A manual for the Modified Powell's Hybrid Method by Yoki Okawa

June 29, 2020

1 Introduction

In this document, I explain the modified Powell's Hybrid method to solve the system of non-linear equations. The modified Powell's Hybrid method has several advantages: avaiability of public domain high quality code; and popularity in various packages. This method seems "the method" used in almost all packages I checked so far. It includes IMSL, NAG, Matlab, and Octave. The algorithm is based on Powell (1970b,a). This algorithm was implemented in MINPACK, a high-quality optimization package in public domain[1]. It is implemented in Fortran 77 without the detail of the algorithm.

This document is intended to provide the code and explanation at the same time. There are many good explanations of the idea behind this algorithm. For example, see Nocedal and Wright (2000). Also, there are high-quality free codes like MINPACK. But it is hard to connect two. It is easy to get confused with what is "psi" or "flag3" in the code. So I am trying to build a bridge between algorithm and code by creating a one-to-one mapping of code and documentation.

2 Basic Idea

For the basic idea of trust region method and nonlinear optimization in general, Nocedal and Wright (2000) provide readable yet detailed explanation.

We consider a following problem.

$$\begin{pmatrix} f_1(x_1,\dots,x_n) \\ f_2(x_1,\dots,x_n) \\ \vdots \\ f_n(x_1,\dots,x_n) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Or, equivalently,

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}. \tag{1}$$

Instead of solving the equation directly, we consider the minimization of the residual function

$$r(\mathbf{x}) = \|\mathbf{f}(\mathbf{x})\|^2 = \sum_{i=1}^{n} f_i(\mathbf{x})^2$$

The nonlinear equations are solved by repeatedly solving the locally Quadratic approximation of the minimization problem of $r(\mathbf{x})$ around the point $\mathbf{x}_i$:

$$\min_{\mathbf{p}} \left[r(\mathbf{x}_i) + \tilde{\mathbf{p}}^T J^T \mathbf{f}(\mathbf{x}_i) + \frac{1}{2}\mathbf{p}^T J^T(\mathbf{x}_i) J(\mathbf{x}_i)\mathbf{p} \right] \tag{2}$$

[1]You can access MINPACK from http://www.netlib.org/minpack/

where J is a Jacobian matrix and

$$J_{kj} = \frac{\partial f_k(\mathbf{x})}{\partial x_j}$$

Quadratic approximation, which takes advantage of second derivative. Benefit of using second derivative is substantial. The first order convergence, only using first derivative will typically make the error half per iteration. This means 10^3 improvement requires 10 iterations. In the second order convergence, the error becomes square of the original error. If original error was 10^{-3}, 10^3 improvement can be obtained by only one iteration.

The challenge for second order methods are calculating the second order derivative, or the Hessian. The nice point here is that we do not explicitly calculate the Hessian of $r(\mathbf{x})$ because its functional form of the sum of the square implies simple form of the Hessian, $J^T(\mathbf{x}_i)J(\mathbf{x}_i)$ The solution to the problem is

$$\mathbf{p} = -J^{-1}\mathbf{f}(x_i), \qquad \mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{p}_i$$

This is the Newton-Raphson algorithm.

Newton-Raphson update can fail spectacularly when Jacobian changes fast. For illustration, let's consider one dimensional problem $f(x) = x^3 + 1 = 0.$. It has a unique solution at $x = -1$. Note that $f'(x) = 2x^2$, which is a strictly concave function. Someone may expect nice convergence features. However, if we start from $x_0 = 0.01$ and apply the Newton-Raphson update, x_1 is 3090. It is not approaching to the true solution at all.

Why the Newton Raphson failed? Since $f'(0.01) = 0.003$, the algorithm thinks $f(x)$ does not respond sharply with the change x. And $f(0.01)$ is 1.0303, which is not too close to zero. So the algorithm thinks it has to take a large step to make $f(x)$ zero.

The problem is that the Newton-Raphson algorithm is taking locally Quadratic approximation too seriously. The approximation is valid for the region such that Jacobian is not moving too fast. In our example, $f'(-0.1) = 0.3$, which is more than hundred times larger than the case of $x = 0.01$. So it is not a good idea to move x by three thousands. One solution is restricting the step size endogenously. We modify (2) by the following manner:

$$\min_{\mathbf{p}} \left[r(\mathbf{x}_i) + \tilde{\mathbf{p}}^T J^T \mathbf{f}(\mathbf{x}) + \frac{1}{2}\mathbf{p}^T J^T(\mathbf{x}_i)J(\mathbf{x}_i)\mathbf{p} \right] \tag{3}$$

$$\text{subject to} \qquad \|\mathbf{p}\| \leq \Delta \tag{4}$$

Δ is our parameter for the size of the region we trust the quadratic approximation. We adjust Δ smartly (details below).

3 Algorithm

3.1 Overview

This is a basic component of the algorithm. First, we decompose the Jacobian as $J = QR$.

The first step of the loop is finding the possible increment $\mathbf{p}$. This is a compromise of Newton direction and steepest descents of $F(\mathbf{x}) = \sum_{k=1}^{N} f_k(\mathbf{x})^2$ called dogleg method.

Algorithm 1 Main Algorithm (Subroutine Hybrid())

```
1: Set Initial Value of $\Delta$ and $J$
2: Do QR Decomposition of $J$: $QR = J$
3: while abs($F(\mathbf{x})$) > $tol$ do
4:    Calculate $\mathbf{p}$ given $Q, R$ and $\Delta$
5:    if $F(\mathbf{x}+\mathbf{p}) < F(\mathbf{x})$ then
6:       $\mathbf{x} \leftarrow \mathbf{x}+\mathbf{p}$
7:       Calculate $\mathbf{f}(\mathbf{x}+\mathbf{p})$
8:    end if
9:    Update $\Delta$ using $F(\mathbf{x}+\mathbf{p}), F(\mathbf{x}), J\mathbf{p}$
10:   Update $J$ using $f(\mathbf{x})$
11:   Update $Q, R$ using $J$
12: end while
```

Δ is the size of the trust region. We adjust Δ based on how well the current Jacobian is fitting the data. We decrease Δ if the current improvement is not as good as we hoped because it means our Jacobian is a poor proxy of the true nature of the objective function. We do not want to move a lot based on a poor estimate.

Next, we update the Jacobian. We usually use Broyden's Rank 1 update for fast calculation. If the last few steps was bad move, it might imply our Jacobian contains substantial errors. So we recalculate Jacobian in that case. The last step of the iteration is an update of Q, R. Because of their special structure, it is relatively fast. There are many exceptional exits of the program. We explain it at the end of the algorithm section.

Convergence This algorithm is designed so that if the Newton method fails to improve the residual function $r(\mathbf{x})$, it chooses the Steepest Descent method with a gradually smaller step length. A small step to the Steepest Descent direction always improves the objective function value. So the convergence to some local optimal point is mathematically guaranteed. The mathematical theorem does not guarantee convergence in a reasonable time and precision. But this algorithm pays special attention to rounding errors and speed. So it would also likely to converge, at least much more than the Newton-Raphson.

3.2 Update of p

This is an update of $\mathbf{p}$ by solving (5). We do not solve the problem precisely because it takes time and gain is not that big.

First, we try the Newton Correction, which is an unconditional solution of (5). The solution of the linear equation takes just $O(n^2)$ operation because R is an upper triangular matrix. We can apply successive substitutions. If it is not moving more than Δ, that is the exact solution.

If not, we try the Steepest descent direction of $r(\mathbf{x}) = \mathbf{f}(\mathbf{x})^T\mathbf{f}(\mathbf{x})$, which is $-J^T\mathbf{f}(\mathbf{x})$. In a usual Steepest Descent method (or a more sophisticated algorithm like Conjugate gradient method or Quasi-Newton method), we perform line search along the suggested direction to

Algorithm 2 Calculate $\mathbf{p}$, (subroutine dogleg)

```
1: INPUT: $R, Q, \Delta, \mathbf{f}(\mathbf{x}), \Psi$
2: OUTPUT: $\mathbf{p}$
3: Calculate Newton Correction by solving $R\boldsymbol{\nu} = -Q^T\mathbf{f}(\mathbf{x})$
4: **if** $\|\boldsymbol{\nu}\| \le \Delta$ **then**
5:    Accept Newton Correction : $\mathbf{p} \leftarrow \boldsymbol{\nu}$
6: **else**
7:    /* Try Steepest descent Correction */
8:    Calculate steepest descent Correction: $\boldsymbol{g} = -(QR)^T\mathbf{f}(\mathbf{x})$
9:    Calculate the predicted bottom at the direction of $\boldsymbol{g}$: $\mu = \|\boldsymbol{g}\|^2/\|QR\boldsymbol{g}\|^2$
10:   **if** $\mu\|\Psi^{-1}\mathbf{g}\| \ge \Delta$ **then**
11:      Move along Steepest descent correction $\mathbf{p} = \Delta\mathbf{g}/\|\mathbf{g}\|$
12:   **else**
13:      /* Accept neither. Try linear combination */
14:      Find $\theta$ such that $\|(1-\theta)\mu\mathbf{g} + \theta\Psi\nu\| = \Delta$
15:      $\mathbf{p} \leftarrow (1-\theta)\mu\mathbf{g} + \theta\nu$
16:   **end if**
17: **end if**
```

find minimum on the line. But we don't go for that because the line search requires several evaluations of the objective function, which can be costly.

Instead, we explores the special feature of $r(\cdot)$. Since $r(\cdot)$ is the sum of squares, we can derive the Hessian without performing error-prone finite difference. Namely, the Hessian is J^TJ. With Hessian and Jacobian, we can directly derive the minimum of a function that is a quadratic approximation of the objective function at the point, which is $\mathbf{x} + \mu\mathbf{g}$. In this way, we can avoid costly line search. If the minimum is outside of the permissible area, or $\mu\|\mathbf{g}\| \ge \Delta$, we accept up to Δ.

If the minimum along $\mathbf{g}$ is closer than Δ, we move a bit to the ν too. We take a linear combination of both $\boldsymbol{\nu}$ and $\mathbf{g}$ so that $\|\mathbf{p}\| = \Delta$. This can be attained by setting

$$\theta = \frac{\Delta^2 - \|\mu\mathbf{g}\|^2}{(\nu - \mu\mathbf{g}, \mu\mathbf{g}) + \sqrt{\{(\nu, \mu\mathbf{g}) - \Delta^2\}^2 + \{\|\nu\|^2 - \Delta^2\}\{\Delta^2 - \|\mu\mathbf{g}\|^2\}}}$$

Normalization In this algorithm, it is important to normalize equations so that the problem would be well behaved. Instead of considering the constraint in (5) as a ball, we can consider an eclipse constraint.

$$\|\Psi\mathbf{p}\| < \Delta$$

where Ψ is a diagonal normalization matrix. Let $\Psi\mathbf{p} = \tilde{\mathbf{p}}$. The problem (5) becomes

$$\min_{\tilde{\mathbf{p}}} \left[r(\mathbf{x}_i) + \tilde{\mathbf{p}}^T\Psi^{-1}J^T\mathbf{f}(\mathbf{x}_i) + \frac{1}{2}\tilde{\mathbf{p}}^T\Psi^{-1}J^TJ\Psi^{-1}\tilde{\mathbf{p}} \right] \tag{5}$$

$$\text{subject to} \qquad \|\tilde{\mathbf{p}}\| < \Delta \tag{6}$$

The solution of the problem above, $\tilde{\mathbf{p}}$ is equal to the solution of unconstrained dogleg problem with $\tilde{J} = J\Psi^{-1}$. So the dogleg problem with normalization is follows:

Algorithm 3 Calculate $\mathbf{p}$ with normalization

```
1: INPUT: $R, Q, \Delta, \mathbf{f}(\mathbf{x}), \Psi$
2: OUTPUT: $\mathbf{p}$
3: $\tilde{R} \leftarrow R\Psi^{-1}$
4: Apply algorithm 2. Let its solution as $\tilde{\mathbf{p}}$
5: $\mathbf{p} = \Psi^{-1}\tilde{\mathbf{p}}$
```

3.3 Revision of Δ and Recalculating J

Algorithm 4 is a update of the trust region size, Δ.

Algorithm 4 Update Δ (subroutine UpdateDelta)

```
1: Calculate predicted $f(\mathbf{x}+\delta)$:  $\boldsymbol{\phi} = \mathbf{f}(\mathbf{x}) + J\mathbf{p}$
2: Calculate Predicted $r(\mathbf{x}+\delta)$:  $\Phi = \|\boldsymbol{\phi}\|$
3: if $\frac{r(\mathbf{x})-r(\mathbf{x}+\mathbf{p})}{r(\mathbf{x})-\Phi} < 0.1$ then
4:     Change of objective function is too small. Reduce $\Delta$ by $\Delta \leftarrow \mathit{DeltaSpeed} \cdot \Delta$
5:     GoodJacobian = 0
6:     BadJacobian = BadJacobian + 1
7: else
8:     BadJacobian = 0
9:     GoodJacobian = GoodJacobian +1
10:    if GoodJacobian> 1 OR $\frac{r(\mathbf{x})-r(\mathbf{x}+\mathbf{p})}{r(\mathbf{x})-\Phi} > 0.5$ then
11:        $\Delta = \max(\Delta, 2\|\mathbf{p}\|)$
12:    else if abs$\left(\frac{r(\mathbf{x}+\mathbf{p})-\Phi}{r(\mathbf{x})-\Phi}\right) < 0.1$ then
13:        $\Delta = 2\|\mathbf{p}\|$
14:    end if
15: end if
16: if BadJacobian = 2 then
17:     Recalculate Jacobian by forward differences.
18:     BadJacobian = 0
19: end if
```

Line 3 checks if J is good or not. If J is a good prediction, the predicted reduction of the objective function $r(\mathbf{x}) - \Phi$ is almost equal to the actual reduction. If the actual reduction is less than 10% of the predicted reduction, there is something wrong with J. It is either $\mathbf{f}(\mathbf{x})$ is very nonlinear or calculated J contains too much error from the history. These errors generally become smaller if the trust region Δ shrinks.

If the reduction is relatively satisfactory, we start to weakly increase Δ. There is not too much justification for this increase. But Powell found that the result is numerically quite satisfactory. Note that usually $\|\mathbf{p}\|$ is equal to Δ. We do not want to modify Δ all the time because we might fall into the oscillation of increase $\to$ decrease $\to$ increase $\cdots$. To avoid that, GoodJacobian is checking if it is the first attempt to increase Δ. If it is after the second attempt to increase Δ and we attain a modest increase in the objective function, which is that actual reduction is more than 50% of predicted reduction, we double the trust region.

If our Jacobian is a pretty good estimate ($r(\mathbf{x}+\mathbf{p}) - \Phi$ is small), we also expand the trust region.

BadJacobian is counting the number of consecutive failures of predicting changes in the objective function. If the algorithm fails to predict the change twice consecutively, we suspect that the current Jacobian contains a serious error which came from previous updates. The previous points may be far from the current $\mathbf{x}$. We recalculate it based on the forward difference from scratch, instead of updating it.

3.4 Update of J

We update Jacobian using the Broyden's rank 1 update. For the detail, see Numerical Recipes Ch. 9.7., Broyden's method. :

$$J \leftarrow J + \frac{1}{\|\mathbf{p}\|^2}(\mathbf{f}(\mathbf{x}+\delta) - f(\mathbf{x}) - J\mathbf{p})\mathbf{p}^T$$

3.5 QR Decomposition

QR decomposition is decomposing a matrix A to multiple of Q, orthogonal matrix ($Q^{-1} = Q^T$), and upper triangular matrix R. So $A = QR$. It is useful because solving $Ax = b$ takes $O(n^2)$ step once we know QR decomposition of A, which is faster than $O(n^3)$. Although QR decomposition itself takes $O(n^3)$, once we calculate the decomposition, its update only takes $O(n^2)$. So it is useful when we have to solve the linear equation repeatedly with slightly different coefficients. It is the case when we calculate the Newton direction, which requires to solve $J\delta = \mathbf{f}(\mathbf{x})$ in every iteration. In this subsection, I assume that input is a square matrix. The Fortran code is written in the way it accepts the rectangular matrix.

3.5.1 Decomposition

This is called at first and whenever J is recalculated using finite difference. The code uses Householder transformation repeatedly. Let a_j as an arbitrary vector and we want to find P_j such that P_j is orthogonal and $P_j a_j = (b_1, 0, \ldots, 0)^T$. We can set

$$P_j = I - \frac{2}{\|u_j\|^2} u_j u_j^T, \qquad u_j = a_j - \|a_j\| e_1$$

where I is identity matrix and $e_1 = (1, 0, \ldots, 0)^T$ is a unit vector. We can apply this repeatedly until all matrix becomes upper triangular. Many books about numerical algorithm covers QR decomposition with Householder transformation.

3.5.2 Update

The update is called at the end of each iteration. Our goal is given

$$A^{next} = A + uv^T, \qquad A = QR$$

find Q^{next} and R^{next} such that $Q^{next}R^{next} = A^{next}$. For more information, see Bjorck and Dahlquist (2008) section 8.4.

transform a bit Let $w = Q^T u$. Using this,

$$A^{next} = Q^{next}(R + wv^T)$$

We are going to make $R + wv^T$ upper triangular matrix using Givens transformation.

Givens transformation of w We find a sequence of Givens transformation such that

$$P_1 \dots P_{n-1} w = \alpha e_1.$$

Givens transformation is very similar to Jacobi transformation. It is a matrix defined by three parameters (k, l, θ) such that (i, j) element is

$$P(k,l,\theta)_{i,j} = \begin{cases} \cos\theta & i = j = k \\ \cos\theta & i = j = l \\ -\sin\theta & k = i, l = j \\ \sin\theta & k = j, l = i \\ 1 & i = j \neq k, i = j \neq l \\ 0 & otherwise \end{cases}$$

It is zero other than diagonal elements, (k, l) element, and (l, k) element. P is orthogonal matrix. Consider $P(n-1, n, \theta)w$. nth element of $P(n-1, n, \theta)w$ is zero if

$$\cos\theta = \frac{1}{\sqrt{t^2+1}}, \qquad \sin\theta = t\cos\theta, \qquad t = \frac{w_n}{w_{n-1}}$$

We can repeat this until all element other than the first element is zero. Let

$$P^w = P_1 \dots P_{n-1}$$

Transform upper Hessenberg matrix to upper triangular matrix Now,

$$A^{next} = Q(P^w)^T(P^W R + P^w wv^T)$$

Note that elements in $P^w wv^T$ are zero other than the first row and $P^W R$ is upper Hessenberg matrix. So $P^W R + P^w wv^T$ is an upper Hessenberg matrix. We are going to eliminate (1,2) elements to (n-1,n) elements using Givens transformation. Let $H = P^W R + P^w wv^T$. To eliminate (1,2) element, we need Givens transformation such that

$$\cos\theta = \frac{1}{\sqrt{t^2+1}}, \qquad \sin\theta = t\cos\theta, \qquad t = \frac{h_{21}}{w_{11}}$$

We can repeat this until all lower triangular elements are zero. Let P^H as the accumulation of the transformation. We complete the algorithm by

$$Q^{next} = Q(P^w)^T(P^H)^T, \qquad R^{next} = P^H(P^W R + P^w wv^T)$$

3.6 Termination of the Algorithm

There are several conditions for the termination of this algorithm.

Successful Convergence We call it successful if

$$\frac{1}{\sqrt{n}}\|\mathbf{f}(\mathbf{x})\| = \sqrt{\frac{1}{n}\sum_{i=1}^{m}(f_i(\mathbf{x}))^2} < ftol$$

This means that the average squared residual is small. If this condition is satisfied, $|\mathbf{f}_i(\mathbf{x})|$ is at most $n \cdot ftol1$

Trust region size shrinks to the tolerance level The program stops execution if $\Delta < xtol(\|\mathbf{x}\| + xtol)$, the relative size of the trust region is smaller than $xtol$. This happens if the Jacobian changes rapidly and wildly around the point the program terminated. This is the most common type of unsuccessful termination. It can happen for the following reasons:

- The problem you are going to solve has large second order derivatives. In this case, Jacobian is sensitive to where it is evaluated. The local quadratic approximation is inappropriate. This might be the case if you have highly discontinuous Jacobian, for example.
- You set the $ftol$ too small. You might get this message even if the algorithm is in indeed the solution. If $ftol$ is too small, the algorithm can not attain it with the precision required.
- You set the speed of shrinking Δ too small.
- Your subroutine to be solved contains bugs. My personal experience suggests this explains most of this error message.

Here are possible solutions:

- Check the $\mathbf{f}(\mathbf{x})$ and see if it is small enough or not.
- Increase *DeltaSpeed*
- Increase *ftol*
- Decrease $JacobianStep$
- Use different initial guesses
- If you are using single precision real numbers, use double precision instead.

Too much function call If the number of function call exceeds MaxFunEval, the program stops the execution.

Algorithm reaches Local minima If the program terminates for the reason other than "Successful Convergence", we recalculate the gradient of the objective function $\nabla r(\mathbf{x}) = J^T\mathbf{f}(\mathbf{x})$ by finite difference. If $\|\nabla r(\mathbf{x})\| < gtol(\|\mathbf{x}\| + gtol)$, we are at the locally optimal point, although average residual is not close enough to zero in the tolerance required. This happens if the algorithm is captured in the local minima. The solution is trying the different initial guess. This can also happen if $ftol$ is too small to attain. You can check that by seeing the $\mathbf{f}(\mathbf{x})$ at the output.

4 Usage of the Subroutine

I explain inputs and outputs of this subroutine. Many arguments are optional.

fun Subroutine which returns the residual vector. It should take two arguments, and first argument should be $\mathbf{x}$ and second argument should be the $\mathbf{f}(\mathbf{x})$.

x0 Initial value of $\mathbf{x}$

xout Solution of the least square problem.

info (optional) Variable for the information of the termination of the algorithm. For the detail of the conditions, see the Termination of the Algorithm.

- **info = 0** Improper input values
- **info = 1** Successful convergence.
- **info = 2** First order optimality satisfied
- **info = 3** Trust region size shrinks to the tolerance level
- **info = 4** Too much function call

fvalout (Optional) Output vector of residuals at $\mathbf{x} = xout$.

Jacobianout (Optional) Output Jacobian Matrix at $xout$

xtol (Optional) Tolerance of the x. If trust region size is smaller than $xtol(\|\mathbf{x}\| + xtol)$, the program terminates. If this is not provided, it would be set to the default value of 10^{-8}

ftol (Optional) Tolerance of the residual. The program terminates if $\sqrt{\sum_{i=1}^{m}(f_i(\mathbf{x}))^2/m} < ftol$. If this is not provided, it would be set to the default value is 10^{-8}.

gtol (Optional) Tolerance of the gradient. If $\|\nabla r(\mathbf{x})\| < gtol(\|\mathbf{x}\| + gtol)$ holds, we regard it first order optimality is satisfied. The default value is 10^{-8}

JacobianStep (Optional) Relative step size for calculating the Jacobian by Finite difference. Default value is 10^{-3}.

display (Optional) Option to specify the display preference. If display is 2, the algorithm shows the iteration. If display is 0, the algorithm shows nothing. Default value is 0.

maxFunctionCall (Optional) Number of function to be called before terminating the algorithm. Default value is $100n$, where n is number of equations.

noupdate (Optional) Dummy variable for the Broyden's rank 1 update. If noupdate is 1, the algorithm recalculate Jacobian with finite difference in each iteration. This would almost always make the algorithm converged with fewer iteration. But this does not mean it would speed up in terms of CPU time because in general the calculation of the Jacobian is costly if the evaluation of the objective function is costly or n is large. Turning off the update can speed up the total computation time when n is very small, (say less than 5). Default value is 0, so the Broyden's update is the default.

DeltaSpeed (Optional) Controls the speed of Δ when it shrinks. In algorithm 4, line 4, the trust region size Δ shrinks if the actual reduction do not match with predicted reduction. DeltaSpeed controls how fast it should shrink. It should always be always less than one. The smaller the value is, the faster the Δ shrinks. The default value is 1/4.

5 Other issues

5.1 Improving Performance

Current implementation prefer the readability of the code to performance. If you are dealing with large scale problems, you might want to change some parts of the code. In general, this code is written for the case that number of parameters to be solved is not that large, (for example, less than 50) and evaluation of the objective function is costly. If this is not your problem, you might be able to get better performance for the following modifications.

QR factorization Other than the evaluation of the objective function, QR decomposition would take most of the time in this algorithm. It is called at first and called occasionally afterward if the current approximation is suspected to contain substantial errors. If n or m is large (more than a few hundred n or more than 10,000 m) and the evaluation of the objective function is not expensive, MINPACK's slow implementation of QR decomposition is likely to be a bottleneck. You should consider using high-quality linear algebra package like LAPACK. Also, LAPACK routine would improve the numerical stability of QR decomposition.

Better solution of Locally quadratic problem Also, faster speed can be attained by improving the dogleg method to the nearly exact solution using Cholesky factorization. It takes more time than the dogleg method per iteration. But improvement per iteration will be better. This appropriate when the evaluation of the function is costly and the number of the variables are small.

Memory The memory should not be a concern for modern computers if n is less than 1000.

If n is larger than that, it might create a problem. This code stores a few n by n matrixes. To give you an idea for how serious this constraint is, here is the memory required to store matrixes: To store a 1,000 by 1,000 matrix with 8 bytes (double) precision, it requires 8 Megabytes of memory. For 5,000 by 5,000 matrix, that is 190 Megabytes and for 10,000 by 10,000 matrix, that is 762 Megabytes. In case memory binds, it is not too difficult to decrease the memory usage by 50% or more with a little modification. For more memory savings, probably you need a substantial rewrite of the code based on conjugate gradient method instead of Newton Based or use sparse matrix.

5.2 Comparison with various methods

Here is the comparison of various methods with this method.

Compared to Newton-Raphson method with BFGS, DFP, or BHHH Great advantage of Newton-Raphson type method is its speed. But it is unstable, even if we update Jacobian/Hessian smartly using BFGS, DFP, or BHHH. The Newton Raphson method just has local convergence. It might fall into a cycle or diverge even if the function is smooth and globally concave unless the initial guess is sufficiently close to the true value. So here is the origin of the agony of initial guess. Also, Newton's method might take you to the saddle point. (see Bjorck and Dahlquist (2008) section 11.3 for more discussion) On the other hand, the modified Powell's Hybrid has global convergence property. It always converges if the first derivative is continuous and bounded, no matter what the initial point is. This is a great improvement in terms of convergence. For speed, the Hybrid method tries to use Newton-Raphson update whenever it looks safe. The Hybrid method is as good as that in Newton-Raphson if the initial guess is good. If the initial guess is not so good, only the Hybrid method has guaranteed converges.

Compared to Newton-Raphson with Line search (see Bjorck and Dahlquist (2008) section 11.3 for more discussion) To obtain global convergence, there are two ways. One is the trust region method which the Hybrid method is relied on, and the other is the line search method. They are different in what to do after we get the direction for improvement. For line search, it searches the extremum along the direction. And trust-region picks step size without function evaluation by local quadratic approximation.

Compared to Conjugate Gradient and its derivatives The Conjugate Gradient method stops relying on second order derivative entirely. Second order derivative is n by n matrix, which won't fit in the memory if n is very large, such as the deep learning. The Powell's method cannot be applied for that case.

Compared to Ameba/Downhill Simplex Their advantage is an ability to deal with discontinuity. If the objective function contains severe discontinuities near the solution, you should consider this type. But it is VERY slow. In my experience, the Hybrid method is not that fragile to the mild discontinuity in both objective function and its derivative because of its trust region feature, although there is no mathematics to support the claim. I suggest trying to the Hybrid method first because it is much faster. Sophisticated termination criterion might be fooled by the discontinuity so you might want to restart it once it converges.

Compared to Grid Search/Simulated annealing The global convergence property of the Hybrid method is that it would converge to local extrema. Like any derivative-based method, it does not guarantees the global property of the point obtained. If there are many local extrema and you need a global solution, probably there is no choice other than Grid search or other globally convergent methods like the simulated annealing.

Compared to bisection/Brent's method If the problem contains only one equation, we have completely different algorithms. Bisection and its improvement, Brent's method, is regarded as robust algorithms that can be applied to highly discontinuous functions. The problem is that it requires two initial guesses which brackets the solution. It is not always easy to find them. I am not sure if they are so robust or fast if we consider the step of finding the bracket. The hybrid algorithm is faster than these algorithms with only one initial guess. But it may be trapped at the local extrema of the function. So it is your call to pick.

Fixed point approach Su and Judd (2008) is fiercely criticizing this approach. It makes some sense. But their solution, extensive usage of the canned package is not always a good idea. It is not so hard to understand the algorithm, once we have nice documentation.

6 Glossary

Here, $\mathbf{f}(\mathbf{x})$ is system of equations which we want to make it zero, and $F(\mathbf{x})$ is scalar objective function which we want to minimize.

Jacobian Jacobian is a matrix of first order derivatives, which comes from a vector valued function.

$$J_{ij} = \frac{\partial f_i(\mathbf{x}))}{\partial x_j} \qquad J = \begin{pmatrix} \frac{\partial f_1(\mathbf{x}))}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x}))}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(\mathbf{x}))}{\partial x_1} & . & \frac{\partial f_n(\mathbf{x}))}{\partial x_n} \end{pmatrix}$$

Gradient Gradient is a vector of first order derivatives, which comes from a scalar valued function. This is another name of first order conditions. $\bigtriangledown F(\mathbf{x}) = (\frac{\partial F(\mathbf{x}))}{\partial x_1}, \ldots, \frac{\partial F(\mathbf{x}))}{\partial x_N})$

Hessian Hessian is a matrix of second order derivatives, which comes from a scalar valued function. $H = \bigtriangledown^2 F(\mathbf{x}) = \triangle F(\mathbf{x})$. $H_{ij} = \frac{\partial^2 F(\mathbf{x})}{\partial x_i \partial x_j}$. Hessian is a symmetric and positive semidefinite.

Hessian and Jacobian Since the optimization problem is the same as finding zero in the first order conditions, Hessian of the objective function is Jacobian of first order conditions. But Jacobian from the first order conditions behaves better than ordinary Jacobian, because Hessian has several special features. Many variants of the Newton-Raphson algorithm exploit the fact for an efficient algorithm.

Newton-Raphson algorithm This algorithm is both for solving a system of nonlinear equations and optimization. If it is applied for solving a system of nonlinear equations, it uses the locally linear approximation of the equations. Linear approximation requires a slope, which is Jacobian. If it is applied for the optimization problem, it uses a locally linear approximation to the first order conditions. Jacobian of the first order conditions is Hessian. So it requires Hessian.

BFGS/DFP update Smart way to calculate Hessian in the Newton Raphson algorithm for optimization. BFGS update is weakly better than DFP update in a sense that BFGS and DFP performs almost identically for most of the problems and there are some cases BFGS is clearly better than DFP update.

BHHH update Another smart way to calculate Hessian in the Newton Raphson algorithm for maximum likelihood problems. Although this is popular for econometricians, plain-vanilla implementation of the BHHH update is usually inferior to BFGS/DFP update.

Gauss-Newton's algorithm Yet another smart way to calculate Hessian applicable to nonlinear least square problem.

Steepest Descent algorithm/Cauchy algorithm This algorithm is for solving a optimization problem. It simply picks a direction which is inverse of gradient. If the step size is sufficiently small, it always improves the value of the objective function. For the determination of step size, a line search is usually used. It is also called the Cauchy algorithm.

References

Bjorck, Ake and Germund Dahlquist, *Numerical Methods in Scientific Computing Volume II* 2008.

Nocedal, Jorge and Stephen Wright, *Numerical Optimization*, Springer, 2000.

Powell, Michael J.D., "A Fortran Subroutine For Solving Systems of Nonlinear Algebraic Equations," in Philip Rabnowitz, ed., *Nonlinear Methods for Nonlinear Algebraic Equations*, Gordon and Breach, Science Publishers Ltd., 1970, chapter 7.

_ , "A Hybrid Method For Nonlinear Equations," in Philip Rabnowitz, ed., *Nonlinear Methods for Nonlinear Algebraic Equations*, Gordon and Breach, Science Publishers Ltd., 1970, chapter 6.

Su, Che-Lin and Kenneth L. Judd, "Constrainted Optimization Approaches to Estimation of Structural Models," January 2008, (1460).