Improving the Efficiency of the Levenberg-Marquardt Algorithm Using Partial-Rank Jacobian Updates

Michael Zairan

A senior thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Dr. Mark Transtrum, Advisor

Department of Physics and Astronomy

Brigham Young University

April 2016

ABSTRACT

Improving the Efficiency of the Levenberg-Marquardt Algorithm
Using Partial-Rank Jacobian Updates

Michael Zairan
Department of Physics and Astronomy, BYU
Bachelor of Science

Fitting non-linear models to data is a notoriously difficult problem. The standard algorithm, known as Levenberg-Marquardt (LM), is a gradient search algorithm based on a trust region approach that interpolates between gradient decent and the Gauss-Newton methods. Algorithms (including LM) often get lost in parameter space and take an unreasonable amount of time to converge, especially for models with many parameters. The computational challenge and bottleneck is calculating the derivatives of the model with respect to each parameter to construct the so-called Jacobian matrix. We explore methods for improving the efficiency of LM by approximating the Jacobian using partial-rank updates. We construct an update method that reduces the computational cost of the standard Levenberg-Marquardt routine by a factor of .64 on average for a set of test problems.

Keywords: Data-Fitting, Optimization, Levenberg-Marquardt, Gauss-Newton, Broyden's Method, Partial-Rank Update

ACKNOWLEDGMENTS

I would first like to thank my thesis advisor Dr. Transtrum from the College of Physcial Math and Sciences at BYU. He believed in me and patiently advised me as I asked endless questions to understand the research better. He also allowed me to be curious and explore the topic, but steered me in the right direction along the way.

I would also like to thank Alex Shumway for writing code for the singular value decomposition method and for pushing me to work harder and be better.

Contents

Chapter 1

Background

Physicists often want to fit models to experimental data. Motivated by physical considerations, they generally have an idea of the sort of function that will fit the data points best. These functions often depend on several parameters θ, where θ could be a vector. The goal is to find the values of each parameter that make the function fit the data best. This is often done by a method known as least squares. The quality of the fit is measured by finding how far off each data point is from the model via

$$r_i(\theta_i) = \frac{d_i - y_i}{\sigma_i}, \tag{1.1}$$

where d_i is the i^{th} data point measured with uncertainty σ_i, $y_i(\theta)$ is the corresponding model prediction, and $r_i(\theta)$ is the so-called residual that measures the deviation between the model and the experiment. The sum of each residual squared yields a value called the cost, defined by

$$C = \frac{1}{2}\sum_i r_i^2. \tag{1.2}$$

The set of values for the parameters that minimizes C are known as the best fit by the least squares method (Bates et al. 1988).

As a precursor to the commonly used Levenberg-Marquardt algorithm, we first consider the Gauss-Newton method for finding the best fit (Hartley 1961). The method starts at initial point in parameter space θ_0 and calculates $r(\theta_0)$ through 1.1. Ideally, θ_0 should be close to the best fit values of the parameters in parameter space. The Taylor series approximation of $r(\theta)$ at a point θ nearby θ_0 is

$$r_m(\theta) = r_m(\theta_0) + J_{m\mu}(\theta_\mu - \theta_{\mu 0}) \tag{1.3}$$

where J is the Jacobian matrix given by

$$J_{m\mu} = \frac{\partial r_m}{\partial \theta_\mu}. \tag{1.4}$$

The next step in the algorithm is to compute the cost. Using the linear approximation, the total cost by the least squares method is

$$C = \frac{1}{2}\sum_m [r_m(\theta_0) + \sum_\mu J_{m\mu}(\theta_\mu - \theta_{\mu 0})]^2. \tag{1.5}$$

Taking the derivative of both sides with respect to θ_ν yields:

$$\frac{\partial C}{\partial \theta_\nu} = \sum_m [r_m(\theta_0) + \sum_\mu J_{m\mu}(\theta_\mu - \theta_{\mu 0})] J_{m\nu}. \tag{1.6}$$

At the minimum, $\frac{\partial C}{\partial \theta_\nu} = 0$, thus, we have

$$\sum_m [r_m(\theta_0) J_{m\nu} + \sum_\mu J_{m\mu} J_{m\nu}(\theta_\mu - \theta_{\mu 0})] = 0, \tag{1.7}$$

which can be written without indices as

$$J^T r_0 + (J^T J)(\theta - \theta_0) = 0. \tag{1.8}$$

Solving for $\theta - \theta_0$ gives

$$d\theta = (\theta - \theta_0) = -(J^T J)^{-1}(J^T r). \tag{1.9}$$

Each iteration of the Gauss-Newton algorithm calculates a step $d\theta$ by 1.9, then iteratively updates the parameter values $\theta \to \theta + d\theta$ until it has reached the minimum. A potential problem arises if $J^T J$ is nearly singular. In such a case, the eigenvalues of $(J^T J)^{-1}$ are large, making $\theta - \theta_0$ large. This makes the Taylor series approximation in Eq. 1.3, which assumed $\theta - \theta_0$ is small, a poor estimate. In order to prevent this from happening, Levenberg added a term λI to the matrix $J^T J$ in 1.9 (Moré 1978). If v is an eigenvector of $(J^T J)$ with eigenvalue λ_1:

$$(J^T J)v = \lambda_1 v, \tag{1.10}$$

then the new matrix $(J^T J + \lambda I)$ satisfies

$$(J^T J + \lambda I)v = (J^T J)v + \lambda I v = \lambda_1 v + \lambda v, \tag{1.11}$$

so that v is an eigenvector of $J^T J + \lambda I$ with eigenvalue $\lambda_1 + \lambda$. Thus $J^T J + \lambda I$ will have no eigenvalues smaller than λ. It can also be shown that the step size $d\theta$ decreases monotonically with λ. The resulting algorithm is known as the Levenberg-Marquardt algorithm (Moré 1978).

The algorithm controls the size of the step $d\theta = -(J^T J + \lambda)^{-1} J^T r$ by modifying λ. A large value of λ corresponds to large eigenvalues and a small step size, while small values of λ correspond to smaller eigenvalues and large steps. The art of the Levenberg-Marquardt algorithm is fine-tuning λ to take the correct step size. If the step is too large and ends up increasing the cost, the algorithm will reject the step and increase λ to try to take a smaller step in the next iteration.

In summary, the algorithm proceeds by applying the following steps:

1. Choose an initial value of parameters θ and damping parameter λ.

2. Calculate the residuals $r(\theta)$, the cost $C(\theta)$ and the Jacobian $J(\theta)$.

3. Calculate the step $d\theta$ using 1.9.

4. Evaluate the cost $C(\theta + d\theta)$ at the new parameters.

5. If $C(\theta + d\theta) < C(\theta)$ then let $\theta = \theta + d\theta$ and $\lambda = \lambda/3$ Otherwise, do not change θ and set $\lambda = 2\lambda$

6. Repeat steps 2 - 5 until a minimum is found.

The value of λ also controls the direction of the step. Note that the gradient of the cost is given by $\nabla C = J^T r$. For $\lambda = 0$, dθ points in the Gauss-Newton direction:

$$d\theta = -(J^T J)^{-1} \nabla C. \tag{1.12}$$

As we have seen, this will converge to the minimum of the cost if the algorithm is already close, but at farther distances it can actually lead the algorithm to a point with a higher cost. In such cases, the algorithm will reject the step and increase λ. As λ increases, the step moves away from the Gauss-Newton direction and towards the $-\nabla C$ direction, which is the steepest decrease in the cost. This can be seen because if λ»1, $(J^T J + \lambda)^{-1} \approx \frac{1}{\lambda} I$, so that

$$d\theta = -\frac{1}{\lambda}(\nabla C). \tag{1.13}$$

Therefore, the algorithm uses λ both to interpolate between the Gauss-Newton direction and the direction of the gradient and to modify the step size.

Geometrically, this process can be visualized for a typical cost surface, as in Figure 1.1. Each location on the surface represents a set of values for θ and the colors show cost contours, with the cooler colors representing lower costs. The algorithm seeks to reach the lowest contour on the surface which represents the minimum cost possible, represented by the dark blue color. In the figure, the black line represents all possible steps that (found by varying λ) the algorithm can take if the algorithm's current location is the red dot. We can see from this how increasing λ moves away from the Gauss-Newton direction, which extends well into the orange contour, and towards the direction of -∇C, decreasing the size of the step in the process.

The process is simple enough for problems with few parameters. However, when there are many parameters to consider, it becomes much more difficult and computationally costly to explore the parameter space and find the right fit. In science today, the trend is to use increasingly bigger models. Weather models, for example, could use hundreds of parameters to make forecasts in the

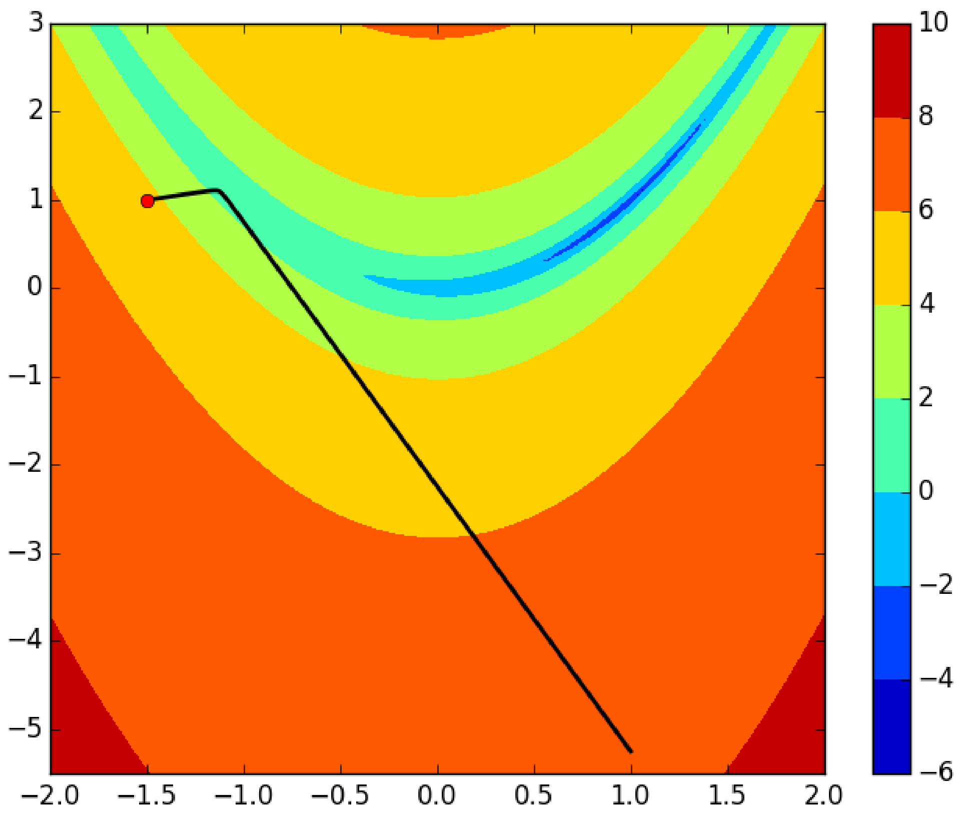

Figure 1.1 A graph illustrating a typical cost contour. If the algorithm's current position is the red dot, the possible steps it can take by varying lambda are given by the black line, with points farther from the red dot representing steps with small λ values and the Gauss-Newton direction. Nearer points represent steps taken with larger λ values shift towards the direction of -∇C. The color bar uses a log scale.

weather (Soman et al. 2010).

One approach that attempts to reduce the computational cost of the Levenberg-Marquardt algorithm is Broyden's method. Since the most computaionally expensive part of the algorithm is the calculation of the Jacobian matrix, Broyden's method only calculates it during the first iteration. Instead of recalculating the full Jacobian matrix in subsequent steps, it uses partial information from the Jacobian contained in the previous iteration to update the Jacobian using a rank-one update (Gay & Schnabel 1978). If J_{i-1} is the Jacobian from the previous iteration and J_i is the Jacobian at the current parameter values, then J_i can be estimated by

$$J_i \approx J_{i-1} + \frac{\Delta r_i - J_{i-1} d\theta_i}{|d\theta_i|^2} d\theta_i^T \tag{1.14}$$

where $\Delta r_i = r_i - r_{i-1}$ is the change in the residual vector and $d\theta_i = \theta_i - \theta_{i-1}$.

Broyden's method can greatly reduce the computational cost, but the Jacobian approximation can get increasingly poor with each iteration. Thus, the problem researched in this thesis is finding a more efficient algorithm: one that reduces the computational cost of calculating J without sacrificing accuracy. There is a large gap in the information content between Broyden's method, which calculates only the first Jacobian and approximates the rest by updating in one direction, and the full Jacobian update at every iteration.

There is considerable evidence that the right balance could exist between these two extremes. First, Broyden's method is proven to converge for the related problem of root finding (Dennis 1971). If J is a square matrix (i.e. there are the same number of parameters as data points), then there may exist some value of θ for which $r(\theta) = 0$. However, the method only has mixed success when there are more residuals than parameters (Transtrum & Sethna 2012). This suggests that with the right information, an approximation can do very well.

Recently, the Levenber-Marquardt algorithm has been improved by including a second-order correction known as the geodesic acceleration (Transtrum & Sethna 2012). Geodesic acceleration gives us more reason to believe that not all information is essential to achieving accuracy. To calculate the second-order approximation to $d\theta$, the formula is

$$d\theta = d\theta_1 \delta t + \frac{1}{2} d\theta_2 \delta t^2. \tag{1.15}$$

The $d\theta_2$ vector gives a more accurate approximation for $d\theta$, but it only requires one additional calculation of r once J is calculated. That substantial improvements can be made using little information suggests that an efficient method for updating the Jacobian may be possible.

Lastly, a higher-order singular value decompostion of the second-derivative tensor, A, suggests that changes in J are of low rank. Using the definition for J given by Eq. 1.4, we have

$$A = \frac{\partial}{\partial \theta_\nu} J_{m\mu} = \frac{\partial^2 r_m}{\partial \theta_\mu \partial \theta_\nu}. \tag{1.16}$$

This tensor (the Jacobian of the Jacobian) tells us how the Jacobian is changing, and it often reveals that there are many directions in which the Jacobian barely changes at all. That means information from J used in previous iterations could be still be valid as an approximation for J in the next iteration. This is another reason to believe the right balance between derivative calculation and approximation could yield more efficient results.

Chapter 2

Comparing Algorithms

2.1 Measuring Success

Some sort of objective measure is necessary to determine how successfully a method balances accuracy and computational cost. We want a measure that can extrapolate in the limit of large models, i.e., those with many parameters. In this way, we can efficiently test methods on small problems and extrapolate those results to more realistic models.

One criterion to consider would be how often the algorithm thinks it has been successful. As the Levenberg-Marquardt algorithm moves around parameter space in search of the minimum cost, it is possible that it will get lost. There are many potential stopping criteria that one could consider to that signal to the algorithm to terminate. The ones we implement are a maximum number of steps for the algorithm to take (maxstop), a minimum step size (mindx) and a minimum change to the cost (ftol) to stop the algorithm when the position in parameter space is hardly changing, and a maximum value for λ (maxlam). Also, we implement a cutoff value for the cost that tells the algorithm it is close enough to the known minimum cost for a specified test problem (min cost). In practice, it would be impossible to know the minimum cost before running the algorithm, but implementing mincost works for the purpose of running trial problems. Of the stopping criteria, mincost, ftol, and mindx are considered successes, while the others would be signs that the algorithm got lost and are considered failures. The reason why the algorithm stopped is helpful in determining the success of the algorithm, so we classify each run of the algorithm as a "claimed succes" or "claimed failure."

Altough the algorithm stops for a reason that signifies a success, that does not necessarily mean that it has found the minimum cost or is near it. Another criterion to determine the success of the algorithm would be whether it is an "objective success" or "objective failure." To determine whether or not the final cost that the algorithm ended up at was close enough to the minimum cost to constitute an objective success, we set a threshold value for the cost below which all costs are considered objective successes. The threshold value is found by multiplying the known minimum cost C_{min} by a constant β that is specific to each problem. If M is the number of data points and N is the number of parameters, then β is found by using the Snedecor-Fisher F-distribution via the formula:

$$\beta = \frac{N}{M-N} F(\alpha, N, M-N) + 1. \tag{2.1}$$

If the algorithm found a cost $C < \beta C_{min}$ then we classify it as an objective success.

The last factor to consider is how hard the algorithm has to work to find the minimum. We use a measure called the effictive number of Jacobian evaluations ($njev_{eff}$). Because a Jacobian calculation amounts to N function evaluations (where N is the number of parameters), we define $njev_{eff}$ by

$$njev_{eff} = njev + \frac{nfev}{N}, \tag{2.2}$$

Where $njev$ is the number of full Jacobian evaluations and $nfev$ is the number of residual function evaluations.

We seek a measure of algorithm performance that accounts for the likelihood of the algorithm having an objective success while also accounting for the computational cost of that success. Ultimately, the number we decide to use was a number we called the effieciency score. The efficiency score is the expected number of effective Jacobian evaluations necessary to achieve one objective success. We calculate the average number of algorithm runs to achieve an objective success by modeling algorithm's success rate as a geometric random variable with probability p (found through trial data) of finding the minimum. The probability of achieving a success on the k^{th} attempt is $(1-p)^{k-1}p$. Thus, the expected value of the random variable X is

$$E(X) = \sum_{k=1}^{\infty} k(1-p)^{k-1}p. \tag{2.3}$$

Exapanding this yields

$$E(X) = p[\sum_{k=1}^{\infty}(1-p)^{k-1}p + \sum_{k=2}^{\infty}(1-p)^{k-1}p + \sum_{k=3}^{\infty}(1-p)^{k-1}p...]. \tag{2.4}$$

Notice that each summation is an infinite geometric series with ratio $(1-p)$. Since the value of a geometric series is given by $\sum_{k=1}^{\infty} x^p = 1/(1-x)$ if $x < 1$, we can apply that formula to every term in the expected value to get

$$E(X) = p[1/p + (1-p)/p + (1-p)^2/p + ...], \tag{2.5}$$

which simplifies to

$$E(X) = 1 + (1-p) + (1-p)^2 + ... = \sum_{k=1}^{\infty}(1-p)^{k-1} = 1/p. \tag{2.6}$$

Thus, the number of attempts to achieve an objective success on average is $1/p$ where p is the objective success rate. Multiplying this by the average amount of effective Jacobian evaluations will yield the efficiency score, giving us a way to compare algorithms while taking into account both the computational cost and the accuracy.

2.2 Test Problems

Once a way to compare algorithm efficiency has been established, we are equipped to try different algorithms on test problems. We choose six previously studied test problems of varying difficulty from the minpack test suite (see (Averick et al. 1991) for details). The first problem we include we call IAD. It is a five parameter, 40 data point problem that solves a differential equation to simulate thermal reactions. Another problem included is EDF, which also has five parameters. This problem models exponential data fitting and has 33 data points. As a variation of this problem, we test EDFproj, which has the same properties but solves for the two linear parameters in terms of the other parameters, reducing it to a three parameter problem. The problem GDFproj that we include is like EDFproj in that it models exponential data and reduces the problem by solving for linear parameters. The difference, however is that it models Gaussian exponentials instead. The problem has 65 data points and four parameters. The last two problems we include from the minpack test suite are one we call ATRproj, which is a 16 data point, two parameter problem that simulates thermistor resistance, and another that we call AERproj, which is an 11 data point, two parameter problem that simulates enzyme reactions.

Since the problems were created in the 1990's, we also include a newer test problem called DANN3, which models an artificial neural network in the brain. The problem is typical of the kind of non-linear problems one would find in neuroscience, and involves solving the differential equation

$$\frac{\partial}{\partial t} y_i = \tanh(\sum_j (w_{ji} - y_j) - y_i). \tag{2.7}$$

We test the algorithms on these problems by generating an ensamble of starting points that are different distances from the minimum cost and running the algorithms from each point. In order to test the algorithms on more test problems, we also create two versions of each problem, an "easy" version and a "hard" version. The difficulty of a problem is altered by making the starting points farther away from the minimum cost, making the algorithms work harder to find the minimum. The efficiency score is calculated based on the results of these trials.

Chapter 3

Update Methods and Results

3.1 Partial-rank updates

We want to use a rank-one update to the Jacobian from the previous iteration to only update in one direction (equivalent to $1/N^{\text{th}}$ the computational cost of a full Jacobian update). Given the old Jacobian J_O, we can calculate the directional derivative V of the true Jacobian J_{true} for the next iteration in the direction v without knowing J_{true}. This is because the formula for the directional derivative is found analytically beforehand and the problems have a function that uses that formula to compute V given v. Therefore, we have the directional derivative

$$J_{true}v = V. \tag{3.1}$$

We would want that to hold true for our approximation of the new Jacobian, J_{new}, so

$$J_{new}v = V \tag{3.2}$$

should be true, but we would want other directions to remain unchanged. For any direction u orthogonal to v,

$$J_{new}u = J_O u \tag{3.3}$$

should hold as well. The correct approximation for J_{new} is ultimately given by

$$J_{new} = J_O + \frac{V - J_O v}{v^2} v^T. \tag{3.4}$$

It should be noted that the approximate Jacobian given by 3.4 is actually the same formula as the approximation of the Jacobian used by Broyden's method in 1.14. Broyden's method just chooses the direction of the step $d\theta$ as the direction v in which to update each iteration. We will now show the approximation satisfies condition 3.2 and 3.3. First, we will establish that it satisfies 3.2. Plugging 3.4 into 3.2 yields

$$J_O v + \frac{V - J_O v}{v^2} v^T v = V. \tag{3.5}$$

$v^T v = v^2$ so that will cancel with the denominator, leaving

$$J_O v + V - J_O v = V \implies V = V. \tag{3.6}$$

The second condition is similarly satisfied. Plugging in 3.4 into 3.3 yields

$$J_O u + \frac{V - J_O v}{v^2} v^T u = J_0 u. \tag{3.7}$$

Since u and v are orthogonal, $v^T u = 0$, this leaves only

$$J_O u = J_0 u. \tag{3.8}$$

Therefore, 3.4 can be used to update the old Jacobian in only one direction.

The key to improving the algorithm, then, would be strategically picking the right directions in which to update the Jacobian. We may also consider choosing to update in several directions in one step, i.e. performing a partial-rank update. We come up with seven different ways to choose that direction. In the next section, we will explain what motivated each method, and the trial data produced by each method are presented at the end up the chapter in Tables 3.1-3.16.

3.2 Update Directions

With no information or insights into the problem, a way to pick a direction in which to update would be picking a completely random direction each iteration. We implement this method and call it "Random". It updates in a random direction, and then in the next step updates in another random direction orthogonal to the direction in which it just updated. We first compare random to the existing methods: the full Jacobian update ("Full") and Broyden's method ("Broyden"). Surprisingly, Random has a better efficiency score than both Full and Broyden when it comes to hard problems. With Random implemented, our goal is to use information from the problem to achieve a lower efficiency score (a lower score corresponds to a more efficient algorithm) than random can achieve.

It was speculated in chapter 1 that some directions may contain more information than others and updating in those directions would be more essential to the success of the algorithm. The direction that we suspected would be most pivital would be the direction of the singular value decomposition with the largest singular value. A singular-value decomposition (SVD) can be performed on any non-square matrix, and it involves taking a matrix J and writing it as

$$J = U\Sigma V^T, \tag{3.9}$$

where U and V are orthogonal matrices (i.e. $V^T V = I$) and Σ is a diagonal matrix with all positive values. While determining the direction $d\theta$ to step in, the algorithm finds the inverse of $J^T J$ in 1.9. Writing $J^T J$ in terms of the singular-value decomposition gives

$$(J^T J)^{-1} = V\Sigma U^T U\Sigma V^T. \tag{3.10}$$

Since U is orthogonal, we get

$$(J^T J)^{-1} = V\Sigma^2 V^T. \tag{3.11}$$

Therefore, the square of the values in Σ are the eigenvalues of the inverse of $J^T J$. Thus, the large singular values of J correspond to greater eigenvalues for $J^T J$, which motivated us to think that the directions (contained in V in 3.9) corresponding to the larger singular values contained more critical information and could have a more profound impact on the Jacobian.

To test the effectiveness of the update method that performs an SVD and updates in the direction of the largest singular value each iteration ("SVmax"), we compare it against the three previously tested methods. Ultimately, we find SVmax to be less efficient as seen in Tables 3.15 and 3.16. We believe this is because SVmax updates in a similar direction each iteration, leaving several directions unchanged for many steps. The Jacobian thus becomes an increasingly poor approximation in those directions.

We decide to attempt three other methods, all utilizing an SVD. One we call "SVmaxcycle", which performs an SVD and then updates in the direction corresponding to the largest singular value in the first iteration, then the second largest singular value in the second iteration and so on until the Jacobian has been updated in every direction contained in the SVD. It then recalculates the SVD and repeats the updating process. Another method we implement is called "SVmaxp5", which calculates an SVD every step and updates in the half the directions possible, corresponding to the larger half of the singular values. For example, if there were 10 parameters, SVmaxp5 would update in the 5 directions corresponding to the 5 largest singular values. Lastly, we use a method that is a mix between the previous two methods called "SVmaxp5cycle" that updates in half the directions with larger singular values in one step and then the rest of the directions in the next step before recalculating the SVD.

3.3 Results and Conclusions

Tables 3.1-3.14 present the data from trial runs on the seven test problems, each with two difficulty levels. The tables display both the success rate and the efficiency relative to the full Jacobian update, then rank each method. Also included are two tables 3.15-3.16 with the cumulative results

Table 3.1 Results from the easy version of AERproj. The relative efficiency and relative success rate are in relation to the full Jacobian update. The methods were then ranked.

AERproj Easy

Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.9259	7	0.7892	3
Full	1.0000	6	1.0000	7
Random	1.0211	3	0.8194	6
Svmax	1.0793	1	0.7825	1
Svmaxcycle	1.0211	3	0.7912	4
Svmaxp5	1.0793	1	0.7825	1
Svmaxp5cycle	1.0211	3	0.7912	4

Table 3.2 Results from the hard version of AERproj.

AERproj Hard

Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	1.0270	4	0.8505	1
Full	1.0000	7	1.0000	7
Random	1.0540	3	0.8557	2
Svmax	1.0810	1	0.8684	3
Svmaxcycle	1.0270	4	0.8996	5
Svmaxp5	1.0810	1	0.8684	3
Svmaxp5cycle	1.0270	4	0.8996	5

Table 3.3 Results from the easy version of ATRproj.

ATRproj Easy				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.7331	7	0.5644	1
Full	1.0000	1	1.0000	7
Random	0.9831	6	0.6388	2
Svmax	1.0000	1	0.7223	5
Svmaxcycle	0.9873	4	0.6997	3
Svmaxp5	1.0000	1	0.7223	5
Svmaxp5cycle	0.9873	4	0.6997	3

Table 3.4 Results from the hard version of ATRproj.

ATRproj Hard				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.6271	7	0.9023	4
Full	1.0000	2	1.0000	7
Random	1.1356	1	0.7270	1
Svmax	0.9153	5	0.9822	5
Svmaxcycle	0.9492	3	0.8901	2
Svmaxp5	0.9153	5	0.9822	5
Svmaxp5cycle	0.9492	3	0.8901	2

Table 3.5 Results from the easy version of DANN3.

DANN3 Easy				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.9394	5	0.8414	5
Full	1.0000	1	1.0000	6
Random	0.9798	2	0.6743	1
Svmax	0.9798	2	0.6748	2
Svmaxcycle	0.8889	7	0.7474	3
Svmaxp5	0.9798	2	1.0165	7
Svmaxp5cycle	0.9091	6	0.7999	4

Table 3.6 Results from the hard version of DANN3.

DANN3 Hard				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.7778	5	1.0173	5
Full	1.0000	2	1.0000	4
Random	1.0476	1	0.6062	2
Svmax	0.7778	5	1.0933	7
Svmaxcycle	0.9048	4	0.6199	3
Svmaxp5	0.9206	3	1.0285	6
Svmaxp5cycle	0.6349	7	0.5650	1

Table 3.7 Results from the easy version of EDF.

EDF Easy				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.8037	6	0.5338	1
Full	1.0000	2	1.0000	6
Random	0.9018	4	0.7234	4
Svmax	0.7607	7	0.9255	5
Svmaxcycle	0.9571	3	0.5460	2
Svmaxp5	0.8650	5	1.0140	7
Svmaxp5cycle	1.0123	1	0.6788	3

Table 3.8 Results from the hard version of EDF.

EDF Hard				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.1646	7	1.9686	7
Full	1.0000	2	1.0000	4
Random	0.8101	4	0.3127	1
Svmax	0.4557	6	1.3490	6
Svmaxcycle	1.0759	1	0.3642	2
Svmaxp5	0.6582	5	1.1921	5
Svmaxp5cycle	0.9620	3	0.6795	3

Table 3.9 Results from the easy version of EDFproj.

EDFproj Easy				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	1.0000	1	0.285054348	1
Full	1.0000	1	1	7
Random	1.0000	1	0.731929348	4
Svmax	1.0000	1	0.89048913	5
Svmaxcycle	0.992	6	0.724957269	2
Svmaxp5	1.0000	1	0.89048913	5
Svmaxp5cycle	0.992	6	0.724957269	2

Table 3.10 Results from the hard version of EDFproj.

EDFproj Hard				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	1.0000	1	0.2619	1
Full	1.0000	1	1.0000	7
Random	1.0000	1	0.7260	4
Svmax	1.0000	1	0.8556	5
Svmaxcycle	0.9920	6	0.7118	2
Svmaxp5	1.0000	1	0.8556	5
Svmaxp5cycle	0.9920	6	0.7118	2

Table 3.11 Results from the easy version of GDFproj.

GDFprojA				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.876	6	1.1867	6
Full	1.000	1	1.0000	4
Random	1.000	1	0.9606	3
Svmax	0.488	7	6.3046	7
Svmaxcycle	0.972	4	0.6983	1
Svmaxp5	1.000	1	1.1817	5
Svmaxp5cycle	0.972	4	0.7850	2

Table 3.12 Results from the hard version of GDFproj.

GDFprojB				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.538	7	0.787	5
Full	1.000	2	1.000	7
Random	0.977	4	0.453	2
Svmax	0.632	6	0.851	6
Svmaxcycle	0.988	3	0.335	1
Svmaxp5	0.912	5	0.778	4
Svmaxp5cycle	1.006	1	0.539	3

Table 3.13 Results from the easy version of IAD.

IAD Easy				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	1.0000	1	0.4587	1
Full	1.0000	1	1.0000	7
Random	1.0000	1	0.6091	3
Svmax	1.0000	1	0.6306	4
Svmaxcycle	0.9760	6	0.5886	2
Svmaxp5	1.0000	1	0.6996	5
Svmaxp5cycle	0.9760	6	0.7044	6

Table 3.14 Results from the hard version of IAD.

IAD Hard				
Method	Rel. Succ. Rate	Succ. Rank	Rel. Eff.	Eff. Rank
Broyden	0.8300	7	1.4850	7
Full	1.0000	6	1.0000	6
Random	1.0900	4	0.9637	5
Svmax	1.1800	1	0.9060	4
Svmaxcycle	1.1200	3	0.6394	1
Svmaxp5	1.1300	2	0.7483	3
Svmaxp5cycle	1.0400	5	0.7128	2

Table 3.15 Summary of overall results. Average were taken from the 7 easy problems for average relative efficiency as well as average relative rank. Those averages were then ranked.

Easy Problems				
Method	Avg. Rel. Eff.	Rank	Avg. Eff. Rank	Rank
Broyden	0.67	1	2.57	2
Full	1.00	6	6.29	7
Random	0.74	3	3.29	3
Svmax	1.56	7	4.14	5
Svmaxcycle	0.69	2	2.43	1
Svmaxp5	0.90	5	5.00	6
Svmaxp5cycle	0.74	4	3.43	4

Table 3.16 Summary of overall results from the 7 hard problems.

Hard Problems				
Method	Avg. Rel. Eff.	Rank	Avg. Eff. Rank	Rank
Broyden	1.04	7	4.29	4
Full	1.00	6	6.00	7
Random	0.66	2	2.43	2
Svmax	0.99	5	5.14	6
Svmaxcycle	0.64	1	2.29	1
Svmaxp5	0.92	4	4.43	5
Svmaxp5cycle	0.71	3	2.57	3

from the trial problems. They give the average relative efficiency score relative to a full Jacobian update as well as the average efficiency rank, calculated by averaging each method's average rank in relative efficiency over the seven problems.

Table 3.16 shows that the most efficient algorithm is SVmaxcycle. We believe this is the most efficient algorithm because the directions with higher singular values do, indeed, contain more vital information about the Jacobian. At the same time, the method keeps the Jacobian well updated by eventually updating in all directions, instead of SVmax which only updates in the direction with the largest singular values every time.

With these results on test problems, we are confident that SVmaxcycle is a partial-rank Jacobian update method that can be used on real life problems. With models getting continually bigger, SVmaxcycle can be used to lower the computational cost with minimal loss of accuracy.

Appendix A

Appendix Title

A.1 testlm.py

This file is the main python script that would test different update methods on the various test problems.

```
import testing

reload(testing)

test = testing.test

organizedata = testing.organizedata

suborganizedata = testing.suborganizedata

getdetailedstats = testing.getdetailedstats

getstatssummary = testing.getstatssummary


import numpy as np


import models.toy.DANN3 as model
```

```
cmin=2.8467

alpha=2.1

lamda=.0001

modelname="DANN3"



#import models.minpack2mkt.IAD as model

#cmin=.0002484

#lamda=.0001

#alpha=3.485

#modelname="IAD"



# import models.minpack2mkt.ATRproj as model

# cmin=43.96

# lamda=5000000

# alpha=4.411

# modelname="ATRproj"


# import models.minpack2mkt.GDFproj as model

# cmin=.02006

# lamda=.0001

# alpha=3.052

# modelname="GDFproj"
# import models.minpack2mkt.AERproj as model

# cmin=1.53

# lamda=.0001

# alpha=5.12

# modelname="AERproj"


# import models.minpack2mkt.EDF as model

# cmin=2.73244e-5

# lamda=.00001

# alpha=3.558

# modelname="EDF"


# import models.minpack2mkt.EDFproj as model

# cmin=.0027432

# lamda=.00001

# alpha=3.558

# modelname="EDFproj"


difficulty=1.0

maxstop=250


np.random.seed(0)

xis=model.xi + difficulty*np.random.randn(50,model.N)

N=len(xis[0])
```

```
import Updaterlmfull

Updaterfull=Updaterlmfull.Updaterlmfull(model.N)


import Updaterlmbroyden

Updaterbroyden=Updaterlmbroyden.Updaterlmbroyden(model.N)


import Updaterlmrandom

Updaterrandom=Updaterlmrandom.Updaterlmrandom(model.N)


import Updaterlmsvmax

Updatersvmax=Updaterlmsvmax.Updaterlmsvmax(model.N)


Updaters = {

    "full": Updaterfull,

    "broyden": Updaterbroyden,

    "random": Updaterrandom,

    "svmax": Updatersvmax,

}


f=open("Results.csv","a")


for Updater in Updaters.keys():

    print "Running %s" %Updater

    infos=test(model,xis,Updaters[Updater],lamda,maxstop,alpha,cmin)
    objectivesuccesscosts=organizedata(infos,cmin,alpha,1)

    suborganizedata(infos,cmin,alpha,1)

    detailedstats=getdetailedstats(infos,N)

    statssummary,successrate,efficiencyscore=getstatssummary(detailedstats,objectivesucc

    line="%s,%f,%i,%s,%f,%f \n" %(modelname,difficulty,maxstop,Updater,successrate,effic

    print line

    f.write(line)



f.close()
```

A.2 levmarclass.py

This file is the python script containing the Levenberg-Marquardt algorithm

```
import numpy as np

from numpy.linalg import inv


class levmar:


    def __init__(self,Updater,maxstop=50,mincost=0,mindx=-0.00000001,maxlam=10000000,fto

        self.maxstop=maxstop

        self.mincost=mincost

        self.mindx=mindx

        self.maxlam=maxlam
        self.ftol=ftol

        self.Updater=Updater


    def levmar(self,r,v,x,args=(),lamda=.001):

        self.N = len(x)

        self.lamda=lamda

        self.step=0

        self.r=r

        self.v=v

        self.x=x

        self.args=args

        self.functioncounter=0

        self.jaccounter=0.0

        self.f=r(x,*args)

        self.functioncounter+=1

        self.jaccounter+=1.0/self.N

        self.J=np.array([v(x,vj,*args) for vj in np.eye(len(x))]).T

        self.Updater.reset(self) #Need to reset to load svd into updater.

        self.jaccounter+=1

        self.fold=self.f.copy()

        self.dx=x*np.inf

        self.msg=""

        self.ftolcounter=0

        while not self.checkstop():

            self.step=self.step+1
            #if printlevel==1:
            #    print step,lamda,sum(f**2)
            self.proposestep()
            self.fold=self.f.copy()
            #Update Jacobian
            self.Updater.update(self)

            #Accept or reject
            if  sum(self.fnew**2)<sum(self.f**2):
                self.x=self.x+self.dx
                self.f=self.fnew
                self.lamda=self.lamda/2
                self.Updater.reset(self)
            else:
                self.lamda=3*self.lamda

    def proposestep(self):
        self.dx=np.linalg.solve(np.dot(self.J.T,self.J)+self.lamda*np.eye(len(self.x)),-
        self.fnew=self.r(self.x+self.dx,*self.args)
        self.functioncounter+=1

    def checkstop(self):
        if self.step>=self.maxstop:
            self.msg="maxstop"
            return True
        if sum(self.f**2)/2<self.mincost:
            self.msg="mincost"
            return True
        if (sum(self.f**2)-sum(self.fold**2))!=0 and abs(sum(self.f**2)-sum(self.fold**2
            if self.ftolcounter>3:
                self.msg="ftol"
                return True
            else:
                self.ftolcounter+=1
        else:
            self.ftolcounter=0
        if self.lamda>self.maxlam:
            self.msg="maxlam"
            return True
        if np.linalg.norm(self.dx)<self.mindx:
            self.msg="mindx"
            return True
        else:
            return False
```

A.3 Updaterlm.py

This python file contains the class from which all updater classes inherit.

```
import numpy as np
```

```
class Updater(object):

    def __init__(self,N):
        self.N=N
        self.nupdated=0

    def update(self,lmclass):
        pass

    def reset(self,lmclass):
        self.nupdated=0
```

A.4 Updaterlmfull.py

This python file contains the class used to perform the full Jacobian update during the Levenberg-Marquardt algorithm.

```
import numpy as np
from Updaterlm import Updater

class Updaterlmfull(Updater):
    def __init__(self,N):
        Updater.__init__(self,N)
    def update(self,lmclass):
        if self.nupdated==0:
            lmclass.J=np.array([lmclass.v(lmclass.x,vi,*lmclass.args) for vi in np.eye(l
            lmclass.jaccounter+=1
            self.nupdated=self.N
```

A.5 Updaterlmbroyden.py

This python file contains the class used to perform the Broyden update during the Levenberg-Marquardt algorithm.

```
import numpy as np
from Updaterlm import Updater


class Updaterlmbroyden(Updater):
    def __init__(self,N):
        Updater.__init__(self,N)

    def update(self,lmclass):
        if self.nupdated<self.N:
            lmclass.J=lmclass.J+np.outer(lmclass.fnew-lmclass.f-np.dot(lmclass.J,lmclass
            lmclass.jaccounter+=1.0/self.N
            self.nupdated+=1
```

A.6 Updaterlmrandom.py

This python file contains the class used to perform the rank-1 update in a random direction during the Levenberg-Marquardt algorithm.

```
import numpy as np
from Updaterlm import Updater
from UpdateJ import UpdateJ
import sys


class Updaterlmrandom(Updater):
    def __init__(self, N):
        Updater.__init__(self,N)
        self.Updated = np.zeros((N,N))



    def reset(self,lmclass):
        self.nupdated = 0
        self.Updated *= 0.0

    def ProjVOrthogonal(self, v):
        vnew = v.copy()
        vnew -= np.dot(np.dot( self.Updated, self.Updated.T), vnew)
        return vnew/np.linalg.norm(vnew)

    def GetNextV(self):
        """
        This should be overloaded

        Default is a random v

        """

        v = np.random.randn(self.N)

        return v/np.linalg.norm(v)


    def update(self,lmclass):

        if self.nupdated < self.N:

            v = self.ProjVOrthogonal( self.GetNextV() )

            self.Updated[:,self.nupdated] = v[:]

            self.nupdated += 1

            lmclass.jaccounter+=1.0/self.N

            lmclass.J = UpdateJ(lmclass.J, v, lmclass.v(lmclass.x,v,*lmclass.args) )
```

A.7 Updaterlmsvmax.py

This python file contains the class used to perform the rank-1 update using the svmax method.

```
import numpy as np

from Updaterlmrandom import Updaterlmrandom


class  Updaterlmsvmax(Updaterlmrandom):

    def __init__(self, N):

        Updaterlmrandom.__init__(self,N)


    def reset(self, lmclass):
        Updaterlmrandom.reset(self,lmclass)
        u,s,vh = np.linalg.svd(lmclass.J)
        self.vh = vh



    def GetNextV(self):
        return self.vh[self.nupdated]/np.linalg.norm(self.vh[self.nupdated])
```

A.8 Updaterlmsvmaxcycle.py

This python file contains the class used to perform the rank-1 update using the svmaxcycle method.

```
import numpy as np
from Updaterlmrandom import Updaterlmrandom


class  Updaterlmsvmaxcycle(Updaterlmrandom):
    def __init__(self, N):
        self.i = 0
        Updaterlmrandom.__init__(self,N)

    def reset(self, lmclass):
        Updaterlmrandom.reset(self,lmclass)
        self.J = lmclass.J
        # u,s,vh = np.linalg.svd(lmclass.J)
        # self.vh = vh



    def GetNextV(self):
        if self.i == self.N:
            u,s,vh = np.linalg.svd(self.J)
            self.vh = vh
            self.i = 1
            return vh[0]
        else:
            self.i += 1
            return self.vh[self.i - 1]
```

A.9 Updaterlmsvmaxfrac.py

This python file contains the class used to perform the partial-rank update using the svmaxp5 method.

```
import numpy as np
from Updaterlmrandom import Updaterlmrandom


class  Updaterlmsvmaxfrac(Updaterlmrandom):
    def __init__(self, N, frac):
        self.frac = frac
        Updaterlmrandom.__init__(self,N)



    def reset(self, lmclass):
        Updaterlmrandom.reset(self,lmclass)
        u,s,vh = np.linalg.svd(lmclass.J)
        self.vh = vh

    def GetNextV(self):
        return self.vh[self.nupdated]/np.linalg.norm(self.vh[self.nupdated])

    def update(self,lmclass):
        for i in range(int(self.N*self.frac)):
            Updaterlmrandom.update(self, lmclass)
```

A.10 Updaterlmsvmaxfraccycle.py

This python file contains the class used to perform the partial-rank update using the svmaxp5cycle method.

```
import numpy as np
from Updaterlmrandom import Updaterlmrandom

class  Updaterlmsvmaxfraccycle(Updaterlmrandom):
    def __init__(self, N, frac):

        self.frac = frac

        self.i = 0

        Updaterlmrandom.__init__(self,N)


    def reset(self, lmclass):

        Updaterlmrandom.reset(self,lmclass)

        self.J = lmclass.J

        # u,s,vh = np.linalg.svd(lmclass.J)

        # self.vh = vh



    def GetNextV(self):

        if self.i == self.N:

            u,s,vh = np.linalg.svd(self.J)

            self.vh = vh

            self.i = 1

            return vh[0]

        else:

            self.i += 1

            return self.vh[self.i - 1]



    def update(self,lmclass):

        for i in range(int(self.N*self.frac)):
```

```
        Updaterlmrandom.update(self, lmclass)
```

Bibliography

Averick, B. M., Carter, R. G., & Moré, J. J. 1991, The MINPACK-2 test problem collection (preliminary version), Tech. rep., Argonne National Lab., IL (USA). Mathematics and Computer Science Div.

Bates, D. M. W., Bates, D. G. D. M., & Watts, D. G. 1988, Nonlinear regression analysis and lts applications No. 519.536 B3

Dennis, J. 1971, Mathematics of Computation, 25, 559

Gay, D. M., & Schnabel, R. B. 1978, Nonlinear Programming, 3, 245

Hartley, H. O. 1961, Technometrics, 3, 269

Moré, J. J. 1978, in Numerical analysis (Springer), 105–116

Soman, S. S., Zareipour, H., Malik, O., & Mandal, P. 2010, in North American Power Symposium (NAPS), 2010, IEEE, 1–8

Transtrum, M. K., & Sethna, J. P. 2012, arXiv preprint arXiv:1201.5885