##              [,1]       [,2] [,3] [,4]        [,5]        [,6]
## [1,] 1.3323e-15 1.7764e-15    0    0 -3.5527e-15 -7.1054e-15
```

```
cat("Sum of squares of S2b=", as.numeric(crossprod(S2b)),"\n")
```

```
## Sum of squares of S2b= 63
```

Approximate solution – Ridge regression

An approximate solution via the normal equations can be found by adding a small diagonal matrix to the sum of squares and cross products `AxA`.

```
AxA <- AxA + diag(rep(1e-10,4))
print(AxA)
```

```
##     one   v  vx   v2
## one   6  21  51   91
## v    21  91 196  441
## vx   51 196 451  896
## v2   91 441 896 2275
```

```
xxx<-try(solve(AxA,Axy))
print(t(xxx))
```

```
##            one      v       vx v2
## [1,] -0.40762 1.5185 -0.51848  1
```

```
print(t(Ax %*% xxx - y))
```

```
##              [,1]       [,2]        [,3]        [,4]        [,5]      [,6]
## [1,] 1.6885e-10 8.0442e-12 -8.0778e-11 -9.7611e-11 -4.2455e-11 8.469e-11
```

We note that this solution is rather different from `S2` or `S2b`. There is a large (and often misguided and confusing) literature about this sort of approach under the title **ridge regression**. Note that the Marquardt stabilization of the Gauss-Newton equations uses the same general idea.

Dealing with singularity in QR solutions

We already saw that we got errors when trying to use `qr.solve()`, but there are ways to use the QR decomposition that overcome the singularity. Here are some illustrations. Note that the size of the solution as measured by the sum of squares of the coefficients (ignoring the NA usng `na.rm=TRUE`) are different. Moreover, they are different from the minimum length solution in the next subsection.

```
## This fails
xx <- try(qr.solve(Ax,y))
```

```
## Error in qr.solve(Ax, y) : singular matrix 'a' in solve
```

```
print(xx)
```

```
## [1] "Error in qr.solve(Ax, y) : singular matrix 'a' in solve\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in qr.solve(Ax, y): singular matrix 'a' in solve>
## So does this
xxe<-try(qr.solve(Ax, ye))
```

```
## Error in qr.solve(Ax, ye) : singular matrix 'a' in solve
```

```
print(xxe)
```

```
## [1] "Error in qr.solve(Ax, ye) : singular matrix 'a' in solve\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in qr.solve(Ax, ye): singular matrix 'a' in solve>
```

```
## Let us compute a QR decomposition, using LINPACK then LAPACK
qrd1<-qr(Ax, LAPACK=FALSE)
qrd2<-qr(Ax, LAPACK=TRUE)
# and get the solutions
xx1 <- try(qr.coef(qrd1,y))
print(xx1)
```

```
## one   v  vx  v2
##  -3   1  NA   1
```

```
xx2 <- try(qr.coef(qrd2,y))
print(xx2)
```

```
##      one        v       vx       v2
## -1.83224  1.23355 -0.23355  1.00000
```

```
# and computer the sum of squares of the coefficients (size of solution)
try(sum(xx1^2))
```

```
## [1] NA
```

```
try(sum(xx1^2, na.rm=TRUE))
```

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

```
try(sum(xx2^2))
```

```
## [1] 5.9333
```

Minimum length least squares solution

The minimum length least squares solution, which is unique, is found using the Moore-Penrose inverse of `Ax` (the article https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse gives a quite good overview) which can be computed from the Singular Value Decomposition. R has a function `svd()` which is adequate to our needs here.

```
Z<-svd(Ax) # get the svd
D1 <- 1/Z$d # invert S, the singular values diagonal matrix
print(D1)
```

```
## [1] 1.9126e-02 1.0613e-01 1.5144e+00 1.5931e+15
```

```
D1[4]<-0 # to remove linear dependency (small singvals are set to zero)
# D1 is diagonal of S+ now
# minimum length LS solution A+ = V S+ t(U)
minsol2 <- Z$v %*% (diag(D1) %*% (t(Z$u) %*% y))
print(minsol2)
```

```
##           [,1]
## [1,] -0.40741
## [2,]  1.51852
## [3,] -0.51852
## [4,]  1.00000
```

```
cat("SS of minsol2=",as.numeric(crossprod(minsol2)),"\n")
```

```
## SS of minsol2= 3.7407
```

```
resminsol<-Ax%*%minsol2-y
print(t(resminsol))
```

```
##            [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
## [1,] 4.3299e-15 7.1054e-15 1.0658e-14 1.4211e-14 1.4211e-14 1.4211e-14
```

QR decomposition variants

There are a number of ways to compute a QR decomposition.

- Householder transformations have good vector-computation performance and properties. They use matrix transformations that can be called **reflections** to be used to build the Q matrix as a product of these transformation matrices as the initial rectangular matrix A is rendered upper triangular, or R
- Givens' transformations are **plane rotations** that zero one element at a time of the current working matrix that starts out as A. Once again, Q is a product of the transformations and R the result of zeroing out sub-diagonal elements.
- The Gram-Schmidt method, which has several forms, builds the Q matrix by subtracting a multiple of the first column of A from the second and then normalizing those columns so their sum of squares is 1. The process is then applied to the third column by subtracting multiples of the first two. The R matrix is formed from the norms and subtraction multipliers in the process. There are row- and column-wise approaches. See https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process

Using a QR approach

Let us try 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

$$Q^t A = Q^t QR = R \tilde{=} Q^t b$$

R is an upper triangular matrix R_n stacked on an $m - n$ by n matrix of zeros. But $z = Q^t b$ 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 $z_2^t z_2$. 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 of John C. Nash (1979), 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).

QR approach to the Marquardt stabilization

We can arrange to solve the normal equations approximately as under Ridge Regression above by adding a unit matrix scaled by e.g., 1e-5, and padding the RHS with zeros. The solution is "almost but not quite" the same because the formation of the sum of squares and cross products matrix explicitly in the normal equations can introduce errors that are partially avoidable in the QR approach below.

```
Dx <- diag(rep(1e-5,4))
print(Dx)
```

```
##       [,1]  [,2]  [,3]  [,4]
## [1,] 1e-05 0e+00 0e+00 0e+00
## [2,] 0e+00 1e-05 0e+00 0e+00
## [3,] 0e+00 0e+00 1e-05 0e+00
## [4,] 0e+00 0e+00 0e+00 1e-05
```

```
Ap<-rbind(Ax, Dx)
print(Ap)
```

```
##         one     v      vx      v2
##  [1,] 1e+00 1e+00 6.0e+00 1.0e+00
##  [2,] 1e+00 2e+00 7.0e+00 4.0e+00
##  [3,] 1e+00 3e+00 8.0e+00 9.0e+00
##  [4,] 1e+00 4e+00 9.0e+00 1.6e+01
##  [5,] 1e+00 5e+00 1.0e+01 2.5e+01
##  [6,] 1e+00 6e+00 1.1e+01 3.6e+01
##  [7,] 1e-05 0e+00 0.0e+00 0.0e+00
##  [8,] 0e+00 1e-05 0.0e+00 0.0e+00
##  [9,] 0e+00 0e+00 1.0e-05 0.0e+00
## [10,] 0e+00 0e+00 0.0e+00 1.0e-05
```

```
## we need to pad y with zeros
yp<-c(y, rep(0,4))
xxp<-qr.solve(Ap,yp)
print(xxp)
```

```
##      one        v       vx       v2
## -0.40741  1.51852 -0.51852  1.00000
## Previous solution via normal equations
print(as.vector(xxx))
```

```
## [1] -0.40762  1.51848 -0.51848  1.00000
```

Choices when solving the normal equations

Whether the normal equations arise from the original Gauss-Newton or Marquardt approaches to nonlinear least squares, we have several choices of how to solve them.

Traditional solutions to the solution of linear equations used elimination techniques under names such as **Gaussian Elimination**, **Gauss-Doolittle**, **Crout's Method**, **Gauss-Jordan** and many others. All these can be considered as LU decompositions. That is, they can be expressed as a transformation of the matrix $A = J^t J$ or its Marquardt stabilized variant into a product of a lower triangular and an upper triangular matrix, possibly with permutations of rows and/or columns. Row permutations of A give

$$PA = LU$$

The origin of our matrix A means that it is at least non-negative definite. This means that we can form a special LU decomposition that does NOT need permutations and has

$$A = LL^t$$

which is called the Choleski decomposition. In practice, we need A computationally non-singular (not just theoretically so), but the Marquardt stabilization can be used to ensure this.

If we are very worried about storage space, we can note that A is symmetric, so that we only need to store the lower (or upper) triangle, including diagonals. And some bright programmers realized we could do our storage in a single array, so that a matrix of order n could be stored in a vector of length $n(n+1)/2$. John C. Nash and Walker-Smith (1987b) uses this Cholesky decomposition structure in BASIC code.

Similarly, a Gauss Jordan variance could be programmed using such a format for A and with only a temporary storage of size n, matrix A could be overwritten with its inverse. This is the code of Bauer and Reinsch (1971) which is Algorithm 9 of John C. Nash (1979).

Both these "single vector" storage programs are challenging to check and maintain. While I embraced them in the age when 16K of storage was the norm for personal computers, they are simply too finicky to follow easily, and highly prone to errors.

References

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, et al. 1999. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial; Applied Mathematics.

Bauer, F. L., and C. Reinsch. 1971. "Inversion of Positive Definite Matrices by the Gauss-Jordan Method." in Wilkinson et al. 1971, pages 45–49.

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. 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.

———. 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. 1987b. *Nonlinear Parameter Estimation: An Integrated System in BASIC.* Book. Marcel Dekker Inc.: New York.

———. 1987a. *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.*