On computing with general sparse third derivatives in unconstrained optimization

Geir Gundersen
Department of Informatics
University of Bergen
Norway
geirg@ii.uib.no

Trond Steihaug
Department of Informatics
University of Bergen
Norway
trond@ii.uib.no

Abstract

In this paper it is shown that when the sparsity structure of the problem is utilized higher order methods are competitive to second order methods (Newton), for solving large unconstrained optimization problems when the objective function is three times continuously differentiable. It is also shown how to arrange the computations of general sparse third derivatives.

1 Introduction

The use of higher order methods using exact derivatives in unconstrained optimization have not been considered practical from a computational point of view. We will show when the sparsity structure of the problem is utilized higher order methods, like the Halley method, are competitive compared to Newton's method that is a second order method. The sparsity structure of the tensor is induced by the sparsity structure of the Hessian matrix. This is utilized to make efficient algorithms for Hessian matrices with general sparsity structure. Third order methods will in general use fewer iterations than a second order method to reach the same accuracy. However, the number of arithmetic operations per iteration is higher for third order methods than second order methods. However, in [4] it is shown that for a large class of sparse problems the ratio of number of arithmetic operations of a third order method and Newton's method is constant per iteration. Further, there is an increased memory requirement. For sparse systems we show that this increase and the increase in arithmetic operations are very modest. In this paper we will extend the algorithms in [5] to general sparsity structure.

This paper was presented at the NIK-2006 conference; see http://www.nik.no/.

2 Methods for Solving Nonlinear Equations

One of the central problems of scientific computation is the efficient numerical solution of the system of n equations in n unknowns

$$F(x) = 0 \tag{1}$$

where $F : \mathbb{R}^n \to \mathbb{R}^n$ is sufficiently smooth and the Jacobian $F^{'}(x^*)$ is nonsingular.

Consider the Halley class of iterations [6] for solving (1), where $k = 0, 1, \ldots$

$$x_{k+1} = x_k - \{I + \frac{1}{2}L(x_k)[I - \alpha L(x_k)]^{-1}\}(F^{'}(x_k))^{-1}F(x_k), \tag{2}$$

and

$$L(x) = (F^{'}(x))^{-1}F^{''}(x)(F^{'}(x))^{-1}F(x).$$

This class contains the classical Chebyshev's method ($\alpha = 0$), Halley's method ($\alpha = \frac{1}{2}$), and Super Halley's method ($\alpha = 1$). More on Chebyshev's and Halleys Method [2, 7, 9, 10, 11], and Super Halley's method [6, 7]. All members in the Halley class are cubically convergent. The formulation (2) is not suitable for implementation. By rewriting the equation we get the following iterative method for $k = 0, 1, \ldots$
Solve for $s_k^{(1)}$:

$$F'(x_k)s_k^{(1)} = -F(x_k) \tag{3}$$

Solve for $s_k^{(2)}$:

$$(F'(x_k) + \alpha F^{''}(x_k)s_k^{(1)})s_k^{(2)} = -\frac{1}{2}F^{''}(x_k)s_k^{(1)}s_k^{(1)} \tag{4}$$

Update the iterate:

$$x_{k+1} = x_k + s_k^{(1)} + s_k^{(2)}$$

The first equation (3) is the Newton equation. The second equation (4) is an approximation to the Newton equation

$$F^{'}(x_k + s_k^{(1)})s_k^{(2)} = -F(x_k + s_k^{(1)})$$

since

$$\begin{aligned} F(x_k + s_k^{(1)}) &\simeq F(x_k) + F^{'}(x_k)s_k^{(1)} + \tfrac{1}{2}F^{''}(x_k)s_k^{(1)}s_k^{(1)} \\ &= \tfrac{1}{2}F^{''}(x_k)s_k^{(1)}s_k^{(1)} \end{aligned}$$

and

$$F^{'}(x_k + s_k^{(1)}) \simeq F^{'}(x_k) + \alpha F^{''}(x_k)s_k^{(1)}$$

With starting points x_0 close to the solution x^* these methods have a superior convergence compared to Newton's method.

Unconstrained Optimization

Consider the unconstrained optimization problem

$$\min_{x\in\mathbb{R}^n} f(x) \tag{5}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is sufficiently smooth.
A necessary condition at a solution x^* of (5) is that the gradient is zero

$$\nabla f(x^*) = 0.$$

Thus in (1) we have that

$$F_i = \frac{\partial f}{\partial x_i}, \;\; i = 1, \ldots, n.$$

and for the unconstrained optimization case the Hessian matrix and $F^{''}(x) = \nabla^3 f(x)$ is the second and third derivatives of f at x.

$$(F^{'})_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} \text{ and } (F^{''})_{ijk} = \frac{\partial^3 f}{\partial x_i \partial x_j \partial x_k}, \;\; 1 \le i, j, k \le n$$

In the symmetric case we will use the notation $\mathcal{T} = \nabla^3 f$ and the matrix $F^{''}s$ will be denoted $(s\mathcal{T})$.

Motivation

The following statements show that higher order methods is not considered practical.

(Ortega and Rheinboldt 1970) [9]: Methods which require second and higher order derivatives, are rather cumbersome from a computational view point. Note that, while computation of F' involves only n^2 partial derivatives $\partial_j F_i$, computation of $F^{''}$ requires n^3 second partial derivatives $\partial_j \partial_k F_i$, in general exorbiant amount of work indeed.

(Rheinboldt 1974) [10] Clearly, comparisons of this type turn out to be even worse for methods with derivatives of order larger than two. Except in the case $n = 1$, where all derivatives require only one function evaluation, the practical value of methods involving more than the first derivative of F is therefore very questionable.

(Rheinboldt 1998) [11]: Clearly, for increasing dimension n the required computational work soon outweighs the advantage of the higher-order convergence. From this view point there is hardly any justification to favor the Chebyshev method for large n.

3 Computations with the Third Derivative

Let f be a three times continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$. For a given $x \in \mathbb{R}^n$ let

$$g_i = \frac{\partial f(x)}{\partial x_i}, \; H_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}, \; \mathcal{T}_{ijk} = \frac{\partial^3 f(x)}{\partial x_i \partial x_j \partial x_k}, \;\; 1 \le i, j, k \le n. \tag{6}$$

Then H is a symmetric matrix and $\mathcal{T}$ a super-symmetric tensor. We say that a $n \times n \times n$ tensor is super-symmetric when

$$\begin{aligned} \mathcal{T}_{ijk} = \mathcal{T}_{ikj} = \mathcal{T}_{jik} = \mathcal{T}_{jki} = \mathcal{T}_{kij} = \mathcal{T}_{kji}, \quad & i \neq j, j \neq k, i \neq k \\ \mathcal{T}_{iik} = \mathcal{T}_{iki} = \mathcal{T}_{kii}, \quad & i \neq k. \end{aligned}$$

Since the entries of a super-symmetric tensor are invariant under any permutation of the indices we only need to consider $1 \leq k \leq j \leq i \leq n$. Further we only need to store the $\frac{1}{6}(n+2)(n+1)n$ nonzero elements $\mathcal{T}_{ijk}$ for $1 \leq k \leq j \leq i \leq n$ [8], as illustrated in Figure 1 for $n = 9$. For the matrix H we only need to store (the lower triangular part) H_{ij}, where $j \leq i$.

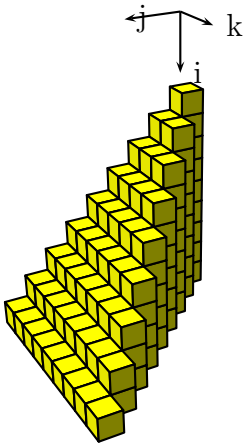

Figure 1: The stored elements for a dense super-symmetric tensor where $n = 9$.

In this section we present the following tensor computations

$$g = (p\mathcal{T})p \in \mathbb{R}^n \text{ and } H = (p\mathcal{T}) \in \mathbb{R}^{n \times n}$$

and $(p\mathcal{T})$ is a symmetric matrix. These operations are needed for the local methods in the Halley class.
The elements in $g = (p\mathcal{T})p$ are

$$g_i = \sum_{j=1}^{n} \sum_{k=1}^{n} p_j p_k \mathcal{T}_{ijk}, \quad 1 \leq i \leq n \tag{7}$$

By taking into account super-symmetry we have the following

$$g_i = g_i^{(1)} + g_i^{(2)} + g_i^{(3)} + g_i^{(4)} + g_i^{(5)} + g_i^{(6)}, \quad 1 \leq i \leq n \tag{8}$$

where we only use the elements in $\mathcal{T}_{ijk}$ so that $1 \leq k \leq j \leq i \leq n$.

$$g_i^{(1)} = \sum_{j=1}^{i}\sum_{k=1}^{j} p_j p_k \mathcal{T}_{ijk} \quad (9) \qquad g_i^{(4)} = \sum_{j=i+1}^{n}\sum_{k=1}^{i} p_j p_k \mathcal{T}_{jik} \quad (12)$$

$$g_i^{(2)} = \sum_{j=1}^{i}\sum_{k=j+1}^{i} p_j p_k \mathcal{T}_{ikj} \quad (10) \qquad g_i^{(5)} = \sum_{j=i+1}^{n}\sum_{k=i+1}^{j} p_j p_k \mathcal{T}_{jki} \quad (13)$$

$$g_i^{(3)} = \sum_{j=1}^{i}\sum_{k=i+1}^{n} p_j p_k \mathcal{T}_{kij} \quad (11) \qquad g_i^{(6)} = \sum_{j=i+1}^{n}\sum_{k=j+1}^{n} p_j p_k \mathcal{T}_{kji} \quad (14)$$

where $1 \leq i \leq n$.
The elements in $(p\mathcal{T})$ are

$$H_{ij} = \sum_{k=1}^{n} p_k \mathcal{T}_{ijk} = H_{ij}^{(1)} + H_{ij}^{(2)} + H_{ij}^{(3)}, \;\; 1 \leq i, j \leq n \quad (15)$$

using that a super-symmetric tensor is invariant under any permutation of the indices we have that

$$H_{ij}^{(1)} = \sum_{k=1}^{j} p_k \mathcal{T}_{ijk}, \;\; H_{ij}^{(2)} = \sum_{k=j+1}^{i} p_k \mathcal{T}_{ikj}, \;\; H_{ij}^{(3)} = \sum_{k=i+1}^{n} p_k \mathcal{T}_{kij}, \;\; 1 \leq j \leq i \leq n \quad (16)$$

which can be computed in Algorithm 1, 2 and 3.

Algorithm 1 $H^{(1)}$.

```
for i = 1 to n do
  for j = 1 to i do
    for k = 1 to j do
      H^(1)_ij += p_k T_ijk
    end for
  end for
end for
```

Algorithm 2 $H^{(2)}$.

```
for i = 1 to n do
  for j = 1 to i do
    for k = 1 to j − 1 do
      H^(2)_ik += p_j T_ijk
    end for
  end for
end for
```

Algorithm 3 $H^{(3)}$.

```
for i = 1 to n do
  for j = 1 to i−1 do
    for k = 1 to j do
      H^(3)_jk += p_i T_ijk
    end for
  end for
end for
```

If we want to combine (16) or Algorithm 1, 2 and 3 in one algorithm that computes the lower part of H there are four special cases that we must handle separately 1) the case when no indices are equal: $i \neq j$, $j \neq k$, $i \neq k$, 2) the case when two indices are equal: $k = j$, 3) the case when two indices are equal: $j = i$, and finally 4) the case when all three indices are equal. This is shown in Algorithm 4.

Algorithm 4 $H \leftarrow H + (p\mathcal{T})$.

$\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ is a super-symmetric tensor.
$H \in \mathbb{R}^{n\times n}$ is a symmetric matrix.
Let $p \in \mathbb{R}^n$.
for $i = 1$ to n **do**
  for $j = 1$ to $i-1$ **do**
    for $k = 1$ to $j-1$ **do**
      $H_{ij} += p_k\mathcal{T}_{ijk}$
      $H_{ik} += p_j\mathcal{T}_{ijk}$
      $H_{jk} += p_i\mathcal{T}_{ijk}$
    end for
    $H_{ij} += p_j\mathcal{T}_{ijj}$
    $H_{jj} += p_i\mathcal{T}_{ijj}$
  end for
  for $k = 1$ to $i-1$ **do**
    $H_{ii} += p_k\mathcal{T}_{iik}$
    $H_{ik} += p_i\mathcal{T}_{iik}$
  end for
  $H_{ii} += p_i\mathcal{T}_{iii}$
end for

Algorithm 5 $H \leftarrow H + (p\mathcal{T})$.

$\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ is a super-symmetric tensor.
$H \in \mathbb{R}^{n\times n}$ is a symmetric matrix.
Let $p \in \mathbb{R}^n$.
Let $\mathcal{C}_i$ is the nonzero index pattern of row i of the Hessian matrix.
for $i = 1$ to n **do**
  for $j \in \mathcal{C}_i \wedge j < i$ **do**
    for $k \in \mathcal{C}_i \cap \mathcal{C}_j \wedge k < j$ **do**
      $H_{ij} += p_k\mathcal{T}_{ijk}$
      $H_{ik} += p_j\mathcal{T}_{ijk}$
      $H_{jk} += p_i\mathcal{T}_{ijk}$
    end for
    $H_{ij} += p_j\mathcal{T}_{ijj}$
    $H_{jj} += p_i\mathcal{T}_{ijj}$
  end for
  for $k \in \mathcal{C}_i \wedge k < i$ **do**
    $H_{ii} += p_k\mathcal{T}_{iik}$
    $H_{ik} += p_i\mathcal{T}_{iik}$
  end for
  $H_{ii} += p_i\mathcal{T}_{iii}$
end for

When we extend the Algorithm 4, which is for the dense case, to the general sparse case, as shown in the next section (see Algorithm 5), we will have the same structure of the algorithm. The same would apply if we want to combine (9), (10), (11), (12), (14), and (13) into one algorithm.

4 The Sparsity Structure of the Tensor

In this section will show how we can utilize the sparsity structure of the Hessian matrix to induce the sparsity of the third derivative (tensor), that is we introduce the concept of induced sparsity. Finally, we will focus on objective functions where the Hessian matrix has a sparse structure.

Induced Sparsity

The sparsity of the second derivative $\nabla^2 f$ is defined to be [3]

$$\frac{\partial^2}{\partial x_i \partial x_j} f(x) = 0, \forall x \in \mathbb{R}^n, \;\; 1 \le i, j \le n. \tag{17}$$

If $\mathcal{Z}$ is the set of indices where (17) holds, define

$$\mathcal{N} = \{(i,j)|1 \leq i,j \leq n\} \backslash \mathcal{Z} \tag{18}$$

and $\mathcal{N}$ will be the set of indices for which the elements in the Hessian matrix at x in general will be nonzero.

It follows that

$$\mathcal{T}_{ijk} = 0, \text{ if } (i,j) \in \mathcal{Z}, (j,k) \in \mathcal{Z} \text{ or } (i,k) \in \mathcal{Z}$$

we only need to consider the elements (i,j,k) in the tensor, that is

$$(i,j) \in \mathcal{N} \text{ and } (j,k) \in \mathcal{N} \text{ and } (i,k) \in \mathcal{N}, \;\; 1 \leq k \leq j \leq i \leq n.$$

We say that the sparsity structure of the third derivative (tensor) is induced by the sparsity structure of the Hessian matrix.

In the following we will assume that $(i,i) \in \mathcal{N}$. Define

$$T = \{(i,j,k)|1 \leq k \leq j \leq i \leq n, (i,j) \in \mathcal{N}, (j,k) \in \mathcal{N}, (i,k) \in \mathcal{N}\} \tag{19}$$

and let $\mathcal{C}_i$ be the nonzero indices in row i below the diagonal of the sparse Hessian matrix:

$$\mathcal{C}_i = \{j|j \leq i, (i,j) \in \mathcal{N}\}, \quad i = 1, \ldots, n. \tag{20}$$

Further, let $\mathcal{R}_j$ be the nonzero elements in column j below the diagonal

$$\mathcal{R}_j = \{i|i \geq j, (i,j) \in \mathcal{N}\}, \quad j = 1, \ldots, n. \tag{21}$$

It follows that

$$\begin{aligned} T &= \{(i,j,k)|i = 1, ..., n, j \in \mathcal{C}_i, k \in \mathcal{C}_i \cap \mathcal{C}_j\} \\ &= \{(i,j,k)|j = 1, ..., n, i \in \mathcal{R}_j, k \in \mathcal{R}_i \cap \mathcal{R}_j\} \\ &= \{(i,j,k)|i = 1, ..., n, k \in \mathcal{C}_i, j \in \mathcal{R}_k \cap \mathcal{C}_i\}. \end{aligned}$$

We can immediately discard the choice of using both the sets of row and column indices. The intersection of $\mathcal{C}_i$ and $\mathcal{C}_j$ gives rise to a tube [1] oriented approach. Thus the preferable choice for representing the nonzero elements of the tensor $\mathcal{T}_{ijk}$ is

$$T = \{(i,j,k)|i = 1, ..., n, j \in \mathcal{C}_i, k \in \mathcal{C}_i \cap \mathcal{C}_j\}. \tag{22}$$

The number of stored elements of the Hessian matrix is

$$nnz(H) = |\mathcal{N}| = \sum_{i=1}^{n} |\mathcal{C}_i|,$$

and the number of stored elements of the induced tensor is

$$nnz(\mathcal{T}) = |T| = \sum_{i=1}^{n} \sum_{j \in C_i} |\mathcal{C}_i \cap \mathcal{C}_j|$$

This is further illustrated in the Figures 2, 3, 4 and 5.

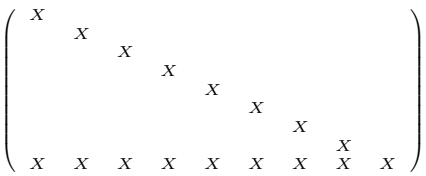

Figure 2: Stored elements of a symmetric arrowhead matrix.

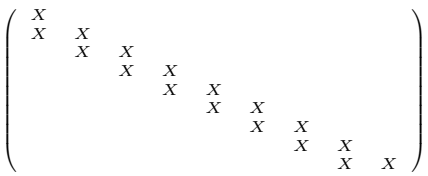

Figure 3: Stored elements of a symmetric tridiagonal matrix.

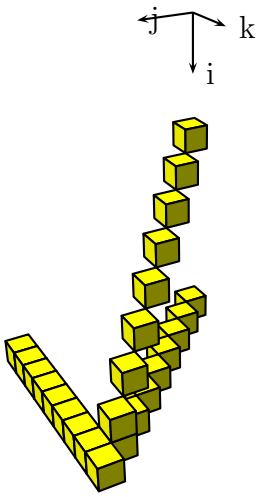

Figure 4: Stored elements of the tensor induced by an arrowhead symmetric matrix where $n = 9$.

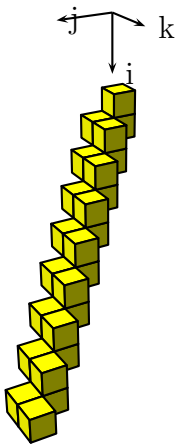

Figure 5: Stored elements of the tensor induced by an tridiagonal symmetric matrix where $n = 9$.

Algorithm 6 shows how to compute $g \leftarrow g + (p\mathcal{T})p$ and Algorithm 5 shows how to compute $H \leftarrow H + (p\mathcal{T})$ for a general sparse tensor, both using the definition (22).

Algorithm 6 $g \leftarrow g + (p\mathcal{T})p$.

$\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ is a super-symmetric tensor.
Let $p, g \in \mathbb{R}^n$.
Let $s \in \mathbb{R}$.
Let $\mathcal{C}_i$ is the nonzero index pattern of row i of the Hessian matrix.

```
for i = 1 to n do
  for j ∈ C_i ∧ j < i do
    s = 0
    for k ∈ C_i ∩ C_j ∧ k < j do
      s+ = p_k T_ijk
      g_k+ = 2 p_i p_j T_ijk
    end for
    g_i+ = p_j (2s + p_j T_ijj)
    g_j+ = 2 p_i (s + p_j T_ijj)
  end for
  s = 0
  for k ∈ C_i ∧ k < i do
    s+ = p_k T_iik
    g_k+ = p_i^2 T_iik
  end for
  g_i+ = p_i (2s + p_i T_iii)
end for
```

Algorithm 7 Algorithmic skeleton.

```
ind = 0;
for i = 0 to n − 1 do
   Compute t_k^(i) = true if k ∈ C_i
  for j ∈ C_i ∧ j > i do
    for k ∈ C_j ∧ k < j do
      if t_k^(i) then
        T_ijk = T[ind]; ind + +;
        ...
      end if
    end for
    T_ijj = T[ind]; ind + +;
    ...
  end for
  for k ∈ C_i ∧ k < i do
    T_iik = T[ind]; ind + +;
    ...
  end for
  T_iii = T[ind]; ind + +;
  ...
end for
```

An ANSI C Implementation with Á Priori Storage

In this section we present an ANSI C implementation of the operation $g \leftarrow g+(p\mathcal{T})p$. For every $(i, j) \in \mathcal{N}$ (18) we need either to store or compute the indices $k \in T$ (22). In this implementation we have á priori storage of the indices for each tube., i.e. for $j \leq i$, $j \in \mathcal{C}_i$ the indices $k \in \mathcal{C}_i \cap \mathcal{C}_j$, of the tensor induced by a general sparse Hessian matrix. Thus the tensor is stored as a general sparse tensor. Consider the following matrix H,

$$H = \begin{pmatrix} 1 & & & & & & \\ 2 & 3 & & & & & \\ & & 4 & & & & \\ & 5 & & 6 & & & \\ & & 7 & & 8 & & \\ 9 & & 10 & & 11 & 12 & \end{pmatrix}$$

Figure 6: Stored elements of a symmetric general matrix.

The Symmetric Compressed Row Storage (CRS) of the H matrix:

```
double valueH[] = {1,2,3,4,5,6,7,8,9,10,11,12};
int indexH[]    = {0,0,1,2,1,3,2,4,0,2,4,5};
int pointerH[]  = {0,1,3,4,6,8,12};
```

The Super-Symmetric Compressed Tube Storage (CTS) of the induced tensor:

```
double valueT[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};
int indexT[]    = {0,0,0,1,2,1,1,3,2,2,4,0,2,2,4,0,2,4,5};
int pointerT[]  = {0,1,2,4,5,6,8,9,11,12,13,15,19};
```

The operation $g \leftarrow g + (p\mathcal{T})p$ using the above data structures:

```
int start = 0, stop = 0, i = 0, j = 0, k = 0;
int ind = 0, tp = 0;
int pi = 0, pj = 0, pk = 0, pipj = 0, kpipj = 0, kpi = 0, kpj = 0, pipi = 0, pkTijk=0;
int startTubek=0, stopTubek=0;
double Tijk = 0, Tijj = 0, Tiii = 0, Tiik = 0;
double ga = 0;
for(i = 0;i<N;i++,ind++,tp++){
    start = pointerH[i];
    j = indexH[start];
    pi = p[i];
    kpi = 2*pi;
    pipi = pi*pi;
    for(;j<i;start++,j=indexH[start],ind++,tp++){
        pj = p[j];
        kpj = 2*pj;
        kpipj = pj*kpi;
        startTubek = pointerT[tp];
        stopTubek  = pointerT[tp+1]-1;
        for(;startTubek<stopTubek;startTubek++,ind++){
            //Handle the case when no indices are equal: i!=j!=k!=i
            Tijk = valueT[ind];
            pkTijk = p[indexT[ind]]*Tijk;
            ga += pkTijk;
            g[k] += kpipj*Tijk;
        }
        //Handle the case when two indices are equal: k=j
        Tijj = valueT[ind];
        g[i] += (kpj*ga+pj*pj*Tijj);
        g[j] += (kpi*ga+kpipj*Tijj);
        ga = 0.0;
    }
    startTubek = pointerT[tp];
    stopTubek  = pointerT[tp+1]-1;
    for(;startTubek<stopTubek;startTubek++,ind++){
        //Handle the case when two indices are equal: j=i
        Tiik = valueT[ind];
        ga += p[indexT[ind]]*Tiik;
        g[k] += pipi*Tiik;
    }
    //Handle the case when all three indices are equal
    g[i] += (kpi*ga + pipi*valueT[ind]);
    ga = 0.0;
}
```

The Hessian matrix is stored by CRS where we store all the values, $nnz(H)$ floating point numbers, and $nnz(H)$ row indices, and finally $n+1$ pointers to each row. The tensor is stored by CTS where we store all the values, $nnz(\mathcal{T})$ floating point numbers, and $nnz(\mathcal{T})$ tube indices, finally $nnz(H)+1$ pointers to each tube.

Less Memory More Flops

If memory is an issue we can save $nnz(\mathcal{T})$ storage by not storing the indices of the induced tensor. Then let `T` be the one-dimensional storage of the general sparse super-symmetric tensor $\mathcal{T}$, where all the tensor elements are stored contiguously.

Further we have a boolean array t of length n where an element $t_k =$`true` if $k \in C_i$. So in the innermost loop for each $k \in C_j$ we must check that $t_k^{(i)}$ is `true`. If that is the case we perform the tensor operation if not we skip it. Thus we have taken the intersection $k \in \mathcal{C}_i \cap \mathcal{C}_j$. This is illustrated in the skeleton Algorithm 7. Clearly we perform additional loop iterations and floating point operations so the CPU time will increase compared to an á priori storage, as shown in the section above.

Numerical Results

Consider the following test functions: Chained Rosenbrock [13] and Generalized Rosenbrock [12]. Both Generalized Rosenbrock and Chained Rosenbrock have a Hessian matrix which is sparse. We compare Newton's method, Chebyshev's method, Super Halley's method and Halley's method. The test cases show that the third order methods are competitive with Newton's method. The termination criteria for all methods are if $\|\nabla f(x_k)\| \leq 10^{-8}\|\nabla f(x_0)\|$.

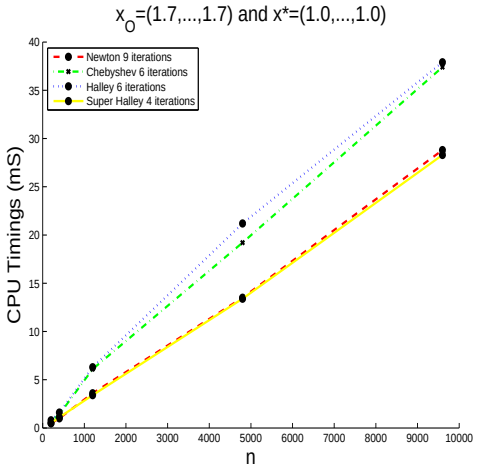

Figure 7: Chained Rosenbrock.

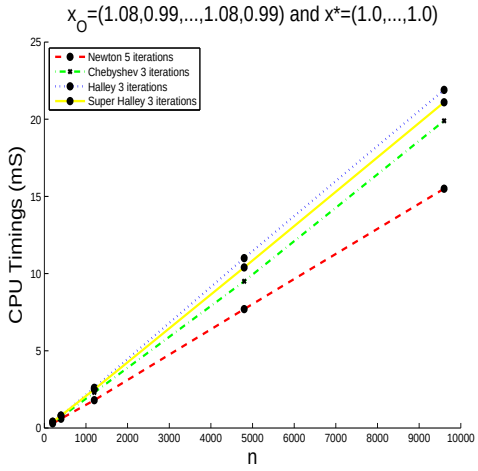

Figure 8: Generalized Rosenbrock.

Figure (7) and (8) shows that the total CPU time to solve the unconstrained optimization problem (5) increases linearly in the number of unknowns and the CPU time is almost identical. If there should be any gain in using third order methods they must have fewer iterations then the second order methods, since the computational cost is higher for each iteration for the third order method. How many fewer iterations the third order methods must have is very problem dependent, since the CPU time is also very dominated by the cost of function, gradient, Hessian and tensor evaluations.

Conclusions

In this paper we have seen examples of third order *local* methods that are competitive with second order methods (Newton). Thus the use of exact third order derivatives

is practical from a computational view. The concept of induced sparsity makes it possible to create data structure and algorithms for tensor operations in a straightforward manner. We have presented some examples of data structures for the tensor for induced by a general sparse Hessian matrix.

References

[1] B. W. Bader and T. G. Kolda. Matlab tensor classes for fast algorithm prototyping. Technical Report 2004-5187, Sandia National Laboratories, Oct. 2004.

[2] N. Deng and H. Zhang. Theoretical efficiency of a new inexact method of tangent hyperbolas. *Optimization Methods and Software*, 19(3-4):247–265, June-August 2004.

[3] A. Griewank and P. L. Toint. On the unconstrained optimization of partially separable functions. *In Michael J. D. Powell, editor, Nonlinear Optimization*, pages 247–265, 1981.

[4] G. Gundersen and T. Steihaug. Cubically convergent methods are no more expensive (almost) than Newton's method. Submitted to The Veszprm Optimization Conference: Advanced Algorithms held in Hungary, December 13-15, 2006.

[5] G. Gundersen and T. Steihaug. Sparsity in higher order methods in optimization. Technical Report 327, Department of Informatics, University of Bergen, June 2006.

[6] J. M. Gutierrez and M. A. Hernandez. An acceleration of Newton's method: Super-Halley method. *Applied Mathematics and Computation*, 117(2):223–239, January 2001.

[7] D. Han. The convergence on a family of iterations with cubic order. *Journal of Computational Mathematics*, 19(5):467–474, 2001.

[8] D. E. Knuth. *Fundamental Algorithms,vol.1 of The Art of Programming, 2nd edition.* Addison-Wesley, Reading, MA, 1973.

[9] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, New York, 1970.

[10] W. C. Rheinboldt. *Methods for Solving Systems of Equations of Nonlinear Equations.* Reg. Conf. Ser. in Appl. Math, Vol. 14. SIAM Publications, Philadelphia, PA, 1974.

[11] W. C. Rheinboldt. *Methods for Solving Systems of Equations of Nonlinear Equations, Second edition.* Regional Conf. Series in Appl. Math., Vol. 70. SIAM Publications, Philadelphia, PA, 1998.

[12] H. P. Schwefel. *Numerical Optimization of Computer Models.* John Wiley and Sons, Chichester, 1981.

[13] P. Toint. Some numerical results using a sparse matrix updating formula in unconstrained optimization. *Mathematics of Computation*, 32(143):839–851, July 1978.