

Variety in the Implementation of Nonlinear Least Squares Program Codes

John C Nash, retired professor, University of Ottawa

16/02/2021 revised 20/01/2022

Contents

Abstract	1
The Problem to be solved	2
Underlying algorithms	2
Gauss Newton method	2
Hartley's method	4
Marquardt stabilized Gauss-Newton	4
Other approaches	6
Sources of implementation variety	7
Algorithm structure	7
Programming language	8
Data linkage to software	8
Operating environment	8
Solver for the least squares or linear equations sub-problems	9
Storage structure	10
Sequential or full Jacobian computation	17
Analytic or approximate Jacobian	17
Interfacing to problems	18
Saving storage	18
Measuring performance	18
Test problems	18
Implementation comparisons	20
Linear least squares and storage considerations	20
Traditional Normal equations approach	25
Implementation choices for an R package	32
References	32

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 (1979a),

the author realized that this variety could be an instructive subject for numerical software designers.

The Problem to be solved

Given a function that can be written as a sum of squared residuals

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

where x is a set of real valued parameters, we wish to find the minimum of $f(x)$ with respect to x . There may be additional conditions that add to the complexity of the problem such as

- bounds (or box) constraints of the form

$$l_i \leq x_i \leq u_i$$

alternatively written in vector form

$$l \leq x \leq u$$

- masks (or fixed parameters), for example,

$$x_i = \text{value}_i$$

. Such constraints reduce the dimensionality of the nonlinear least squares problem, and we could specify the problem without the masked parameters. However, it is sometimes very convenient to be able to write our problem with such parameters that may later be altered in value or allowed to vary in an optimization of the sum of squares.

- inequality constraints, which in general may be written as

$$C_k(x) \geq 0$$

which can be written as a vector form

$$C(x) \geq 0$$

A special and important case is that of **linear inequalities**

$$Ax \geq b$$

that is,

$$C(x) = Ax - b$$

Underlying algorithms

Gauss Newton method

An overview of this method is provided by Wikipedia: https://en.wikipedia.org/wiki/Gauss%E2%80%93Newton_algorithm The method was described by C.F.Gauss in 1809 in *Theoria motus corporum coelestium in sectionibus conicis solem ambientum*.

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 , $i = 1, \dots, n$, 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 , assuming that the method does indeed converge for the problem at hand. Newton (and his sometimes associated worker Raphson) used the algorithm we attribute to them in a very special case and in a manner unrecognizable as that described above. (See https://en.wikipedia.org/wiki/Newton%27s_method).

For our present situation, the $f(x)$ can be written as a sum of squares as above. It is worth repeating the vector form of this

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

because it is less fussy to write and read. This is clear when we expand the evaluation of the Hessian of this sum of squares function.

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

If we define the **Jacobian** matrix

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

then the gradient, in vector form, 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 (which have elements of the residual vector times second derivatives of the residual). This gives the **Gauss-Newton** equations (cancelling a factor 2)

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

We can use this in an iteration similar to the Newton iteration. Note that these equations are equivalent to the linear least squares problem that can be written

$$J\delta \approx -r$$

Like most other treatments, we will skate over the thin ice that $J^t J$ is a good approximation of the Hessian and that the resulting method converges.

Hartley's method

Conditions for convergence of the Newton iteration and the Gauss-Newton iteration are rather a nuisance to verify. In any case we want a solution to the original nonlinear least squares problem. 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 α . This one-dimensional minimization can take quite a lot of computational effort, however, and typically a value of α that permits a suitable reduction in the sum of squares $f(x)$ is accepted, and the iteration repeated from

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

Hartley (1961) recommends a quadratic (parabolic) line search where two “new” points are evaluated, followed by a third try at the suggested minimum of the parabola fitted to the existing and two new sum of squares function values. Tactics are needed to deal with the case that the proposed point is not an improvement in the sum of squares.

A different approach is the R function `nls()`, which calls the C routine `nls.c` to compute the Gauss Newton direction after which a step-halving “line-search” is used.

In `nls.c` we also see that once a satisfactory “new” set of parameters has been obtained, we reset the α value to the minimum of $2 * \alpha$ and 1.0. Oddly, there is no provision for the user to set the tactical choices of the maximum value, the reduction or the increase in α , despite our experience that other choices can be advantageous for some problems. Moreover, x_{new} is deemed satisfactory if its **deviance** (essentially the weighted sum of squares of the residuals) is **no greater than** the current best value. Our experience is that it may be better to insist on a decrease in the value, or even to use an “acceptable point” approach that enforces a sufficient decrease using a measure based on the gradient of the deviance projected on δ .

Clearly there are many tactical choices that give rise to a variety of particular algorithms. One concern is that the direction δ may give no lower value of the sum of squares, since there is an approximation involved in using $J^t J$ rather than H . For example, δ could be perpendicular to any downhill direction.

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

$$(J^t J + \lambda \mathbf{1}_n) \delta = A \delta = -J^t r$$

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 Levenberg likely never exercised them in practice, while Marquardt actually had a well-used program. Before computers were commonly available, many papers were written with computational ideas that were given at most a cursory trial. (In 1984, I had the pleasure of showing Marquardt my own BASIC program running on a Radio Shack TRS80 Model 100 and he commented on how the computations were converging to a solution as we watched the iterations proceed.)

Note that we can define a matrix K which is J augmented below by a suitable diagonal matrix that is the square root of λ times the unit matrix $\mathbf{1}_n$. Then the Levenberg-Marquardt equations are equivalent to

$$K \delta \approx -r_{augmented}$$

where we have added n zeros to r to make $r_{augmented}$. Similar adjustments are used for scaled Marquardt methods mentioned below.

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 we wish to minimize to “fit” our nonlinear model, one set of choices is:

- set an initial λ of 0.0001
- **A** Compute J and g ,
- **B** set up Marquardt equations for current λ and find δ , compute new set of parameters

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

- if $x_{new} \neq x$, 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, apart from the many possibilities that “find δ ” allows, an algorithm is defined by the initial value, decrease factor (e.g., 0.4) and increase factor (e.g., 10) for λ . However, there are a number of further 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 x , 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 number $(x_{new} + \text{OFFSET})$ matches bit for bit $(x + \text{OFFSET})$, then we take the unmodified numbers as equal.

Many practitioners dislike using IF statements (comparisons) of floating point numbers. Tolerances are often suggested to define “small,” but this invites users to make decisions that lead to poor outcomes and which ignore the precision and other characteristics of the arithmetic at hand. There are, of course, well-written

codes that address the issues of what “small” means, but such code is generally quite involved. The R function `all.equal()` presents a well-considered approach. However, I have never seen a case where the *OFFSET* approach caused difficulties, and it is trivial to code. 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.

Automatic scaling

In Marquardt (1963), it is shown that the use of the diagonal elements of $J^t J$ instead of $\mathbf{1}_n$ 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 (Nash (1977)) is to slightly modify the equations to

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

where ϕ is a modest number such as 1.

Note that we form $J^t J$ in `nlsr::nlfb()` to provide the D matrix for the scaled stabilization, which is quite a lot of computational work. The same program, however, uses a QR approach to solving the linearized least squares sub-problem. To avoid the formation of D , an experimental package `nlsralt` was built to add functions `nlxbx()` and `nlfbx()` that use ONLY the fixed element (i.e., unit matrix) stabilization. A preliminary timing (using Mersmann (2013)) with the Hobbs unscaled problem (Nash (1979b), Example 12.2) shows a slight advantage for `nlsralt` in execution timing with an equivalent solution.

Other approaches

While the above ideas are the dominant theme in nonlinear least squares, there have been many other proposed approaches. The following is a non-exhaustive sample.

Spiral

Jones (1970) seems only to be mentioned in a single paper. There is a flowchart given but I have never been able to find a code (supposedly Fortran V for the Univac 1108.) The motivations are to avoid both matrix equation solutions and one-dimensional line-search minimization. Jones mentions matrix inversion, but it is not clear whether he truly forms the inverse. The main idea in the paper (and presumably the code) is to use a geometric spiral that mimics the Marquadt search trajectory as λ is increased. It would be interesting to actually try some of the ideas, which do, as far as I can determine, have a reasonable basis and could possibly offer some efficiencies.

Approaches to Newton’s method for special models

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 suggested Newton-like approaches, possibly for special types of models where some short cut to the Hessian is available. I regard these approaches as part of the computational infrastructure of the special cases, as it is usually difficult to apply them outside of their context.

Two such examples are the R packages `maxLik` (Henningsen and Toomet (2011)) and `trust` (Geyer (2013)).

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 a 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. Unfortunately, the effort to benchmark the possibilities seems higher than any benefit in replacing quite good existing methods.

Sources of implementation variety

The sources of variety in implementation of nonlinear least-squares methods include:

- structuring of the algorithm, that is, how we set up and sequence the parts of the overall computation
- Linking of data to the problem. In particular, R allows **subsetting** of data at quite a high level, but this capability is almost totally hidden from the user or programmer, so that the feature could be dangerous to use. There appears, moreover, to be no example or test with `nls()` to show how to use this facility, even though `subset` is an argument to the `nls()` function. This is discussed further below.
- programming language
- possible operating environment features
- solver for the least squares or linear equations sub-problems
- structure of storage for the solver, that is, compact or full
- sequential or full creation of the Jacobian and residual, since this 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` versus packages `minpack.lm` and `nlsr`, or as a general optimization problem
- incorporation of constraints on the parameters, namely bounds, masks or inequalities.

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 `nlsr` (John C Nash and Duncan 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 the `nls()` structure

A major criticism of the `nls()` 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 sufficiently 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 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.

Data linkage to software

Quite reasonably, linking particular data to our problem is mediated by software, and hence is influenced by the programming language. This is particularly true for R, since R allows us to **subset** data. Essentially this provides an index to the rows of data, most easily thought of in terms of a data frame structure.

Data subsets

The `nls()` function has an argument **subset** and a test included in the `Croucher-example1.R` file of this project showed this works as expected. However, nowhere have we found a published example of the use of the **subset** argument with `nls()`. Worse, it clearly must be applied to many of the functions used by `nls()` but makes no explicit appearance. Thus it is VERY difficult to ensure subsetting is correctly applied when we wish to modify the code, for instance, to provide for newer features especially more reliable solvers.

Note that the **subset** ARGUMENT to `nls()` is not the same as the `subset()` function that is usable to restrict the scope of an R object.

OUR HYPOTHESIS: `nls()` uses **subset** to automatically adjust the data to which all the relevant functions are applied. We note that `minpack.lm::nlsLM` also has a **subset** argument, but package `nlsr` does not at time of writing (2021-7-12), but will use the **weights** mechanism of the next subsection of this document at some future release.

Other approaches to subsetting

In order to provide a sane and well-understood mechanism for subsetting, there are alternatives for implementation.

- **via weights:** We can weight each “row” that we do NOT wish included as 0. This does require us to compute the number of observations as the count of non-zero weights e.g, ‘`sum(weights > 0)`’. Incidentally, this underlines the need to ensure weights are non-negative. ??now in `nlsj`, need to add to `nlsr` code.??
- **via preprocessing:** We could prepare a function to pre-process the problem and create a data collection that is appropriately subset. This may offend against the spirit of R/S, but it has the merit that we can easily reassure ourselves that we have worked with the correct information, correctly used in our methods. ?? Should add this as an option.

PROPOSAL: use the “weights” approach if `nlsj` is called with **subset** active, else recommend the preprocessing approach. The preprocessor could also do a lot of checking, which we might leave out of `nlsj()` in some cases. As of 2021-7-12, the weights mechanism is working in a trial `nlsj()` and should be transferrable to package `nlsr` fairly easily.

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, though the actual performance of the computations may well be influenced by such factors. 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 than in the past.

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 or 7 (decimal) digits of precision. In such environments, where, moreover, guard digits and underflow and overflow behaviour were highly variable

features, it was quite common that arguments to functions like $\exp()$ and $\log()$ were inadmissible for reasons of computability in the local system rather than underlying mathematics. In such environments, where there were also storage limitations, it was important to use very robust algorithms implemented in ways that were simple but “safe.” This led to a number of choices in the development of the codes in Nash (1979a). However, it has turned out that these choices fare rather well even in more capable computational systems.

Solver for the least squares or linear equations sub-problems

There are many choices for solver of the Gauss-Newton or related equations in a nonlinear least squares program. We will consider some of the options. Note that the choice to set up the normal equations will direct part of the algorithm selection.

Solution of the linear normal equations

The Gauss-Newton or Marquardt equations 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 a coefficient matrix A by using several matrix decomposition methods. Thus there are many possible procedures.

- Gauss elimination with partial or complete pivoting: The choice of partial or complete pivoting changes greatly the program code. However, neither of these choices uses the underlying structure and non-negative definite nature of the matrix A .
- Variants of Gauss elimination that build matrix decompositions: We can apply elimination to produce an LU decomposition of A , and this may offer some computational advantages, though in the present case such approaches do not seem to be ideal.
- Gauss-Jordan inversion: Here we compute the inverse of A and then work with this inverse. That is, we use

$$\delta = -A^{-1}g$$

This uses much more computational effort than simply solving for δ , though there is an interesting, and very tiny, method by Bauer and Reinsch (1971). The sheer ingenuity of this code was magnetic, but it has proved to be of limited value.

- Choleski Decomposition and back substitution: There are a number of varieties of Choleski decomposition, but they all take advantage of the symmetric and non-negative definite structure of A . A particularly compact form was used in Nash and Walker-Smith (1987a), though it is debatable whether the saving in storage was not heavily outweighed by the complexity of the code and its documentation.
- Eigendecompositions of the sum of squares and cross-products (SSCP) matrix: the eigenvalues of real, symmetric matrices such as A can be computed in many ways and the eigendecomposition used to solve the Gauss-Newton equations. The computations are much more costly than simple solution of the equations, but the knowledge of the eigenvalues allows us to avoid, or at least be aware of, conditioning issues.

Solution of the least squares sub-problem by matrix decomposition

Here we proposed to solve using the matrix J or an augmented form K directly. There are a number of approaches that can be used, which we will divide into two classes, QR and SVD methods.

The QR approach aims to decompose the working matrix (call it K , with m rows and n columns)

$$K = QR$$

where Q is m by n orthogonal so

$$Q^t Q = I_n$$

and R is upper triangular and n by n . Some variants have Q an m by m orthogonal matrix and R upper triangular with $m - n$ rows of zeros appended below.

There are many ways of computing such decompositions:

- Householder reflections with or without pivoting options
- Givens plane rotations, similarly
- Gram-Schmidt orthogonalization of the columns of K , either directly (one column at a time orthogonalized to all previous columns) or in a modified fashion that proceeds row-wise. There are a number of detailed choices.

SVD approaches compute a decomposition

$$K = U S V^t$$

where U and V have various orthogonality properties and S is a diagonal matrix with elements called the singular values (the square roots of the eigenvalues of $K^t K$ actually). There are many variations depending on whether U is a full m by m matrix, limited to m by n or even truncated to m by r where r is the rank of K .

A detail in choosing our approach is the mechanism of inclusion of the Marquardt (or Marquardt Nash) stabilization. It turns out to be remarkably easy, if a little tedious. We simply augment J by a diagonal matrix that has elements $\sqrt{(\lambda)}$ times the diagonal stabilization elements we wish to use. Then $K^t K$ is the appropriate A for the Marquardt version of the Gauss-Newton.

As we explain below, we do, unfortunately, realize that Marquardt suggested a scaled stabilization where we want to add the diagonals of $J^t J$ normal equation approach. This requires that we perform at least a partial computation of the SSCP matrix, which is relatively costly. For this reason we plan ?? to explore some proxies for these elements.

Avoiding re-computation of the Jacobian

The “original” Gauss-Newton iteration offers no guidance in how to deal with a situation where x_{new} has a poorer fit than x . Hartley (1961) and Marquardt (1963), along with most other practical methods, need to recompute the (weighted) residuals and consequent loss function (sum of squares or deviance) for each value of a step-length or stabilization parameter (λ) until either a “better” point x_{new} is found or we terminate the search. However, as we have seen, Hartley does not have to adjust the search direction.

Thus it could be helpful to separate the computation of residuals from that of the Jacobian. However, the structure of R is such that it is much easier to have the Jacobian as an **attribute** of the residuals object. Indeed, it is the “gradient” attribute for historical reasons. This is the setup for both the finite difference approximations of `nlspkg::numericDeriv()` (the Jacobian routine in base R for `nls()` in file `./src/library/stats/R/nls.R`) as well as the `rjfun()` generated by `nlshr::model2rjfun()`. On the other hand, the computation of Jacobian elements can share many calculations with the computation of the residuals. Evidence is needed as to which de-duplication efforts give worthwhile savings, and in what situations, but is likely of importance only to users with extremely time consuming problems.

Storage structure

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 construction that matrix is symmetric 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 (Nash (1979a)). Algorithms 7 and 8 are the Cholesky decomposition and back-solution using a vector of length $\mathbf{n}*(\mathbf{n}+1)/2$ to store just the lower triangle of the SSCP matrix. Algorithm 9 inverts this matrix *in situ*.

The original Nash (1979a) 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 Nash (1990) and in Nash and Walker-Smith (1987b). However, in the now defunct Nash (2016) and successor John C Nash and Duncan 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 Nash and Walker-Smith (1987a), 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. The diagonal elements are then multiplied by $(1 + \lambda)$ and $\lambda * \phi$ is added to each, then saved in matrix C . λ is our Marquardt parameter and ϕ is the adjustment suggested in Nash (1977) to overcome potential underflow to zero of the accumulation of the diagonal elements in low-precision arithmetic.

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. We do not, however, need to build a new SSCP matrix or gradient g , just make the diagonal adjustment to A to get a new C .

The code in Nash and Walker-Smith (1987a) 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. (Readers may note that the codes were written before “IEEE Standard for Binary Floating-Point Arithmetic” (1985) was applied to most BASICs.)
- 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 Duncan Murdoch (2019), the approach to the augmented Gauss-Newton equations is nominally equivalent to that in Nash and Walker-Smith (1987a), 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 Nash (1979a), 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 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 diagonal matrix $Z1$ 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 in 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.

??FIXME??

```
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, modelepr,
##   newDeriv, newSimplification, nlfb, nlsDeriv, nlsSimplify, nlxb,
##   predict.nlsr, print.nlsr, res, resgr, resss, rjfundoc,
```

[illegible]

[illegible]

[illegible]

[illegible]


```
## 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
## tnlsrh1 <- nlxb(eunsc, data = weeddata1, start = start1) 5.0612 5.1457 5.4073
## median      uq      max neval
## 5.1783 5.2955 7.738 100
##
## > tnlsrxh1
## Unit: milliseconds
##
##                               expr      min      lq      mean
## tnlsrxh1 <- nlxb(eunsc, data = weeddata1, start = start1) 4.3747 4.4895 4.6982
## median      uq      max neval
## 4.5368 4.6115 7.0576 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 each part is well-documented but the general flow and linkage of the parts is 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 decomposition), 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 the speed in getting there.

Interfacing to problems

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, however, 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

Do we plan to use only timing? Or size of code and memory used? ??

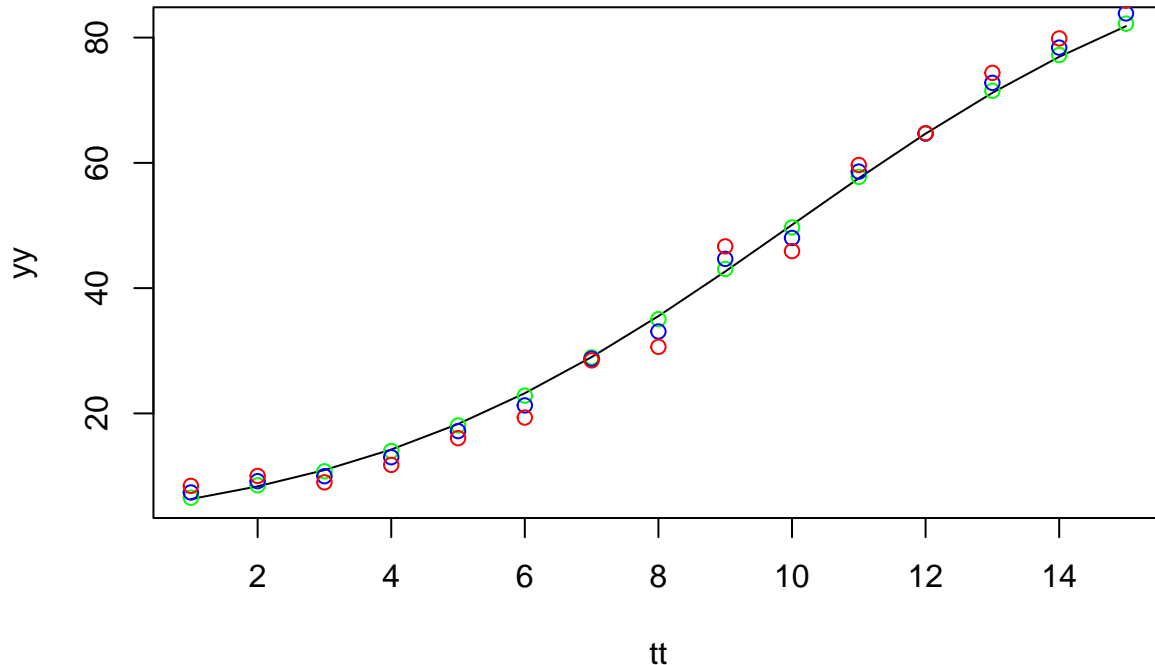
Test problems

```
# 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)
```

```
## nlslr 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 we 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

$$Ax \approx b$$

In this case A is an m by n matrix with $m \geq n$ and b a vector of length 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 Nash (1979a) and 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] = 0"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in solve.default(AxA, Axy): Lapack routine dgesv: system is exactly singular: U[3,3] = 0
```

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 using `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 singuals 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,] [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 (Bindel et al. (2002)) 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

$$Q^t A = Q^t Q R = R \hat{=} Q^t b$$

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 we 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 Nash (1979a), 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))
xqp<-qr.solve(Ap,yp)
print(xqp)

##      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 $n(n+1)/2$. Nash and Walker-Smith (1987a) uses this Cholesky decomposition structure in BASIC code.

Similarly, a Gauss Jordan variant 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 Nash (1979a).

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 coding and maintenance errors.

Implementation choices for an R package

??2021 update to nlsr and ideas for the “Improvements to nls()” project.

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.
- Bindel, D., J. Demmel, W. Kahan, and O. Marques. 2002. “On Computing Givens Rotations Reliably and Efficiently.” *ACM Transactions on Mathematical Software* 28 (2): 206–38.
- Geyer, Charles J. 2013. *Trust: Trust Region Optimization*. <http://CRAN.R-project.org/package=trust>.
- Hartley, H. O. 1961. “The Modified Gauss-Newton Method for Fitting of Nonlinear Regression Functions by Least Squares.” *Technometrics* 3: 269–80.
- Henningsen, Arne, and Ott Toomet. 2011. “maxLik: A Package for Maximum Likelihood Estimation in R.” *Computational Statistics* 26 (3): 443–58. <https://doi.org/10.1007/s00180-010-0217-1>.
- “IEEE Standard for Binary Floating-Point Arithmetic.” 1985. *ANSI/IEEE Std 754-1985*. <https://doi.org/10.1109/IEEESTD.1985.82928>.
- John C Nash, and Duncan Murdoch. 2019. *nlsr: Functions for Nonlinear Least Squares Solutions*.
- Jones, A. 1970. “Spiral—A new algorithm for non-linear parameter estimation using least squares.” *The Computer Journal* 13 (3): 301–8.
- Lefkovich, 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.
- Mersmann, Olaf. 2013. *Microbenchmark: Sub Microsecond Accurate Timing Functions*. <http://CRAN.R-project.org/package=microbenchmark>.
- 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.
- . 1979b. *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Bristol: Adam Hilger.
- . 1979a. *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. 1987a. *Nonlinear Parameter Estimation: An Integrated System in BASIC*. Book. Marcel Dekker Inc.: New York.
- . 1987b. *Nonlinear Parameter Estimation: An Integrated System in BASIC*. New York: Marcel Dekker.