Incorporating Function Values into Quasi-Newton Updates

Samuel R. Buss

June 17, 2006

Abstract

The traditional quasi-Newton method for updating the approximate Hessian is based on the change in the gradient of the objective function. This paper describes a new update method that incorporates also the change in the value of the function. The method effectively uses a cubic approximation of the objective function to better approximate its directional second derivative. The cubic approximation is adjusted when necessary to ensure the positive definiteness of the approximating Hessian matrix.

For the BFGS method this results in an average improvement of about 5% in performance. This improvement is modest, but can be obtained with a correspondingly modest change to the BFGS Hessian update algorithm. The best improvement is obtained with a line search algorithm that uses modified Wolfe conditions; however, in practice, the traditional line search algorithms can be used almost as effectively.

1 Introduction

Quasi-Newton algorithms, such as the DFP and BFGS algorithms, are widely used for minimizing functions, with the BFGS algorithm generally acknowledged to be the best and most widely implemented quasi-Newton method. These algorithms are iterative and work by keeping an approximation to the (inverse) Hessian matrix of the function $f$ that is used to determine the direction for a line search. After each line search, gradient information is used to update the approximation to the (inverse) Hessian matrix.

*Supported in part by NSF grant DMS-0400848. Department of Mathematics, University of California, San Diego, La Jolla, CA 92093-0112. sbuss@math.ucsd.edu
Strictly speaking, quasi-Newton algorithms find only a stationary point, not necessarily a local minimum; however, they can be quite useful for finding local minima, for instance if combined with a restart at a small random displacement from any stationary point to determine if the stationary point is actually a local minimum.

A key step in the quasi-Newton algorithms is maintaining an approximation to the Hessian or the inverse Hessian matrix of the objective function. In the present paper, we present a new method for updating this approximation to the Hessian matrix. Let \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) be the function to be minimized, let \( x_i \in \mathbb{R}^n \) denote the point found after the \( i \)th line search, and let \( g_i \) be the gradient \( g_i = \nabla f(x_i) \) at that point. The traditional quasi-Newton methods use the change in function’s gradient \( g_{i+1} - g_i \) and the change \( \delta_i = x_{i+1} - x_i \) in input point to calculate a new approximation to the (inverse) Hessian. Our new method, as described later in the paper, also takes into account the change in function value \( f(x_{i+1}) - f(x_i) \). The intuition is that this extra information can give a better estimate to the Hessian function. In particular, the new method uses a cubic approximation of the objective function to estimate the second derivative of the objective function in the search direction. However, the cubic approximation must be modified so as to maintain the positive definiteness of the approximate Hessian.

Our numerical experiments show that the new update method does indeed improve the performance of the BFGS method, by a modest, but measurable amount. Namely, the computational cost of the BFGS algorithm is improved by approximately 5%. An analysis using Dolan-Moré style performance profiles [3] confirms the robustness of this improvement. This improvement is achieved by only a very minor change to the BFGS update algorithm, and thus is easily incorporated into any BFGS algorithm. The new update method did not reliably yield any improvement to the DFP method in our experiments.

The outline of the paper is as follows. First, section 2 begins with an overview of the BFGS and DFP algorithms and of line searches with the Wolfe conditions. It then gives the mathematical description of the traditional method for updating the approximation to the inverse Hessian, and describes the mathematical basis for the cubic approximation method for updating the inverse Hessian. We also discuss the conditions that should be satisfied by a line search algorithm. For the new cubic approximation algorithm, it is necessary that the line search satisfy not only the usual Wolfe stopping conditions but also a new “\( \beta \)-condition”. The standard line search algorithm is easily modified to incorporate the \( \beta \)-condition. Or, if one is not able to change the line search algorithm, our experience has been that a line
search algorithm that enforces only the Wolfe conditions can still be used since the \( \beta \)-condition is nearly always satisfied anyway. Section 2 concludes with pseudo-code for implementing the new update method.

Section 3 describes the results of numerical experiments on 196 optimization problems from the CUTE problem set [1]. This includes comparisons of the DFP and BFGS methods with and without the new update method. The experiments indicate that the BFGS method is consistently improved by the new update method. However, the DFP algorithm is not consistently improved.

Section 4 includes proofs of the success of a line search algorithm based on the Wolfe conditions and the \( \beta \)-condition.

2 Mathematical development

2.1 Review of BFGS, DFP, and line search conditions

It is assumed the reader is familiar with the BFGS and DFP algorithms and line searches. However, we give a brief overview of them in order to establish notation; see, for instance, [4, 11] for more complete discussions.

Given an objective function \( f : \mathbb{R}^n \to \mathbb{R} \) with continuous second derivative, we wish to find a local minimum of \( f \). We assume it is feasible to compute the value of \( f(x) \) and its gradient \( g(x) \) at any point \( x \in \mathbb{R}^n \). The quasi-Newton methods iteratively find points \( x_0, x_1, x_2, \ldots \), and maintain an \( n \times n \) matrix \( H_i \) that is intended to approximate the inverse of the Hessian matrix \( G_i \) at the current point \( x_i \).\(^1\) Letting \( g_i \) be the gradient \( g_i = g(x_i) \) a search direction is chosen as \( s_i = -H_i g_i \). A line search then finds a value \( \alpha_i > 0 \) that serves as an approximate local minimizer of \( f(x_i + \alpha_i s_i) \), or more strictly speaking, finds a value \( \alpha_i \) so that the value of \( f \) is sufficiently decreased at \( x_i + \alpha_i s_i \). The next point is defined to be \( x_{i+1} = x_i + \alpha_i s_i \). In addition, the matrix \( H_{i+1} \) is obtained by updating \( H_i \) using the results of the line search.

It is important for quasi-Newton methods that the matrix \( H_i \) is positive definite, and this imposes conditions on the line search. The first traditional condition, the Armijo condition, states that

\[
f(x_{i+1}) \leq f(x_i) + \alpha_i g_i^T s_i.
\]  

The second traditional condition is the curvature condition, which states

\[
f(x_{i+1}) \leq f(x_i) + \alpha_i g_i^T s_i.
\]

\(^1\)Often the Hessian matrix is maintained in its Cholesky decomposition form \( G_i = L_i L_i^T \) where \( L_i \) is lower triangular. This is discussed later in the paper.
that
\[ g_{i+1}^T s_i \geq \sigma g_i^T s_i. \] (2)

Here \( \rho \) and \( \sigma \) are constants with \( 0 < \rho < \sigma < 1 \). Together, conditions (1) and (2) are called the Wolfe conditions. Sometimes a strong curvature condition,
\[ |g_{i+1}^T s_i| \leq -\sigma g_i^T s_i, \] (3)
is used in place of the curvature condition. Conditions (1) and (3) are called the strong Wolfe conditions.

We shall add a third stopping condition to the line search called the “\( \beta \)-condition”, namely,
\[ \alpha_i g_{i+1}^T s_i > f(x_{i+1}) - f(x_i). \] (4)

The rationale for the \( \beta \)-condition will be clearer after the definition of the revised method of updating the inverse Hessian \( H_i \). Note that the \( \beta \)-condition is related to the curvature condition in that both conditions put a lower bound on the slope \( g_{i+1}^T s_i \).

After each line search, the matrix \( H_{i+1} \) is computed by updating \( H_i \). The DFP and BFGS algorithms use different updates, however both base their updates on the values \( \delta = x_{i+1} - x_i \) and \( \gamma = g_{i+1} - g_i \). For the BFGS method, the update is performed as
\[ H_{i+1} = H_i + \left(1 + \frac{\gamma^T H_i \gamma}{\delta^T \gamma}\right) \frac{\delta \delta^T}{\delta^T \gamma} - \left(\frac{\delta \gamma^T H_i + H_i \gamma \delta^T}{\delta^T \gamma}\right), \] (5)
and for the DFP method, the update is
\[ H_{i+1} = H_i + \frac{\delta \delta^T}{\delta^T \gamma} - \frac{H_i \gamma \gamma^T H_i}{\gamma^T H_i \gamma}. \] (6)

Our revised update method will modify the updates (5) and (6) by replacing \( \gamma \) everywhere with \( \nu \gamma \), where \( \nu \) is a scalar that depends on the change \( f(x_{i+1}) - f(x_i) \) in the value of \( f \), as well as on \( \delta \), \( \gamma \), and \( s_i \).

### 2.2 A Cubic Approximation

We now derive a cubic approximation to the objective function that will be used to estimate the second derivative of \( f(x_{i+1}) \) in the search direction. From that, we will define the scalar \( \nu \).
Figure 1: The value $\beta$ is defined in terms of the intersection of the tangent lines to $h(t)$ at $t = 0$ and $t = \alpha$. In this figure, $\beta$ is approximately equal to 0.3.

Define $h(t) = f(x_i + ts_i)$. The line search can viewed as searching for a value $\alpha > 0$ that $t = \alpha$ is approximately a local minimizer of $h$. The value $x_{i+1}$ is then set equal to $x_i + \alpha s_i$. The first derivative of $h$ is

$$h'(t) = g(x_i + ts_i)^T s_i.$$  

The two Wolfe conditions and the $\beta$-conditions restrict the permissible values of $\alpha$; they can be reexpressed in terms of $h$ as:

$$h(\alpha) \leq h(0) + \alpha \rho h'(0)$$  

$$h'(\alpha) \geq \sigma h'(0)$$  

$$\alpha h'(\alpha) > h(\alpha) - h(0).$$

The traditional BFGS and DFP updates, equations (5) and (6), use $\gamma = g_{i+1} - g_i$ and are in effect estimating $h''(\alpha)$ using a secant approximation

$$h''(\alpha) \approx \frac{h'(\alpha) - h'(0)}{\alpha - 0}.$$  

We shall refine this by using a cubic approximation of $h(\alpha)$. Let $k(t)$ be a cubic polynomial that matches the values and derivatives of $h$ at $t = 0$ and $t = \alpha$. That is, $k(0) = h(0)$, $k(\alpha) = h(\alpha)$, $k'(0) = h'(0)$, and $k'(\alpha) = h'(\alpha)$. To give some geometric intuition for the calculation of $k''(\alpha)$, we define

$$\beta = \frac{\alpha h'(\alpha) - (h(\alpha) - h(0))}{\alpha h'(\alpha) - ah'(0)}.$$
The value $\beta$ has a simple geometric interpretation, as shown in Figure 1. Let $L_0$ and $L_\alpha$ be the lines tangent to the graph of $f$ at $t = 0$ and $t = \alpha$, respectively. Then, it is easy to check that $\beta$ is the fraction of the distance from 0 to $\alpha$ where the two lines intersect; that is to say, $L_0$ and $L_\alpha$ intersect at $t = \beta \alpha$.

We must have $h'(0) < 0$, since $H_i$ is positive definite and the search direction $s_i$ is a descent direction. Thus, by the curvature condition (8), $h'(\alpha) - h'(0) > 0$. Therefore, if the $\beta$-condition (9) also holds, it must be that $\beta > 0$.

With some straightforward calculations, it can be shown that the second derivative of $k$ at $t = \alpha$ can be expressed in terms of $\beta$ as follows:

$$k''(\alpha) = (6\beta - 2) \cdot \frac{h'(\alpha) - h'(0)}{\alpha - 0}.$$ 

In light of this, our improved approximation for $h''(\alpha)$ is

$$(6\beta - 2) \cdot \frac{h'(\alpha) - h'(0)}{\alpha - 0} \approx h''(\alpha). \tag{12}$$

Note that approximation for $h''$ differs from that of equation (10) only by the factor $6\beta - 2$.

The following theorem formalizes the derivation of (12) as an approximation to $h''(\alpha)$.

**Theorem 1** Suppose $h(\alpha)$ has a convergent Taylor series. Then

$$h''(\alpha) = (6\beta - 2) \cdot \frac{h'(\alpha) - h'(0)}{\alpha - 0} + O(\alpha^2). \tag{13}$$

**Proof** The Taylor series gives

$$h(\alpha) = h(0) + \alpha h'(0) + \frac{\alpha^2}{2} h''(0) + \frac{\alpha^3}{6} h'''(0) + O(\alpha^4)$$

$$h'(\alpha) = h'(0) + \alpha h''(0) + \frac{\alpha^2}{2} h'''(0) + O(\alpha^3)$$

$$h''(\alpha) = h''(0) + \alpha h'''(0) + O(\alpha^2).$$

From these, we have

$$\beta = \frac{\alpha h'''(0) + \frac{\alpha^2}{2} h'''(0) + O(\alpha^3)}{h'(\alpha) - h'(0)}$$

$$= \frac{\alpha h''(0) + \frac{\alpha^2}{2} h'''(0) + O(\alpha^3)}{h'(\alpha) - h'(0)}$$

$$= \frac{\alpha h'(\alpha) - h'(0)}{h'(\alpha) - h'(0)} = \frac{\alpha h'(\alpha)}{h'(\alpha) - h'(0)}.$$
and then
\[(6\beta - 2)\frac{h'(\alpha) - h'(0)}{\alpha} = h''(0) + \alpha h'''(0) + O(\alpha^2)\]
\[= h''(\alpha) + O(\alpha^2),\]
and the theorem is proved. \(\square\)

Returning to the minimization of the function \(f(x)\) over \(\mathbb{R}^n\), the approximation (12) would suggest setting \(\nu\) equal to \(6\beta - 2\). However, although \(\beta > 0\) is ensured by the \(\beta\)-condition, there is no assurance that \(\beta > 1/3\), and hence no guarantee that \(6\beta - 2 > 0\). This is a problem, since it is necessary that \(\nu\) be positive in order to maintain the positive definiteness of the matrix \(H_{i+1}\) — the positive definiteness of this matrix is crucial for the DFP and BFGS methods. Accordingly, we use an ad-hoc method to keep \(\nu\) positive and define \(\nu\) as

\[\nu = \begin{cases} 6\beta - 2 & \text{if } \beta \geq \frac{1}{2} \\ (2\beta)^3 & \text{otherwise.} \end{cases}\]

This makes \(\nu = \nu(\beta)\) a smooth function that remains positive for \(0 < \beta < 1/2\).

In our implementation, to avoid underflow problems, we check whether \(\beta < 0.001\) before computing \(\nu\). If so, the update is skipped entirely and \(H_{i+1}\) is set equal to \(H_i\).

### 2.3 Implementation details

Based on the mathematical development above, we can summarize the revised algorithm for the BFGS or DFP algorithms as follows. We henceforth call these revised algorithms the “\(\beta\)-BFGS” and “\(\beta\)-DFP” algorithms.

<table>
<thead>
<tr>
<th>Inputs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective function (f).</td>
</tr>
<tr>
<td>Initial point (x_0) and initial gradient (g_0).</td>
</tr>
<tr>
<td>Initial matrix (H_0) (often (H_0) is the identity).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop with (i = 0, 1, 2, \ldots)</td>
</tr>
<tr>
<td>Set the search direction (s_i = -H_ig_i).</td>
</tr>
<tr>
<td>Invoke line search to find (\alpha_i &gt; 0) satisfying</td>
</tr>
<tr>
<td>the Wolfe conditions and the (\beta)-condition.</td>
</tr>
</tbody>
</table>
Set $x_{i+1} = f(x_i + \alpha s_i)$.

Compute the gradient $g_{i+1}$.

Set $\beta = \frac{\alpha_i g_{i+1}^T s_i - (f(x_{i+1}) - f(x_i))}{\alpha_i g_i^T s_i - \alpha_i g_i^T s_i}$.

If $\beta > 0.001$ {
    Set $\nu = \begin{cases} 
        6\beta - 2 & \text{if } \beta \geq \frac{1}{2} \\
        (2\beta)^3 & \text{otherwise}
    \end{cases}$
    Update $H_{i+1}$ from $H_i$ using equation (5) or (6)
    using $\delta = x_{i+1} - x_i$ and $\gamma = \nu \cdot (g_{i+1} - g_i)$.
}

else {
    Set $H_{i+1} = H_i$.
}

If not sufficiently converged, continue looping.

A line search algorithm that satisfies the Wolfe conditions and the $\beta$-condition will be discussed in section 4. It is standard for line searches to satisfy the Wolfe conditions since this is required already for the traditional quasi-Newton algorithms. Our experience has been that in the overwhelming majority of cases, these line searches return a point that also satisfies the $\beta$-condition. For this reason, the algorithm above will work well enough even if the line search does not return always satisfy the $\beta$-condition. In the cases where the $\beta$-condition fails, the test “$\beta > 0.001$” will ensure that $H_{i+1}$ is positive definite, and these cases occur so extremely rarely that they do not have a noticeable impact on the rate of convergence of the algorithms.

As usual, an actual implementation of the algorithm would reuse memory for the $H_i$ matrices, and just calculate $H_{i+1}$ by updating $H_i$. Further memory savings could be achieved by using the symmetry of $H_i$. Similarly, the algorithm saves memory by only storing the most recent two values of the points $x_i$, the values $f(x_i)$ and the gradients $g_i$.

Another popular method of maintaining $H_i$ is to instead store a lower triangular matrix $L_i$ such that $L_iL_i^T$ is the Cholesky decomposition of $G_i = H_i^{-1}$, as advocated by [7]. From $L_i$, it is easy to compute $s_i = -H_i g_i = -L_i^{-T}L_i^{-1} g_i$. A big advantage of this approach is that requiring the diagonal entries of $L_i$ to be non-zero ensures the positive definiteness of $H_i = L_i^{-T}L_i^{-1}$, even in the presence of roundoff errors. Further, there are several efficient methods for updating the matrix $L_i$ with low rank updates; see [7, 6, 8, 12]. Our own experience has been that the BFGS algorithms benefit markedly from the use of the Cholesky decomposition representation of $G_i$. On the other hand, we found that the DFP algorithms performed
slightly worse using the Cholesky decomposition for $G_i$ than they did using a direct representation of $H_i$. Accordingly, in order to focus on only the best-performing methods, the main experimental results reported in Section 3 use the Cholesky representation of $G_i$ for the BFGS and $\beta$-BFGS methods, and use a direct representation of $H_i$ for the DFP and $\beta$-DFP methods.

3 Experimental Results

This section describes experimental tests of the efficiency of BFGS and DFP algorithms that use the cubic approximation for updating the (inverse) Hessian matrix. We report in depth on one particular set of tests. In addition to these tests reported here, we ran a number of other tests with generally similar outcomes; these are discussed at the end of this section.

The experiments used 196 unconstrained optimization problems from the CUTE test suite of optimization problems [1]. The problems were selected by choosing unconstrained CUTE problems, and then subsequently discarding a few that were found experimentally to be initialized at a stationary point or to have no global minimum. The problems were converted from .mod files to .nl files using AMPL [5], and we used custom C++ routines for the BFGS and DFP algorithms both with the traditional update method and with our cubic approximation update method. All programs were run on a Pentium 4 processor running Windows XP, and were compiled under Microsoft Visual C++. All calculations were done in double precision floating point.

The DFP and $\beta$-DFP algorithms used equation (6) to maintain the matrices $H_i$ (with $\nu \gamma$ in place of $\gamma$ for the $\beta$-DFP method). The BFGS and $\beta$-BFGS algorithms maintained the Cholesky decomposition $L_i L_i^T$ for $G_i$, and updated $L_i$ to reflect rank one updates of $G_i$, using the Method C3 of Gill-Golub-Murray-Saunders [6] (see also [12]).

The algorithms all used an inexact line search very similar to the one described in [4]. The line searches used the parameters $\sigma = 9/10$ and $\rho = 1/4$, and when changing step sizes use cubic interpolating polynomials to estimate the position of a local minimum. All the line searches enforced the strong Wolfe conditions; for $\beta$-BFGS and $\beta$-DFP methods, the line searches incorporated extra tests to enforce the $\beta$-condition.

The experimental results of comparing the four methods are shown in Figures 2 through 11. Our results are reported at three different levels of accuracy, $\epsilon = 10^{-3}$, $\epsilon = 10^{-6}$, and $\epsilon = 10^{-9}$. For each minimization
problem, we ran all four methods until either they had converged extremely
close to an apparent local minimum, or until 4000 iterations were completed.
The algorithms were deemed to have converged to be “extremely close” to
a local minimum if either an iteration failed to produce an improvement in
the objective value or until the gradient had magnitude less than $10^{-7}$. For
a particular problem, let $L$ equal the best objective value found. Then,
for a fixed value of $\epsilon$, we checked whether any of the four algorithms had
converged to “$\epsilon$-close” $L$ in less than 2000 iterations. For this, we define
that $L'$ is “$\epsilon$-close” to $L$ provided that either

$$L' - L \leq (1 + \epsilon)|L| \quad \text{or} \quad 0 \leq L \leq L' \leq \epsilon.$$

If any of the four algorithms succeeded in converging $\epsilon$-close to $L$ in less than
2000 iterations, then we determined, for each of the four algorithms, (a) the
number of iterations needed to reach a value $\epsilon$-close to $L$, and (b) the total
number of function evaluations used by the line searches before reaching this
value. If the algorithm did not converge to be $\epsilon$-close to $L$ in less than 4000
iterations, then the algorithm was deemed to have failed.\footnote{The reason for using the two thresholds, 2000 and 4000, is that if one algorithm
converges in just under 4000 iterations and another in just over 4000 iterations, we do
not wish to report that the first algorithm performs substantially better than the second.}

The most relevant value of the three values for $\epsilon$ is $10^{-6}$: convergence
to within only $10^{-3}$ is too weak a condition to be terribly interesting. And,convergence to within $10^{-9}$ means that we are losing any benefit from a
cubic approximation due to lack of floating point precision. The main
results for $\epsilon = 10^{-6}$ are shown in Figures 2 through 4. Figure 2 shows a
Dolan-Moré performance profile graph for the four methods. To interpret
the performance profile graph, recall from [3] that, for each method $M$, the
performance profile shows the graph of the function

$$p_M(\tau) = \frac{1}{N} \left| \{c_{M,i} \leq \tau c_i : i < N\} \right|,$$

where $N$ is the number problems in the test set, where $c_{M,i}$ is the cost of
method $M$ on the $i$th problem, and $c_i$ is the minimum value of $c_{M,i}$ over
all methods $M$. In all our graphs other than Figure 3, the cost $c_{M,i}$ is taken
to equal the number of iterations needed by method $M$ on the $i$th problem
set. Of course, the higher the graph of $p_M$, the better the performance of
method $M$.

Figure 2 indicates that the $\beta$-BFGS method is somewhat better than the
BFGS method, and that the BFGS method is substantially better than the
DFP and β-DFP method. The DFP and β-DFP methods are essentially identical in performance; overall the β-DFP method does very slightly better than the DFP method. By observing the values of the graphs at \( \tau = 0 \), the graph in Figure 2 shows how often each of the four methods was the best, or tied for best: The β-BFGS method was best on 68.9% of the problems, the BFGS method was best on 48.8% of them, the β-DFP method was best for 14.6% of them, and the DFP method was best for 9.2% of them. These values are also reported in the second line of Table 11.

Figure 3 gives the performance profile for the same methods as Figure 2, but using the number of function evaluations as the measure of the cost of the algorithm. Because the β-BFGS and β-DFP algorithms use a different line search than the BFGS and DFP algorithms, it could have been possible that the relative costs as measured in iterations might be quite different from the relative costs as measured in number of function evaluations. However, this is not the case: Qualitatively, Figures 2 and 3 are quite similar, and the same holds true for all the data reported. For this reason, henceforth all performance graphs use the number of iterations only.

Figures 5 and 6 report the same kind of data as Figure 2, but for \( \epsilon = 10^{-3} \) and for \( \epsilon = 10^{-9} \).

The table in Figure 4 shows a summary of the relative performances of the algorithms. For an example of how to interpret the table, consider the second line, comparing β-BFGS and BFGS at \( \epsilon = 10^{-6} \). It shows that, for problems where both methods converged, the β-BFGS method required 5.4% fewer iterations and 5.0% fewer function evaluations than the traditional BFGS method. In addition, there were seven problems where the β-BFGS method converged in fewer than 2000 iterations and the BFGS method did not converge in 4000 iterations. Conversely, there were four problems where the BFGS converged in fewer than 2000 iterations but β-BFGS did not converge within 4000 iterations.

Figures 7 through 9 show the performance profiles for small, medium, and large problems, all with \( \epsilon = 10^{-6} \). “Small” problems are ones with less than 50 variables, “medium” problems have between 50 and 100 variables, and “large” problems have more than 100 variables. (No problem tested had more than 10,000 variables.) The goal here was to see if the β-BFGS method performs better for small or large problems. Here the results were a little mixed: The β-BFGS method had the biggest advantage on large problems, and almost as big an advantage on small problems, but BFGS essentially matched β-BFGS on on medium problems. Since there were only 20 medium-size problems, this last observation may be a statistical fluke. Indeed, it overall appears that the β-BFGS advantage over BFGS is
more-or-less the same regardless of the dimension of the problem.

Figure 9 indicates that the $\beta$-DFP method is advantageous over the DFP method for large dimension problems. However, according to Figures 7 and 8, this does not appear to be true for small dimension problems.

Table 10 summarizes the performance graphs of Figures 7 through 9, showing the relative performance of all four algorithms at an accuracy level of $\epsilon = 10^{-6}$ for different size problem sets. The third line of this table shows that for large problems the $\beta$-BFGS algorithm had a 8.4% advantage over BFGS in terms of iteration counts. The first line shows it has a smaller advantage on small problems. The second line indicates the advantage on medium size problems is less, but again, this may be an artifact of having only 20 medium size problems.

Table 11 shows another view of which algorithm performed the best: this table shows which of the four algorithms was the best performing algorithm on different problem sets with different accuracies. The values in the table can found from the performance profiles graphs by observing the values at $\tau = 0$. By this measure, the $\beta$-BFGS was always the best performing.

We ran a number of tests in addition to the results reported above. In particular, we tested other line searches and also tested BFGS methods that directly maintained the $H_i$ matrix and DFP algorithms that used the Cholesky decomposition of $G_i$. In every case, the results were similar to the results we report here; in fact they usually showed a bit more improvement in $\beta$-BFGS over BFGS that what was reported above. In particular, the various tests consistently found that the cubic approximation method for updating the Hessian would improve BFGS by about 5% to 9% in terms of number of iterations or number of function evaluations. We also consistently found that the cubic approximation update methods would have little or no improvement for the DFP methods. Specifically, we found that the DFP algorithms based on the Cholesky algorithm were only very slightly improved by the use of the cubic approximation for the Hessian update. On the other hand, the BFGS algorithms that used a explicit representation of the inverse Hessian matrix $H_i$ were improved by slightly more than 7% (in contrast to the 5.4% reported in the tests above). When we tested the BFGS algorithm with a line search using $\sigma = 0.3$ and $\rho = 0.075$, the cubic approximation gave an improvement of 9.3% in iterations and 7.9% in function evaluations. This last test also gave a large improvement for large tests with $\epsilon = 10^{-6}$, namely a 20.7% reduction in iterations and a 19.8% reduction in function evaluations. However, this was the only time such a large improvement was seen, so we do not expect it is reliably reproducible. The only time we ever
observed the BFGS being hurt by the use of the cubic approximation is for the medium-size problems with $\epsilon = 10^{-6}$ as reported above. Overall, the tests indicated that cubic approximation update method is a bit more useful for large dimension problems than for small dimension problems, but there were exceptions to this, such as the BFGS algorithms that maintain the matrix $H_i$ explicitly.

The $\beta$-BFGS algorithm differs from the BFGS algorithm in two ways: first, its line search enforces the $\beta$-condition and, second, it uses the cubic approximation to update the Hessian. To verify our intuition that the line search based on the $\beta$-condition is not substantially changing the performance of the BFGS algorithm by itself, we ran tests of the BFGS algorithm using the $\beta$-condition in the line search, but not using the cubic approximation. In a test of 153 problems of dimension $\leq 5000$ with $\epsilon = 10^{-6}$, only six problems showed any difference in performance due to the use of the $\beta$-condition. Over the 153 tests, there was no change in the average number of iterations, and an increase of about 0.1% in the number of function evaluations needed. From this, we conclude that it is use of the cubic approximation, not the $\beta$-condition line search, which is responsible for the improved performance of the $\beta$-BFGS algorithm.

4 Line search with the $\beta$-condition

The $\beta$-condition was defined earlier; however, it still needs to be proved that it is always possible for a line search to find a point that satisfies both the (strong) Wolfe conditions and the $\beta$-condition. The existence of such a point follows from the next theorem.

**Theorem 2** Suppose $h$ is $C^1$-continuous on $[0, \infty)$ and $h'(0) < 0$. Further suppose $h(\alpha)$ attains a global minimum value for some $\alpha > 0$. Then there exists a value $\alpha > 0$ that satisfies the strong Wolfe conditions and the $\beta$-condition.

**Proof** We must prove the existence of a value with satisfies equations (7) and (9), as well as

$$|h'(\alpha)| \leq -\sigma h'(0).$$

Since $h'$ is bounded below, there must exist a least value $\alpha_0$ where $h(\alpha_0)$ equals $h(0) + \alpha \rho h'(0)$. Clearly $h'(\alpha_0) \geq \rho h'(0)$. Since $0 < \rho \leq \sigma < 1$ and $h'(0) < 0$ and by the intermediate value theorem applied to $h'$, there must be values $\alpha \in (0, \alpha_0)$ such that $h'(\alpha)$ lies in the interval $(\sigma h'(0), \rho h'(0))$. 

13
By the choice of $\alpha_0$, any such $\alpha$ satisfies the strong Wolfe conditions and the $\beta$-condition.

Algorithms for finding a value $\alpha$ that satisfies the (strong) Wolfe condition and the $\beta$-condition are easy to design. For instance, the algorithm of Fletcher [4, §2.6] that finds points satisfying the strong Wolfe conditions can readily be adapted to find points that also satisfy the $\beta$-condition. Namely, the algorithm is modified by replacing the tests that check whether the curvature condition (2) holds by tests that check whether both (2) and (4) holds.

It is a natural question whether the $\beta$-condition can be strengthened so that value of $\beta$ is bounded away from zero. That is, is it possible to find $\beta_0 > 0$ so that $\beta > \beta_0$ can always be enforced? This would be particularly useful if $\beta_0 = 1/3$ were possible since then the definition of $\nu(\beta)$ could be simplified to equal just $6\beta - 2$. Unfortunately, this is not possible. Indeed, if $B$ is any fixed value in the interval $(0, 1)$, define

$$h(t) = t^{1/(1-B)} - t.$$

Then, it is straightforward to check that, for any $\alpha > 0$, equation (11) gives $\beta = B$. Since $B$ could be arbitrarily close to 0, it is impossible to bound $\beta$ away from zero.

5 Conclusion

Our experimental results indicate that the $\beta$-BFGS method consistently provides a modest, but measurable improvement of the BFGS method, of about 5% or more on the selected CUTE problems, and a bit better on high-dimension problems. The $\beta$-BFGS algorithm requires only very modest changes to the BFGS algorithm, with essentially no additional computational cost.

There are several directions for further experimentation and potentially further improvements. One question would be whether the limited-memory BFGS (L-BFGS) methods of [10, 9] would be improved by using the $\beta$-BFGS update method. It would be straightforward to adapt the $\beta$-BFGS method to these limited-memory methods. Our expectation is that the $\beta$-BFGS method will provide a similar improvement to limited memory algorithms as to the usual quasi-Newton algorithms; however, this has not been experimentally tested.

Another idea for improving the update of the (inverse) Hessian matrix would be to use intermediate results from the line search algorithm to
improve the estimate of the second derivative of the objective function. This would be, in essence, an extension of the idea behind our use of a cubic approximation for finding the second derivative. Namely, if the line search has to evaluate the objective function $f$ and its gradient at more than one point, then these values could be used to help estimate the second derivative of $f$. This would go against the current philosophy of having the line search act as a “black box” subroutine; rather, the intermediate results of the line search would be used to better estimate the second derivative of $f$. Of course, one would need to be careful to maintain the positive definiteness of the matrix $H_i$. We have not experimented with this idea, but one might expect this to improve BFGS methods by an amount similar to the improvement from the cubic approximation introduced in the present paper.

Finally, it would be nice to have some theoretical explanation of why our new cubic approximation update methods are more effective for BFGS than for DFP. That is to say, why the $\beta$-BFGS improves on BFGS more than $\beta$-DFP improves on DFP. One possible, purely speculative, reason is that perhaps the new update method occasionally does a very poor job of updating the (inverse) Hessian, say when $\beta$ is very large or very small, and that the self-correcting properties enjoyed by the BFGS method [2] allow the BFGS method to better overcome these poorly chosen updates.

Acknowledgements. Philip Gill helped with a number of useful conversations.

References


Figure 2: Performance profile based on number of iterations needed to converge to accuracy of $\epsilon = 10^{-6}$. Based on 164 test problems.

Legend for all figures:
Solid black line: $\beta$-BFGS. Dotted line: BFGS.
Long dashes: $\beta$-DFP. Short dashes: DFP.

Figure 3: Performance profile based on number of function evaluations needed to converge to accuracy of $\epsilon = 10^{-6}$. Based on 164 test problems.
<table>
<thead>
<tr>
<th></th>
<th>Accuracy ($\epsilon$)</th>
<th>Percent fewer iterations</th>
<th>Percent fewer function evaluations</th>
<th># of times only one converges</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta$-BFGS vs BFGS</td>
<td>$10^{-3}$</td>
<td>4.3</td>
<td>4.0</td>
<td>2 / 4</td>
</tr>
<tr>
<td></td>
<td>$10^{-6}$</td>
<td>5.4</td>
<td>5.0</td>
<td>7 / 4</td>
</tr>
<tr>
<td></td>
<td>$10^{-9}$</td>
<td>2.5</td>
<td>2.9</td>
<td>8 / 5</td>
</tr>
<tr>
<td>$\beta$-DFP vs DFP</td>
<td>$10^{-3}$</td>
<td>4.3</td>
<td>5.6</td>
<td>4 / 4</td>
</tr>
<tr>
<td></td>
<td>$10^{-6}$</td>
<td>1.7</td>
<td>1.7</td>
<td>9 / 11</td>
</tr>
<tr>
<td></td>
<td>$10^{-9}$</td>
<td>2.6</td>
<td>1.9</td>
<td>2 / 8</td>
</tr>
<tr>
<td>BFGS vs DFP</td>
<td>$10^{-3}$</td>
<td>34.0</td>
<td>5.6</td>
<td>47 / 2</td>
</tr>
<tr>
<td></td>
<td>$10^{-6}$</td>
<td>40.1</td>
<td>28.8</td>
<td>58 / 2</td>
</tr>
<tr>
<td></td>
<td>$10^{-9}$</td>
<td>40.8</td>
<td>31.5</td>
<td>56 / 1</td>
</tr>
</tbody>
</table>

Figure 4: Comparisons of average performances at three levels of accuracy.

![Figure 5: Performance profile based on number of iterations needed to converge to accuracy of $\epsilon = 10^{-3}$. Based on 173 test problems.](image-url)
Figure 6: Performance profile based on number of iterations needed to converge to accuracy of $\epsilon = 10^{-9}$. Based on 133 test problems.

Figure 7: Performance profile based on number of iterations needed to converge to accuracy of $\epsilon = 10^{-6}$. Based on 92 “small” problems with fewer than 50 variables.
Figure 8: Performance profile based on number of iterations needed to converge to accuracy of $\epsilon = 10^{-6}$, based on 20 “medium” size problems with number of variables between 50 and 100.

Figure 9: Performance profile based on number of iterations needed to converge to accuracy of $\epsilon = 10^{-6}$, based on 52 “large” problems with over 100 variables.
<table>
<thead>
<tr>
<th></th>
<th>Size of problems (ε)</th>
<th>Percent fewer iterations</th>
<th>Percent fewer function evaluations</th>
<th># of times only one converges</th>
</tr>
</thead>
<tbody>
<tr>
<td>β-BFGS vs BFGS</td>
<td>Small</td>
<td>4.2</td>
<td>3.8</td>
<td>4 / 1</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>3.3</td>
<td>1.4</td>
<td>0 / 1</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>8.4</td>
<td>8.7</td>
<td>3 / 2</td>
</tr>
<tr>
<td>β-DFP vs DFP</td>
<td>Small</td>
<td>4.2</td>
<td>0.4</td>
<td>1 / 6</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>-6.3</td>
<td>-4.6</td>
<td>1 / 2</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>9.8</td>
<td>6.8</td>
<td>7 / 3</td>
</tr>
<tr>
<td>BFGS vs DFP</td>
<td>Small</td>
<td>38.4</td>
<td>0.4</td>
<td>36 / 2</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>49.4</td>
<td>35.7</td>
<td>5 / 0</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>38.5</td>
<td>18.9</td>
<td>17 / 0</td>
</tr>
</tbody>
</table>

Figure 10: Comparisons of average performances broken down by problem dimension.

<table>
<thead>
<tr>
<th>Problem set</th>
<th>Accuracy (ε)</th>
<th>β-BFGS</th>
<th>BFGS</th>
<th>β-DFP</th>
<th>DFP</th>
</tr>
</thead>
<tbody>
<tr>
<td>All problems</td>
<td>10^-3</td>
<td>67.6%</td>
<td>51.4%</td>
<td>21.4%</td>
<td>16.2%</td>
</tr>
<tr>
<td></td>
<td>10^-6</td>
<td>68.9%</td>
<td>48.8%</td>
<td>14.6%</td>
<td>9.2%</td>
</tr>
<tr>
<td></td>
<td>10^-9</td>
<td>65.4%</td>
<td>50.4%</td>
<td>10.5%</td>
<td>5.2%</td>
</tr>
<tr>
<td>Small problems</td>
<td>10^-6</td>
<td>76.1%</td>
<td>50.5%</td>
<td>13.0%</td>
<td>13.0%</td>
</tr>
<tr>
<td>Medium problems</td>
<td>10^-6</td>
<td>60.0%</td>
<td>40.0%</td>
<td>10.0%</td>
<td>5.0%</td>
</tr>
<tr>
<td>Large problems</td>
<td>10^-6</td>
<td>55.8%</td>
<td>46.2%</td>
<td>15.4%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Figure 11: Percentage of time a given algorithm was the best performing algorithm, or tied for best performing. Percentages in each row sum to more than one because of ties.