Fwd: NL2SOL cursor response

Date: 17/03/2026, 12:30 [cite: 2]
To: Peter Jaeckel <*****@*****> [cite: 3]

From: Jesper Andreasen <*****@*****> [cite: 7]
Sent: 16 March 2026 08:06 [cite: 10]
Subject: NL2SOL cursor response [cite: 12]

Here's a detailed mathematical walkthrough of the whole algorithm, the loop, and the bug. [cite: 13]

The Big Picture: NL2SOL

NL2SOL solves nonlinear least-squares problems: given a vector-valued residual function $r(x)$, find x that minimises: [cite: 14, 15]

$$f(x) = \frac{1}{2}|r(x)|^2 = \frac{1}{2}\sum_i r_i(x)^2$$

[cite: 16]
It's a Gauss-Newton / Levenberg-Marquardt-style solver. [cite: 17] At each outer iteration (`nl2itr`), it builds a local quadratic model of f and needs to compute a trust-region step s that approximately minimises the model subject to a constraint $|Ds| \leq \Delta$, where D is a diagonal scaling matrix and Δ is the trust-region radius. [cite: 17]

What `gqtstp` Does: The Goldfeld-Quandt-Trotter Step

`gqtstp` computes this trust-region step. [cite: 18, 19] The quadratic model to minimise is: [cite: 19]

$$\psi(s) = g^\top s + \frac{1}{2}s^\top H s$$

[cite: 19]
subject to $|Ds| \leq \Delta$ (the trust region), where: [cite: 20]

- g is the gradient of f [cite: 21]
- H is the (approximate) Hessian [cite: 22]
- D is a diagonal scaling matrix [cite: 23]
- $\Delta = rad =$ the trust-region radius [cite: 24]

Change of Variables

The code works in scaled coordinates. [cite: 25] Define $\tilde{s} = Ds$, so the problem becomes: minimise [cite: 26]

$$\psi(\tilde{s}) = (D^{-1}g)^\top \tilde{s} + \frac{1}{2}\tilde{s}^\top (D^{-1}HD^{-1})\tilde{s}$$

[cite: 27]
subject to $|\tilde{s}| \leq \Delta$. [cite: 28] In the code:

- $dig = (D^{-1}g)$ (the scaled gradient) [cite: 29]
- $dihdi = (D^{-1}HD^{-1})$ (the scaled Hessian, stored as lower-triangular packed format) [cite: 30]
- $rad = \Delta$ [cite: 30]

Let $\tilde{H} = D^{-1}HD^{-1}$ and $\tilde{g} = D^{-1}g$ for brevity. [cite: 31]

The Optimality Condition

By the KKT conditions, the solution satisfies: [cite: 32, 33]

$$(\tilde{H} + \alpha I)\tilde{s} = -\tilde{g}$$

[cite: 34]
for some $\alpha \geq 0$ (the Lagrange multiplier), where: [cite: 35]

- If the unconstrained minimiser $\tilde{s} = -\tilde{H}^{-1}\tilde{g}$ lies inside the trust region ($|\tilde{s}| \leq \Delta$), then $\alpha = 0$. [cite: 36, 38]
- Otherwise, $\alpha > 0$ is chosen so that $|\tilde{s}(\alpha)| = \Delta$ exactly. [cite: 39] This is the "trust-region boundary" case. [cite: 39]
- The matrix $\tilde{H} + \alpha I$ must be positive semidefinite for the step to be a minimiser (not a maximiser). [cite: 40]

The Secular Equation

Define [cite: 41, 42]

$$\phi(\alpha) = |\tilde{s}(\alpha)| - \Delta = |(\tilde{H} + \alpha I)^{-1}\tilde{g}| - \Delta$$

[cite: 43]
The algorithm searches for the root $\phi(\alpha) = 0$. [cite: 43] This is essentially a 1D root-finding problem in α. [cite: 44]

The Algorithm's Inner Loop (L210)

The loop at `L210` is the core iteration that searches for the right α. [cite: 45, 46] Here's what each pass does:

- **Step 1 - Increment iteration counter (line 4724):** `++(*ka);` `ka` counts how many times we've been through this loop. [cite: 47, 48, 49]
- **Step 2 - Safeguard α (lines 4725-4742):** The algorithm maintains bounds $[l_k, u_k]$ on the optimal α: [cite: 50, 51]
  - l_k (lower bound): we know $\alpha^* \geq l_k$ [cite: 52]
  - u_k (upper bound): we know $\alpha^* \leq u_k$ [cite: 53]

  If the current α_k is outside these bounds, it's reset to: [cite: 54]

  $$\alpha_k = u_k \cdot \max\left(0.001, \sqrt{\frac{l_k}{u_k}}\right)$$

  [cite: 55] This is a geometric mean-style interpolation between the bounds (biased towards u_k). [cite: 56] There's also special handling (lines 4736-4741) for when l_k and u_k have converged to nearly the same value—the code gently widens u_k to avoid getting stuck. [cite: 57]
- **Step 3 - Form the shifted matrix (lines 4743-4749):** Add $\alpha_k I$ to the diagonal of $\tilde{H}$: [cite: 58]

  $$\tilde{H} + \alpha_k I$$

  [cite: 59] The for loop at line 4745 does this element-by-element on the packed lower-triangular storage: `dihdi[k] = w[j] + alphak` adds `alphak` to each diagonal entry. [cite: 60]
- **Step 4 - Cholesky factorisation (line 4753):** Try to compute the Cholesky factor L such that: [cite: 61, 62]

  $$\tilde{H} + \alpha_k I = LL^\top$$

  [cite: 63] The Cholesky factorisation succeeds if and only if the matrix is positive definite. [cite: 64]

Path A - Cholesky Fails (lines 4761-4775): Matrix Still Indefinite [cite: 65]
If $\tilde{H} + \alpha_k I$ is not positive definite, then α_k is too small. [cite: 66] The algorithm estimates the most negative eigenvalue using the partial Cholesky factor: [cite: 67]

1. The factorisation failed at row `irc`, producing a negative diagonal element $t = L[j]$ at that row. [cite: 68]
2. It solves a small triangular system to estimate the eigenvector direction. [cite: 69]
3. It computes a better lower bound: [cite: 70]

$$l_k \leftarrow \alpha_k - \frac{t}{|w|^2}$$

   [cite: 71] where t is the negative value encountered and w is the approximate eigenvector. [cite: 72] This uses the fact that if Cholesky fails, the smallest eigenvalue of $\tilde{H} + \alpha_k I$ is approximately $t/|w|^2$, so we need to add at least that much more to α. [cite: 73]

4. `v[dst0] = -lk` is set (a signal that H is not positive definite). [cite: 74]
5. `goto L210` - loop back and try again with a bigger α_k. [cite: 75]

This is where the bug was. [cite: 76] There was no check against the iteration limit `kalim` before looping back. [cite: 76] If the Cholesky kept failing (perhaps due to numerical issues where the eigenvalue estimate wasn't accurate enough to push α high enough), this loop ran forever. [cite: 77]

Path B - Cholesky Succeeds (L250, lines 4779-4821): Check Step Length [cite: 78]
If the Cholesky succeeds, $\tilde{H} + \alpha_k I = LL^\top$ is positive definite. [cite: 78] Now compute the step: [cite: 79]

$$\tilde{s} = -(LL^\top)^{-1}\tilde{g}$$

[cite: 80] Done in two triangular solves (`livmul` and `litvmu`). [cite: 81] Then compute:

$$\phi = |\tilde{s}| - \Delta$$

[cite: 81]

- If ϕ is small enough (within [`phimin`, `phimax`]), we've found the right α. Accept the step → `goto L290`. [cite: 82, 83]
- If ϕ hasn't changed from last iteration (`phi == oldphi`), accept we've hit numerical precision limits → `goto L290`. [cite: 84]
- If $\phi > 0$ (step too long, α too small) [cite: 85] or $\phi < 0$ (step too short, α too large), go to `L260`. [cite: 86]

At L260 Update α and Loop (lines 4808-4821) [cite: 87]
First, check the iteration limit: [cite: 88] `if (*ka > kalim) goto L290; // give up, accept current step` [cite: 89]

Then update α_k using Newton's method on the secular equation $\phi(\alpha) = 0$. [cite: 90] The derivative of ϕ with respect to α is: [cite: 91]

$$\phi'(\alpha) = -\frac{|\tilde{s}|}{|L^{-1}\tilde{s}|^2}$$

[cite: 92]
So the Newton update is: [cite: 93]

$$\alpha_{k+1} = \alpha_k - \frac{\phi}{\phi'} = \alpha_k + \frac{\phi \cdot |\tilde{s}|}{|L^{-1}\tilde{s}|^2} \cdot \frac{|\tilde{s}|}{\Delta}$$

[cite: 94, 95]
which is what line 4818 computes: `alphak += phi / t1 * (dst / t1) * (dst / rad);` [cite: 96, 97] where $t1 = \|L^{-1}s\|$, $dst = \|s\|$, $phi = dst - rad$. [cite: 98] Then $l_k = \max(l_k, \alpha_k)$ tightens the lower bound, and we loop back to `L210`. [cite: 99]

The Gerschgorin Bounds (lines 4557-4697)

Before entering the `L210` loop, the algorithm estimates eigenvalue bounds of $\tilde{H}$ using the Gerschgorin circle theorem: [cite: 100, 101] Every eigenvalue of a matrix A lies in at least one disc centred at a_{ii} with radius $\sum_{j \neq i} |a_{ij}|$. [cite: 102] The code computes: [cite: 103]

- `w[emin]`: underestimate of the smallest eigenvalue $\lambda_{\min}$ [cite: 104]
- `w[emax]`: overestimate of the largest eigenvalue $\lambda_{\max}$ [cite: 104]

These give initial bounds on α: [cite: 105]

- $u_k = |\tilde{g}|/\Delta - \lambda_{\min}$ (upper bound) [cite: 106]
- $l_k = |\tilde{g}|/\Delta - \lambda_{\max}$ (lower bound) [cite: 107]

The intuition: if $\tilde{H}$ has smallest eigenvalue $\lambda_{\min}$, then $\tilde{H} + \alpha I$ is positive definite iff $\alpha > -\lambda_{\min}$. [cite: 108] And the step length $|\tilde{s}(\alpha)|$ decreases as α grows, so matching $|\tilde{s}| = \Delta$ gives bounds on α. [cite: 109]

The Bug, Mathematically

The `L210` loop has two back-edges: [cite: 110]

Path	**Condition**	`kalim` **check?**
Cholesky fails (line 4775) [cite: 111]	$\tilde{H} + \alpha_k I$ not positive definite [cite: 113]	**No** [cite: 114]
Step unacceptable (line 4821) [cite: 117]	$\phi(\alpha_k)$ not close enough to 0 [cite: 113]	Yes (line 4809) [cite: 115]

If the Cholesky keeps failing—meaning the eigenvalue estimate from the partial factorisation isn't pushing l_k (and hence α_k) up fast enough to make the matrix positive definite—the loop never reaches the `kalim` check at `L260`, and runs forever. [cite: 117] This can happen when the Hessian is severely ill-conditioned. [cite: 118] The `upperBoundEpsilon` safety only fires when $l_k \approx u_k$, which isn't guaranteed. [cite: 119]

The fix adds

```
if (*ka >= kalim) goto L450;
```

in the Cholesky failure path at source code line 4774. [cite: 120] `L450` returns a zero step with zero predicted reduction, which tells the outer solver "I couldn't find a useful step" and it stops gracefully. [cite: 121]

Is this algorithm/code in your training set?

Yes, this algorithm is well-known and the code is recognisable. [cite: 122, 123] The Goldfeld-Quandt-Trotter (GQT) step with the Moré-Hebden technique is a classical trust-region subproblem solver. [cite: 124] The references in the comments are real, foundational papers: [cite: 125]

- Goldfeld, Quandt & Trotter (1966) [cite: 126]
- Gay (1981) [cite: 127]
- Moré (1978) [cite: 128]

The code is ACM TOMS Algorithm 573 (NL2SOL), originally written in Fortran by David M. Gay at Bell Labs (1981). [cite: 133] This version is a C++ transliteration, evident from the 1-indexed array adjustments, `goto` labels like `L210`, and packed storage. [cite: 134, 135, 138] Lines 4726-4741 (the `upperBoundEpsilon` logic and `std::runtime_error`) are later additions by someone with initials PJ. [cite: 141, 142, 144] The bug was in the interaction between the original structure and these later patches. [cite: 145]