REPORTS IN INFORMATICS

ISSN 0333-3590

Sparsity in Higher Order Methods in Optimization

Geir Gundersen and Trond Steihaug

REPORT NO 327 June 2006

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL http://www.ii.uib.no/publikasjoner/texrap/ps/2006-327.ps

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at
http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

Sparsity in Higher Order Methods in Optimization

Geir Gundersen and Trond Steihaug
Department of Informatics
University of Bergen
Norway
{geirg,trond}@ii.uib.no

29th June 2006

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 unconstrained optimization problems when the objective function is three times continuously differentiable. It is also shown how to arrange the computations of the higher 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 are competitive compared to second order methods (Newton).

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 a skyline structure.

We show that classical third order methods, i.e Halley's, Chebyshev's and Super Halley's method, can all be regarded as two step Newton like methods. We present numerical results on third order *local* methods that utilizes the sparsity and super-symmetry where the computational cost is so low that they are competitive with Newton's method.

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. Further, there is an increased memory requirement. For sparse systems we show that this increase and the increase in arithmetic operations are very modest. If we compare the number of arithmetic operations to compute the gradient of the third order Taylor approximation to the second order approximation (as for Newton), we find that this ratio can be expressed as the ratio of the memory requirements. In the case for banded Hessian matrices with (half) bandwidth β, this ratio is bounded by $\frac{\beta+2}{2}$. We focus on objective functions where the Hessian matrix has a skyline (or envelope) structure, and we briefly discuss general sparse Hessian matrices and tensors.

In section 2 we discuss methods that have a cubically convergence rate. We show that these methods may be regarded as two step of Newton's method. In section 3 we briefly introduce a global method based on trust-regions. We show that the number of (outer) ietrations using a cubic model is lower than using a quadratic model. In section 4 we summarize the computations using the higher order (third) derivative and show how to utilize the super symmetry. In section 5 we introduce the concept of induced sparsity of the third derivative. In section 6 we briefly discuss data structures for induced sparse tensors. Finally, in section 7 we give some numerical results on the cost going from quadratic to cubic operations, and introduce an efficiency ratio indicator.

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 [13] for solving (1).

$$x_{k+1} = x_k - \{I + \frac{1}{2}L(x_k)[I - \alpha L(x_k)]^{-1}\}(F^{'}(x_k))^{-1}F(x_k), \quad k = 0, 1, \ldots, \tag{2}$$

where

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

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 the Chebyshev's Method [1, 6, 14, 21, 22, 27, 28, 29], Halley's Method [1, 4, 5, 6, 7, 8, 14, 15, 20, 21, 22, 24, 27, 28, 29, 30] and Super Halley's method [7, 13, 14]. 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 $s_k^{(2)}$:

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

The new step is

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

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

$$F(x_k + s_k^{(1)}) \simeq F(x_k) + F^{'}(x_k)s_k^{(1)} + \frac{1}{2}F^{''}(x_k)s_k^{(1)}s_k^{(1)} = \frac{1}{2}F^{''}(x_k)s_k^{(1)}s_k^{(1)}$$

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.

2.1 Unconstrained Optimization

The methods in the previous section also apply for algorithms for the unconstrained optimization problem

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

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is sufficiently smooth and the Hessian $\nabla^2 f(x^*)$ is symmetric positive definite.
The necessary condition at a solution (5) is that the gradient is zero

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

Thus in (1) we have that

$$F(x) = \nabla f(x) \text{ or } F_i = \frac{\partial f}{\partial x_i}(x), \quad i = 1, \ldots, n.$$

Then we have for unconstrained optimization that $F' = \nabla^2 f(x)$ is the Hessian matrix and $F^{''} = \nabla^3 f(x)$ is the third derivatives of f at x.

2.2 Motivation

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

(Ortega and Rheinboldt 1970) [20]: 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) [21] 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) [22]: 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.

A certain modification can be seen - and with recent numerical experiments that utilizes sparsity then these statements are contradicted.

Consider the following test functions: Chained Rosenbrock [26] and Generalized Rosenbrock [23]. Generalized Rosenbrock has a Hessain matrix of arrowhead structure, while for the Generalized Rosenbrock the Hessian matrix structure is banded.

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, also for increasing n.

The termination criteria for all methods are if $\|\nabla f(x_k)\| \leq 10^{-8} \|\nabla f(x_0)\|$.

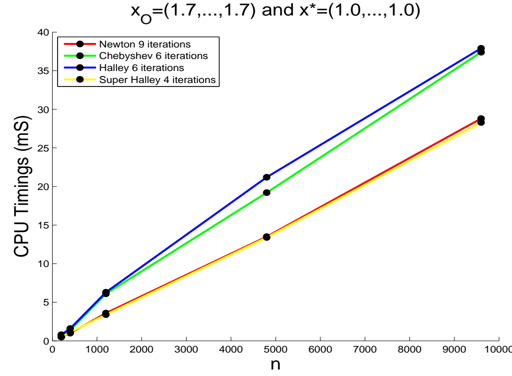

Figure 1: Chained Rosenbrock.

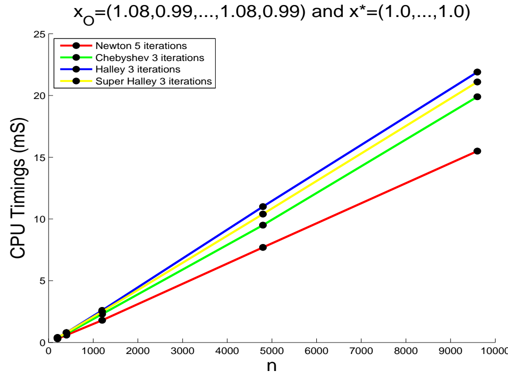

Figure 2: Generalized Rosenbrock.

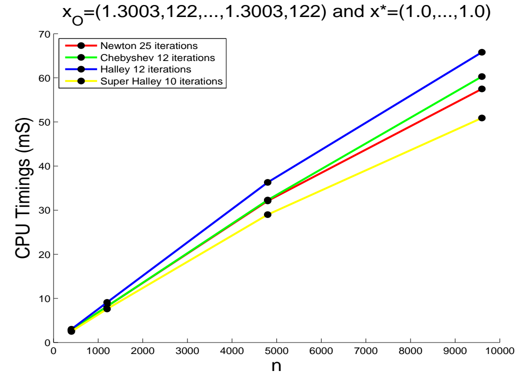

Figure 3: Generalized Rosenbrock.

Figure (1), Figure (2) and Figure (3) 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.

3 Higher Order Global Methods

A third order Taylor approximation of $f(x+p)$ evaluated at x is

$$m(p) = f + g^T p + \frac{1}{2} p^T H p + \frac{1}{6} p^T (p\mathcal{T}) p \tag{6}$$

where $f = f(x)$ and we can expect the model $m(p)$ to be a good approximation for $\|p\|$.

Algorithms for the unconstrained optimization problem

$$\min_{x \in \mathbb{R}^n} f(x) \tag{7}$$

generates a sequence of iterates x_k where at every iterate x_k we generate a model function $m(p)$ that approximates the function. The new iterate is $x_{k+1} = x_k + p$ provided that we have a sufficiently good model.

To solve the unconstrained optimization problem we have implemented a trust-region method [19], both with a cubic and a quadratic model. The trust-region method is a global method, thus it guarantees that it finds an x^* where $\|\nabla f(x^*)\| = 0$, for any starting points.

The trust-region method will require the value, the gradient or the Hessian matrix

$$m(p), \;\; \nabla m(p) = g + Hp + \frac{1}{2}(p\mathcal{T})p, \;\; \nabla^2 m(p) = H + (p\mathcal{T}) \tag{8}$$

of the cubic model (6).

The Trust-Region Subproblem (TRS) (9) must be solved for each iteration of the trust-region method.

$$\min_{p \in R^n} m(p) = g^T p + \frac{1}{2}p^T Hp + \frac{1}{6}p^T(p\mathcal{T})p \quad \text{s.t } \|p\| \leq \Delta \tag{9}$$

Algorithm 1 Trust-Region Method

Let $0 < \gamma_2 \leq \gamma_3 < 1 \leq \gamma_1$, $0 < \beta_0$ and $0 < \beta_1 < 1$
Given $\hat{\Delta} > 0$, $\Delta_0 \in (0, \hat{\Delta})$, and $\eta \in [0, \beta_0)$
for $k = 0, 1, 2, \ldots$ **do**
  Compute $\nabla f(x_k)$, $\nabla^2 f(x_k)$ and $\nabla^3 f(x_k)$.
  Determine an (approximate) solution p_k to the TRS (9).
  Compute $\rho_k = (f(x_k) - f(x_k + p_k))/(m_k(0) - m_k(p_k))$.
  if $\rho_k < \beta_0$ **then**
    $\gamma_2\|p_k\| \leq \Delta_{k+1} \leq \gamma_3\|p_k\|$
  else
    if $\rho_k > \beta_1$ and $\|p_k\| = \Delta_k$ **then**
      $\|p_k\| \leq \Delta_{k+1} \leq \min\{\gamma_1\|p_k\|, \hat{\Delta}\}$
    end if
  end if
  if $\rho_k > \eta$ **then**
    $x_{k+1} = x_k + p_k$
  else
    $x_{k+1} = x_k$
  end if
end for

Preferable values for the constant in Algorithm 1 are $\beta_0 = \frac{1}{4}$, $\beta_1 = \frac{3}{4}$, $\gamma_1 = 2$, and $\gamma_2 = \gamma_3 = \frac{1}{2}$.

3.1 Trust-Region Iterations for Quadratic and Cubic Model

In this section we compare the trust-region method with a cubic model to a trust-region method with a quadratic model on the number of iterations from a starting point x_0 to a solution x^*, where $\|\nabla f(x^*)\| = 0$.

We have taken several test functions and tested them for different n and starting points. Chained Rosenbrock $x_0 = \{(-1.2, 1.0, \ldots, -1.2, 1.0), (-1.0, -1.0, \ldots, -1.0, -1.0)\}$ and $x^* = (1.0 \ldots, 1.0)$, Generalized Rosenbrock $x_0 = (-1.2, 1.0, \ldots, -1.2, 1.0)$ and $x^* = (1.0 \ldots, 1.0)$, BroydenTridiagonal $x_0 = (-1.0, -1.0, \ldots, -1.0, -1.0)$ and $x^* = (-0.57, \ldots, -0.42)$, and Beale [3] $x_0 = \{(4, 4), (3, 3), (2, 2)\}$ and $x^* = (3.5, 0.5)$

Table (1) shows that the trust-region method with a cubic method (C) uses fewer iterations then with a quadratic model (Q), to get to a solution.

The termination criteria for both methods are if $\|\nabla f(x_k)\| \leq 10^{-8}\|\nabla f(x_0)\|$.

Table 1: Trust-Region Method.

Trust-Region Iterations for Quadratic and Cubic Model														
Chained Rosenbrock			Chained Rosenbrock			Generalized Rosenbrock			BroydenTridiagonal			Beale		
n	Q	C	n	Q	C	n	Q	C	n	Q	C	n	Q	C
2	22	15	4	14	12	4	12	6	4	6	4	2	18	12
6	29	16	8	14	10	8	10	6	20	6	4	2	16	9
18	27	19	10	15	12	10	10	6	30	6	4	2	14	11

4 Computations of the Cubic Model

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

$$g_i = \frac{\partial f(x)}{\partial x_i},\ H_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j},\ \mathcal{T}_{ijk} = \frac{\partial^3 f(x)}{\partial x_i \partial x_j \partial x_k}\ , 1 \le k \le j \le i \le n. \tag{10}$$

Then H is a symmetric matrix and $\mathcal{T}$ a super-symmetric tensor, $g \in \mathbb{R}^n$, $H \in \mathbb{R}^{n\times n}$, and $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ using the notation in [2]. We say that a $n \times n \times n$ tensor is super-symmetric when

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

Since the entries of a super-symmetric tensor are invariant under any permutation of the indices we only need to store the $\frac{1}{6}(n+2)(n+1)n$ nonzero elements $\mathcal{T}_{ijk}$ for $1 \le k \le j \le i \le n$, as illustrated in Figure 4 for $n = 9$.

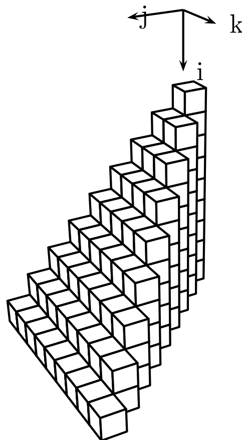

Figure 4: The stored elements for a dense super-symmetric tensor where $n = 9$.

We will present the following tensor computations of the cubic model (4)

$$p^T(p\mathcal{T})p,\ (p\mathcal{T})p \text{ and } (p\mathcal{T})$$

where $p^T(p\mathcal{T})p \in \mathbb{R}$ is the cubic value term, $(p\mathcal{T})p \in \mathbb{R}^n$ is the gradient term and $(p\mathcal{T}) \in \mathbb{R}^{n\times n}$ is the Hessian term. The cubic value term is a scalar, the cubic gradient term is a vector, and the cubic Hessian term is a matrix. These operations are needed for the local methods (The Halley class) and the global method (trust-region).

The cubic value term can is defined as

$$p^T(p\mathcal{T})p = \sum_{i=1}^{n} p_i \sum_{j=1}^{n} p_j \sum_{k=1}^{n} p_k \mathcal{T}_{ijk}. \tag{11}$$

By taking into account super-symmetry, $1 \le k \le j \le i \le n$, we have the following

$$p^T(p\mathcal{T})p = \sum_{i=1}^{n} p_i \left\{ \left[\sum_{j=1}^{i-1} p_j \left(6 \sum_{k=1}^{j-1} p_k \mathcal{T}_{ijk} + 3p_j \mathcal{T}_{ijj} \right) + 3p_i \sum_{k=1}^{i-1} p_k \mathcal{T}_{iik} \right] + p_i^2 \mathcal{T}_{iii} \right\}. \tag{12}$$

The computation (12) leads to the following algorithm

Algorithm 2 Computing $c \leftarrow c + p^T(p\mathcal{T})p$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be a super-symmetric tensor.
Let $p \in \mathbb{R}^n$.
Let $c, s, t \in \mathbb{R}$.
for $i = 1$ to n **do**
  $t = 0$
  for $j = 1$ to $i - 1$ **do**
    $s = 0$
    for $k = 1$ to $j - 1$ **do**
      $s+ = p_k \mathcal{T}_{ijk}$
    end for
    $t+ = p_j(6s + 3p_j\mathcal{T}_{ijj})$
  end for
  $s = 0$
  for $k = 1$ to $i - 1$ **do**
    $s+ = p_k \mathcal{T}_{iik}$
  end for
  $c+ = p_i(t + p_i(3s + p_i\mathcal{T}_{iii}))$
end for

Algorithm 2 requires $\frac{1}{3}n^3 + 3n^2 + \frac{11}{3}n$ number of arithmetic operations.
The cubic gradient term can be defined as

$$g_i = \sum_{j=1}^{n} \sum_{k=1}^{n} p_j p_k \mathcal{T}_{ijk}, \quad 1 \le i \le n. \tag{13}$$

By taking into account super-symmetry we have the following

$$g_i = g_i^{(1)} + g_i^{(2)} + g_i^{(3)} + g_i^{(4)} + g_i^{(5)} + g_i^{(6)}, \quad 1 \le i \le n, \tag{14}$$

where

$$g_i^{(1)} = \sum_{j=1}^{i} \sum_{k=1}^{j} p_j p_k \mathcal{T}_{ijk}, \quad 1 \le i \le n \tag{15}$$

$$g_i^{(2)} = \sum_{j=1}^{i} \sum_{k=j+1}^{i} p_j p_k \mathcal{T}_{ijk} = \sum_{j=1}^{i} \sum_{k=j+1}^{i} p_j p_k \mathcal{T}_{ikj}, \quad 1 \le i \le n, \tag{16}$$

$$g_i^{(3)} = \sum_{j=1}^{i} \sum_{k=i+1}^{n} p_j p_k \mathcal{T}_{ijk} = \sum_{j=1}^{i} \sum_{k=i+1}^{n} p_j p_k \mathcal{T}_{kij}, \quad 1 \le i \le n, \tag{17}$$

$$g_i^{(4)} = \sum_{j=i+1}^{n} \sum_{k=1}^{i} p_j p_k \mathcal{T}_{ijk} = \sum_{j=i+1}^{n} \sum_{k=1}^{i} p_j p_k \mathcal{T}_{jik}, \quad 1 \le i \le n, \tag{18}$$

$$g_i^{(5)} = \sum_{j=i+1}^{n} \sum_{k=i+1}^{j} p_j p_k \mathcal{T}_{ijk} = \sum_{j=i+1}^{n} \sum_{k=i+1}^{j} p_j p_k \mathcal{T}_{jki}, \quad 1 \le i \le n, \tag{19}$$

$$g_i^{(6)} = \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} p_j p_k \mathcal{T}_{ijk} = \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} p_j p_k \mathcal{T}_{kji}, \quad 1 \le i \le n. \tag{20}$$

If we combine the computations of (15) to (20) we get the following algorithm.

Algorithm 3 Computing $g \leftarrow g + (p\mathcal{T})p$.

Let $\mathcal{T} \in \mathbb{R}^{n \times n \times n}$ be a super-symmetric tensor.
Let $p, g \in \mathbb{R}^n$.
Let $s \in \mathbb{R}$.
for $i = 1$ to n **do**
  $s = 0$
  for $j = 1$ to $i - 1$ **do**
    for $k = 1$ to $j - 1$ **do**
      $s += p_k \mathcal{T}_{ijk}$
      $g_k += 2 p_i p_j \mathcal{T}_{ijk}$
    end for
    $g_i += p_j(2s + p_j \mathcal{T}_{ijj})$
    $g_j += 2p_i(s + p_j \mathcal{T}_{ijj})$
  end for
  $s = 0$
  for $k = 1$ to $i - 1$ **do**
    $s += p_k \mathcal{T}_{iik}$
    $g_k += p_i^2 \mathcal{T}_{iik}$
  end for
  $g_i += (2p_i s + p_i^2 \mathcal{T}_{iii})$
end for

Algorithm 3 requires $\frac{2}{3}n^3 + 6n^2 - \frac{2}{3}n$ number of arithmetic operations.
The cubic Hessian term can be defined as

$$H_{ij} = \sum_{k=1}^{n} p_k \mathcal{T}_{ijk}, \quad 1 \le i, j \le n. \tag{21}$$

By taking into account the symmetries we have the following

$$H_{ij} = \sum_{k=1}^{j} p_k \mathcal{T}_{ijk} + \sum_{k=j+1}^{i} p_k \mathcal{T}_{ijk} + \sum_{k=i+1}^{n} p_k \mathcal{T}_{ijk}, \quad j \le i \tag{22}$$

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

$$H_{ij} = H_{ij}^{(1)} + H_{ij}^{(2)} + H_{ij}^{(3)}, \;\; 1 \le j \le i \le n \tag{23}$$

where

$$H_{ij}^{(1)} = \sum_{k=1}^{j} p_k \mathcal{T}_{ijk}, \;\; 1 \le j \le i \le n, \tag{24}$$

$$H_{ij}^{(2)} = \sum_{k=j+1}^{i} p_k \mathcal{T}_{ikj}, \;\; 1 \le j \le i \le n, \tag{25}$$

$$H_{ij}^{(3)} = \sum_{k=i+1}^{n} p_k \mathcal{T}_{kij}, \;\; 1 \le j \le i \le n. \tag{26}$$

If we combine (24), (25) and (26) we get the following algorithm that computes the lower part of H.

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

```
Let T ∈ R^{n×n×n} be a super-symmetric tensor.
Let H ∈ R^{n×n} be a symmetric matrix.
Let p ∈ 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 T_ijk
      H_ik += p_j T_ijk
      H_jk += p_i T_ijk
    end for
    H_ij += p_j T_ijj
    H_jj += p_i T_ijj
  end for
  for k = 1 to i − 1 do
    H_ii += p_k T_iik
    H_ik += p_i T_iik
  end for
  H_ii += p_i T_iii
end for
```

Algorithm 4 requires $n^3 + n^2$ number of arithmetic operations.

5 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 derivatives (tensor), that is we introduce the concept of induced sparsity. Finally, we will focus on objective functions where the Hessian matrix has a skyline (or envelope) structure. We use the structure of partially separable functions to derive the structure of the Hessian matrices and thus the tensor.

5.1 Induced Sparsity

The sparsity of the Hessian matrix is defined to be [10]

$$\frac{\partial^2}{\partial x_i \partial x_j} f(x) = 0, \forall x \in \mathbb{R}^n, \ \ 1 \le i, j \le n. \tag{27}$$

Define

$$C_i = \{ j \le i | \exists x \in \mathbb{R}^n \text{ so that } \frac{\partial^2}{\partial x_i \partial x_j} f(x) \neq 0 \} \tag{28}$$

C_i is the nonzero index structure of row i of the Hessian matrix. We assume that the diagonal element is included in the index structure of the Hessian matrix, i.e. $i \in C_i$.

Theorem 2. *The sparsity structure of the tensor $\mathcal{T}$ is determined by the sparsity structure (27) of the Hessian matrix.*

Proof. Assume that $j \notin C_i$, $j \le i$. It follows that

$$\mathcal{T}_{ijk} = 0, \ \ 1 \le k \le j.$$

□

We say that the sparsity structure of the tensor is induced by the sparsity structure of the tensor.

This is further illustrated in the Figures (5), (6), (7) and (8).

$$\begin{pmatrix} X & & & & & & & & \\ & X & & & & & & & \\ & & X & & & & & & \\ & & & X & & & & & \\ & & & & X & & & & \\ & & & & & X & & & \\ & & & & & & X & & \\ & & & & & & & X & \\ X & X & X & X & X & X & X & X & X \end{pmatrix}$$

Figure 5: Stored elements of a symmetric arrowhead matrix.

$$\begin{pmatrix} X & & & & & & & & \\ X & X & & & & & & & \\ & X & X & & & & & & \\ & & X & X & & & & & \\ & & & X & X & & & & \\ & & & & X & X & & & \\ & & & & & X & X & & \\ & & & & & & X & X & \\ & & & & & & & X & X \end{pmatrix}$$

Figure 6: Stored elements of a symmetric tridiagonal matrix.

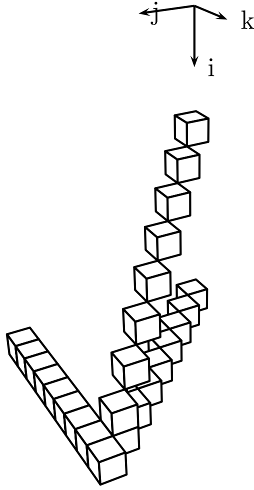

Figure 7: Stored elements of the tensor induced by an arrowhead symmetric matrix where $n = 9$.

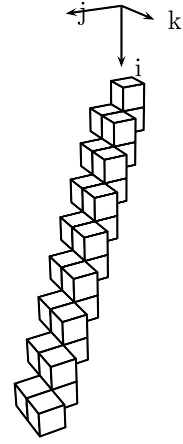

Figure 8: Stored elements of the tensor induced by an tridiagonal symmetric matrix where $n = 9$.

5.2 A Skyline Matrix

In a symmetric skyline storage mode, all matrix elements from the first nonzero in each row to the diagonal in the row are explicitly stored.

We define β_i to be the (lower) bandwidth of row i,

$$\beta_i = \max\{i - j| \text{ for nonzero } H_{ij}, j \leq i\}.$$

Further, define f_i to be the start index for row i in the Hessian matrix as

$$f_i = i - \beta_i. \tag{29}$$

The skyline storage requirement for a symmetric skyline storage (only the lower triangle need to be stored), is $\sum_i \beta_i + n$.

5.3 Partially Separable functions

A function on the form

$$f = \sum_i \phi_i$$

is partially separable where each element function ϕ_i depends on only a few components of x [10]. It follows that the gradients $\nabla\phi_i$ and $\nabla^2\phi_i$ of each element function contains just a few nonzeros.

5.3.1 A Skyline Structure

Theorem 1. *If the function $f = \sum_j \phi_j$ is partially separable on the form*

$$f(x) = \sum_{j=1}^{m} \phi_j(x_{j-\beta_j^{(l)}}, x_{j-\beta_j^{(l)}+1}, \ldots, x_{j+\beta_j^{(u)}-1}, x_{j+\beta_j^{(u)}})$$

and $\beta_j = \beta_j^{(l)} + \beta_j^{(u)} \geq 0$, $j = 1, 2, \ldots, m$. Then the Hessian matrix has a skyline structure, and f_i (29) is monotonically increasing.

Proof. The gradient of f is

$$\begin{aligned} \frac{\partial f}{\partial x_i}(x) &= \sum_{j=1}^{m} \frac{\partial}{\partial x_i} \phi_j(x_{j-\beta_j^{(l)}}, x_{j-\beta_j^{(l)}+1}, \ldots, x_{j+\beta_j^{(u)}-1}, x_{j+\beta_j^{(u)}}) \\ &= \sum_{j \in J_i} \frac{\partial}{\partial x_i} \phi_j(x_{j-\beta_j^{(l)}}, x_{j-\beta_j^{(l)}+1}, \ldots, x_{j+\beta_j^{(u)}-1}, x_{j+\beta_j^{(u)}}), \quad i = 1, 2, \ldots, n \end{aligned}$$

$$\text{where } J_i = \{j : j - \beta_j^{(l)} \leq i \leq j + \beta_u^{(j)}\}, \quad i = 1, 2, \ldots, n$$

J_i is the sum of all element functions containing x_i.
Then an element i in the gradient is dependent of

$$x_{k_l}, x_{k_l+1}, \ldots, x_i, x_{i+1}, \ldots, x_{k_u-1}, x_{k_u}$$

where

$$k_l = \min_{j \in J_i} \{j - \beta_j^{(l)}\}$$

and

$$k_u = \max_{j \in J_i} \{j + \beta_j^{(u)}\}.$$

We have that $f_i = k_l$ and further show that f_i is monotonically increasing.
Let j' be the element function

$$j' \in J_{i+1}$$

and

$$f_{i+1} = k_l^{(i+1)} = j' - \beta_{j'}^{(l)} > 0.$$

Suppose that $f_{i+1} < f_i$ we will show that this is not possible. Since $f_i \leq i$ we have

$$j' - \beta_{j'}^{(l)} < i$$

then $j' \in J_i$, but

$$f_i = \min\{j - \beta_j^{(l)}\} \leq j' - \beta_{j'}^{(l)} = f_{i+1}.$$

This violates the assumption that $f_{i+1} < f_i$ hence $f_{i+1} \geq f_i$, thus f_i is monotonically increasing.

Since the union of closed intervals with a common element is also a closed interval then the sequence $[k_l,\, k_u]$ is closed interval, Thus we have a skyline matrix where each row of H contains the elements of the union of all ϕ_j that contains x_i. i.e. $C_i = \{f_i, \ldots, i\}$. □

5.3.2 A Banded Matrix

In a band matrix β is constant. Which implies that f_i is strictly monotonically increasing, that is $f_i < f_{i+1}$, $i = \beta_i, \ldots, n - \beta$. The exception is in the first rows when $i = 1, 2, \ldots, \beta$, then $f_i = f_{i+1}$.

Corollary 1. *If the function $f = \sum_j \phi_j$ is partially separable on the form*

$$f(x) = \sum_{j=1+\beta^{(l)}}^{n-\beta^{(u)}} \phi_j(x_{j-\beta^{(l)}}, x_{j-\beta^{(l)}+1}, \ldots, x_{j+\beta^{(u)}-1}, x_{j+\beta^{(u)}})$$

where $\beta = \beta^{(l)} + \beta^{(u)}$ is constant. Then the Hessian is a band matrix with (half) bandwidth β.

Proof. Then

$$J_i = \begin{cases} \{j : i - \beta^{(l)} \le j \le i + \beta^{(u)}\} & \text{if } i = 1 + \beta^{(l)}, \dots, n - \beta^{(u)} \\ \{j : 1 \le j \le i + \beta^{(u)}\} & \text{if } i = 1, 2, \dots, \beta^{(l)} \\ \{j : i - \beta^{(l)} \le j \le n\} & \text{if } i = n, n-1, \dots, n - \beta^{(u)} + 1 \end{cases}$$

such that an element i in the gradient is dependent of

$$x_{k_l}, x_{k_l+1}, \dots, x_i, x_{i+1}, \dots, x_{k_u-1}, x_{k_u}$$

where

$$k_l = \min_{j \in J_i} \{j - \beta^{(l)}\} = i - \beta^{(l)} - \beta^{(u)}$$

and

$$k_u = \max_{j \in J_i} \{j + \beta^{(u)}\} = i + \beta^{(l)} + \beta^{(u)}.$$

Thus $f_i = k_l$ and $f_i < f_{i+1}$. Except for $i = 1, 2, \dots, \beta_i$ then $k_l = 1$ and $f_i = f_{i+1}$. □

6 Data Structures for Sparse Tensors

In this section we briefly discuss data structures for induced sparse super-symmetric tensors. The sparsity structure of the tensor is decided by the sparsity structure of the row i and row j of the Hessian matrix, this from (27). Thus we have the intersection of two index sets. Further we define the tube (i, j) to be

$$\mathcal{T}_{ijk}, \quad k = 1, 2, \dots, j$$

$C_i \bigcap C_j$ is the nonzero index structure of (i, j) of the tensor, where C_i and C_j is defined as in (28).

The number of stored nonzero in the tensor is thus

$$nnz(\mathcal{T}) = \sum_{i=1}^{n} \sum_{j \in C_i} |C_i \bigcap C_j|.$$

6.1 A General Sparse Tensor

Creating a data structure for a general sparse super-symmetric tensor we must compute and store the whole index structure *á* priori any computation with the tensor. Then we have at least $2nnz(\mathcal{T})$ memory requirements. Since the structure of the tensor is known by the Hessian matrix it should be exploited to save memory. This can be done by performing the computation $C_i \bigcap C_j$ before each numerical computation with tube (i, j). Thus we only need to store the size of $|C_i \bigcap C_j|$. This is less memory, but the computational cost will increase. This is illustrated in Algorithm 5 computing $g \leftarrow g + (p\mathcal{T})p$.

6.1.1 An ANSI C Implementation with *Á* Priori Storage

In this section we present an ANSI C implementation of the operation $g \leftarrow g + (p\mathcal{T})p$ with *á* priori storage 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 (note that the numerical values are randomly generated for both the matrix H, and its induced tensor and the vector p),

$$
H = \begin{pmatrix}
1 & & & & & \\
1 & 2 & & & & \\
 & & 3 & & & \\
 & 1 & & 4 & & \\
 & & 2 & & 5 & \\
1 & & 1 & & 1 & 6
\end{pmatrix}
$$

Figure 9: Stored elements of a symmetric general matrix.

Then we have the following Symmetric Compressed Row Storage (CRS) in ANSI C of the H matrix:

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

Then we have the following Super-Symmetric Compressed Tube Storage (CTS) in ANSI C of the induced tensor of the matrix H:

```
double valueT[] = {1,2,2,2,3,4,4,4,5,5,5,6,6,6,6,6,6,6,6};
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};
```

Finally, we have the implementation of the operation $g \leftarrow g + (p\mathcal{T})p$ using the above data structures:

```
int N = 6;
double p[] = {1,1,1,1,1,1};
double g[] = {0,0,0,0,0,0};
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];
    stop  = pointerH[i+1]-1;
    j = indexH[start];
    pi = p[i];
    kpi = 2*pi;
    pipi = pi*pi;
    for(;j<i;start++,ind++,tp++){
        j = indexH[start];
        pj = p[j];
        pipj = pi*pj;
        kpipj = 2*pipj;
        kpj = 2*pj;
        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
            k = indexT[ind];
            Tijk = valueT[ind];
            pk = p[k];
            pkTijk = pk*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;
        j = indexH[start+1];
    }
    startTubek = pointerT[tp];
    stopTubek  = pointerT[tp+1]-1;
    for(;startTubek<stopTubek;startTubek++,ind++){
        //Handle the case when two indices are equal: j=i
        k = indexT[ind];
        Tiik = valueT[ind];
        pk = p[k];
        ga += pk*Tiik;
        g[k] += pipi*Tiik;
    }
    //Handle the case when all three indices are equal
    g[i] += (kpi*ga + pipi*valueT[ind]);
    ga = 0.0;
}
```

We have used a linear arrays to store both the Hessian matrix and its induced tensor. If flexibility is an issue jagged arrays can perform just as well as linear arrays considering efficiency [11, 17]. This would apply for the languages C, C# and Java [12].

6.2 Banded and Skyline Tensors

It is more straightforward creating data structures for banded or skyline tensors since we only need the index structure of the Hessian matrix without any loss in performance.

For a skyline we have that

$$C_i = \{f_i \leq j \leq i\}$$

and

$$C_i \bigcap C_j = \{\max\{f_i, f_j\} \leq k \leq j\},\ \ j \leq i.$$

Then a band tensor has the start index $k = f_j, \ldots, j$ for tube (i, j).

All the elements from the start index k to j are nonzero elements for a banded tensor.

Further a skyline tensor has the start index $k = \max\{f_i, f_j\}, \ldots, j$ for tube (i, j). All the elements from the start index k to j are nonzero elements for a skyline tensor. This is illustrated in Algorithm 6 computing $H \leftarrow H + (p\mathcal{T})$ for a skyline tensor.

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

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be a super-symmetric tensor.
Let $p, g \in \mathbb{R}^n$.
Let $s \in \mathbb{R}$.
C_i is the nonzero index pattern of row i of the matrix H.
C_j is the nonzero index structure of row j of the Hessian matrix.
for $i = 1$ to n **do**
  $s = 0$
  for $j \in C_i \wedge j < i$ **do**
    for $k \in C_i \bigcap C_j \wedge k < j$ **do**
      $s+ = p_k \mathcal{T}_{ijk}$
      $g_k+ = 2p_i p_j \mathcal{T}_{ijk}$
    end for
    $g_i+ = p_j(2s + p_j \mathcal{T}_{ijj})$
    $g_j+ = 2p_i(s + p_j \mathcal{T}_{ijj})$
  end for
  $s = 0$
  for $k \in C_i \wedge k < i$ **do**
    $s+ = p_k \mathcal{T}_{iik}$
    $g_k+ = p_i^2 \mathcal{T}_{iik}$
  end for
  $g_i+ = (2p_i s + p_i^2 \mathcal{T}_{iii})$
end for

Figure 10: Algorithm 5 requires $4nnz(\mathcal{T}) + 8nnz(H) - 6n$ number of arithmetic operations.

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

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be a super-symmetric tensor.
Let $H \in \mathbb{R}^{n\times n}$ be a symmetric matrix.
Let $p \in \mathbb{R}^n$.
for $i = 1$ to n **do**
  for $j = f_i$ to $i - 1$ **do**
    for $k = \max\{f_i, f_j\}$ 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 $k = f_i$ 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

Figure 11: Algorithm 6 requires $6nnz(\mathcal{T}) - 4nnz(H)$ number of arithmetic operations.

7 The Efficiency Ratio Indicator

A measure of the complexity of working with the third derivative (tensor) compared to the second derivative (matrix) we will use the ratio of the number of nonzero elements in the tensor and number of nonzero elements in Hessian matrix.

Cubic and quadratic value term ratio is

$$\frac{flops(p^T(p\mathcal{T})p)}{flops(p^THp)} = \frac{2nnz(\mathcal{T}) + 5nnz(H) - n}{2nnz(H) + 3n} = \frac{nnz(\mathcal{T})}{nnz(H)} + 2.5 + O(\frac{1}{n}),$$

cubic and quadratic gradient term ratio is

$$\frac{flops((p\mathcal{T})p)}{flops(Hp)} = \frac{4nnz(\mathcal{T}) + 8nnz(H) - 6n}{4nnz(H) - n} = \frac{nnz(\mathcal{T})}{nnz(H)} + 2 + O(\frac{1}{n})$$

where

$$\lim_{n\to\infty} O(\frac{1}{n}) \to 0.$$

The nonzero elements that we store of the banded Hessian matrix is

$$nnz(H) = \sum_{i=0}^{\beta}(n-i) = (\beta+1)(n-\frac{\beta}{2}).$$

The number of nonzero elements in a tensor is further derived from the uniform banded Hessian matrix with band width β and where n is the dimension of the Hessian matrix.

$$nnz(\mathcal{T}) = (\sum_{i=0}^{\beta}(\beta-i+1)) \times (n-\beta) + \sum_{i=0}^{\beta}\frac{1}{2}(i+1)i = \frac{1}{2}(\beta+1)(\beta+2)(n-\frac{2}{3}\beta).$$

For a uniform band matrix we have the ratio of number of nonzero elements of the tensor and the Hessian matrix is

$$nnz(\mathcal{T})/nnz(H) = \frac{1}{2}(\beta+1)(\beta+2)(n-\frac{2}{3}\beta)/(\beta+1)(n-\frac{\beta}{2}) = \frac{1}{2}(\beta+2)(n-\frac{2}{3}\beta)/(n-\frac{\beta}{2}) \le \frac{\beta+2}{2}.$$

Then the ratio of number of nonzero elements of the tensor and the band Hessian matrix is

$$\frac{\beta+2}{2}$$

where β and the number of nonzero elements of the tensor is of the same order as for the band Hessian matrix. Taking into account symmetry this ratio is $\frac{n+2}{3}$ for a dense matrix and tensor.

7.1 The Cost of the Halley Class versus the Newton Cost

For each iteration of Newton's method and one iteration of the Halley class there is a difference in cost. We will for each of these methods outline a computational cost and compare them to each other. The computational cost is for a banded Hessian matrix, thus the bandwidth β is fixed.

Since for a small β the square root is a significant amount of the computation, we use an LDL^T factorization. Further we solve the system $LDL^Tx = b$ with $Ly = b$, $Dz = y$, and $L^Tx = z$ [9].

The computational effort of one step of Newtons method is

1. evaluation of $f(x_k)$, $\nabla f(x_k)$, $\nabla^2 f(x_k)$
2. factorization $LDL^T = \nabla^2 f(x_k)$ requires $n\beta^2 + 4n\beta$ flops
3. solution of $LDL^T s_k^{(1)} = -\nabla f(x_k)$ requires $4n\beta + n$ flops
4. updating the solution $x_{k+1} = x_k + s_k^{(1)}$ requires n flops

The total computational effort for one Newton step without the cost of function, gradient and Hessian evaluations is $n(\beta^2 + 8\beta + 2)$ flops.

The computational effort of one step of Chebyshev's method is

1. evaluation of $f(x_k)$, $\nabla f(x_k)$, $\nabla^2 f(x_k)$, and $\nabla^3 f(x_k)$
2. factorization $LDL^T = \nabla^2 f(x_k)$ requires $n\beta^2 + 4n\beta$ flops
3. solution of $LDL^T s_k^{(1)} = -\nabla f(x_k)$ requires $4n\beta + n$ flops
4. evaluation of $(p\mathcal{T})p$ requires $2(\beta+1)(\beta+2)(n - \frac{2}{3}\beta) + 8(\beta+1)(n - \frac{\beta}{2}) - 6n$ flops
5. solution of $LDL^T s_k^{(2)} = -\frac{1}{2}(p\mathcal{T})p$ requires $4n\beta + n$ flops
6. updating the solution $x_{k+1} = x_k + s_k^{(1)} + s_k^{(2)}$ requires $2n$ flops

The total computational effort for one Chebyshev step without the cost of function, gradient, Hessian and tensor evaluations is $3n\beta^2 - \frac{4}{3}\beta^3 + 26n\beta - 8\beta^2 - \frac{20}{3}\beta + 10n$ flops.

The computational effort of one step of Halley's method is

1. evaluation of $f(x_k)$, $\nabla f(x_k)$, $\nabla^2 f(x_k)$, and $\nabla^3 f(x_k)$
2. factorization $LDL^T = \nabla^2 f(x_k)$ requires $n\beta^2 + 4n\beta$ flops
3. solution of $LDL^T s_k^{(1)} = -\nabla f(x_k)$ requires $4n\beta + n$ flops
4. evaluation of $(p\mathcal{T})$ requires $3(\beta+1)(\beta+2)(n - \frac{2}{3}\beta) - 4(\beta+1)(n - \frac{\beta}{2})$ flops
5. evaluation of $(p\mathcal{T})p$ (matrix-vector product) requires $4n\beta + 2n$ flops
6. evaluation of $\nabla^2 f(x_k) + \frac{1}{2}(p\mathcal{T})$ requires $2n\beta + 2n$ flops
7. factorization $LDL^T = \nabla^2 f(x_k) + \frac{1}{2}(p\mathcal{T})$ requires $n\beta^2 + 4n\beta$ flops
8. solution of $LDL^T s_k^{(2)} = -(p\mathcal{T})p$ requires $4n\beta + n$ flops
9. updating the solution $x_{k+1} = x_k + s_k^{(1)} + s_k^{(2)}$ requires $2n$ flops

The total computational effort for one Halley step without the cost of function, gradient, Hessian and tensor evaluations is $5n\beta^2 - 2\beta^3 + 27n\beta - 4\beta^2 - 2\beta + 10n$ flops.

The computational effort of one step of Super Halley's method is

1. evaluation of $f(x_k)$, $\nabla f(x_k)$, $\nabla^2 f(x_k)$, and $\nabla^3 f(x_k)$
2. factorization $LDL^T = \nabla^2 f(x_k)$ requires $n\beta^2 + 4n\beta$ flops

3. solution of $LDL^T s_k^{(1)} = -\nabla f(x_k)$ requires $4n\beta + n$ flops
4. evaluation of $(p\mathcal{T})$ requires $3(\beta+1)(\beta+2)(n - \frac{2}{3}\beta) - 4(\beta+1)(n - \frac{\beta}{2})$ flops
5. evaluation of $(p\mathcal{T})p$ (matrix-vector product) requires $4n\beta + 2n$ flops
6. evaluation of $\nabla^2 f(x_k) + (p\mathcal{T})$ requires $n\beta + n$ flops
7. factorization $LDL^T = \nabla^2 f(x_k) + (p\mathcal{T})$ $n\beta^2 + 4n\beta$ flops
8. solution of $LDL^T s_k^{(2)} = -(p\mathcal{T})p$ requires $4n\beta + n$ flops
9. updating the solution $x_{k+1} = x_k + s_k^{(1)} + s_k^{(2)}$ requires $2n$ flops

The total computational effort for one Super Halley step without the cost of function, gradient, Hessian and tensor evaluations is $5n\beta^2 - 2\beta^3 + 26n\beta - 4\beta^2 - 2\beta + 9n$ flops.

It is important to notice that we have not taken into account the evaluation of the test function i.e function value, gradient, Hessian and/or tensor since they are problem dependent. This unknown factor might have a crucial impact on the total running time of the methods.

7.1.1 The Efficiency Ratio Indicator for the Halley Class

The ratio indicators below shows how much more the cost of using third order method's compared to a Newton's method for one iteration.

The Chebyshev to Newton cost is

$$\frac{flops(Chebyshev)}{flops(Newton)} = \frac{3n\beta^2 - \frac{4}{3}\beta^3 + 26n\beta - 8\beta^2 - \frac{20}{3}\beta + 10n}{n(\beta^2 + 8\beta + 2)} = \frac{3\beta^2 + 26\beta + 10}{\beta^2 + 8\beta + 2} + O(\frac{\beta}{n}),$$

the Halley to Newton cost is

$$\frac{flops(Halley)}{flops(Newton)} = \frac{5n\beta^2 - 2\beta^3 + 27n\beta - 4\beta^2 - 2\beta + 10n}{n(\beta^2 + 8\beta + 2)} = \frac{5\beta^2 + 27\beta + 10}{\beta^2 + 8\beta + 2} + O(\frac{\beta}{n}),$$

the Super Halley to Newton cost is

$$\frac{flops(SuperHalley)}{flops(Newton)} = \frac{5n\beta^2 - 2\beta^3 + 26n\beta - 4\beta^2 - 2\beta + 9n}{n(\beta^2 + 8\beta + 2)} = \frac{5\beta^2 + 26\beta + 9}{\beta^2 + 8\beta + 2} + O(\frac{\beta}{n}),$$

where

$$\lim_{n\to\infty} O(\frac{\beta}{n}) \to 0.$$

7.2 Numerical Results

In this section we present numerical results where the theoretical flops count for the operations is compared to the ratio $\frac{nnz(\mathcal{T})}{nnz(H)}$, and we have the measured CPU times for the operations compared to the ratio $\frac{nnz(\mathcal{T})}{nnz(H)}$.

Comparing the ratio $\frac{nnz(\mathcal{T})}{nnz(H)}$ for all of the test cases shows that the ratio indicator has a linear behaviour for increasing n thus it indicates the cost from going from a quadratic to a cubic operation.

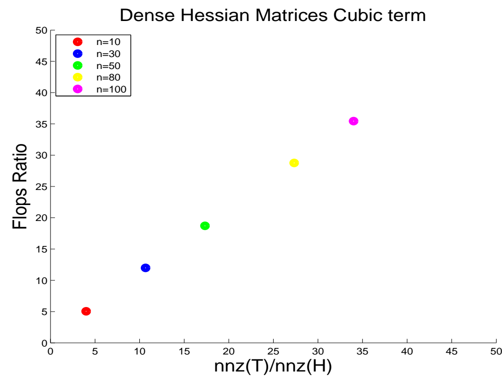

Figure 12: Dense matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the flops ratio.

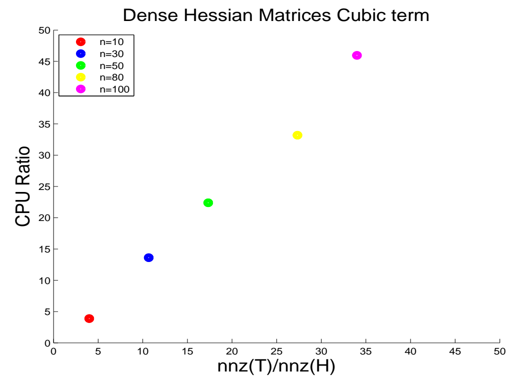

Figure 13: Dense matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the CPU ratio.

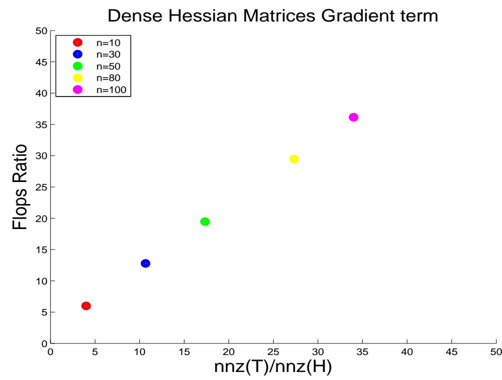

Figure 14: Dense matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the flops ratio.

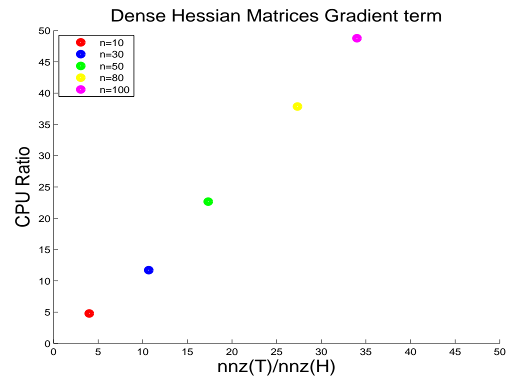

Figure 15: Dense matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the CPU ratio.

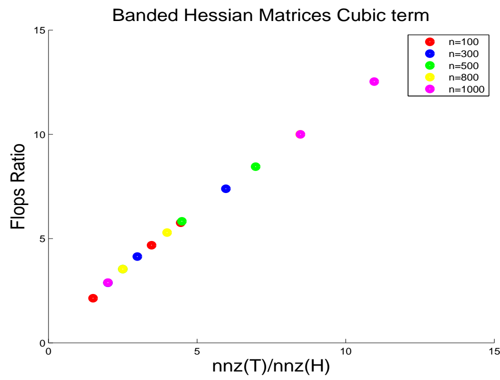

Figure 16: Banded matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the flops ratio.

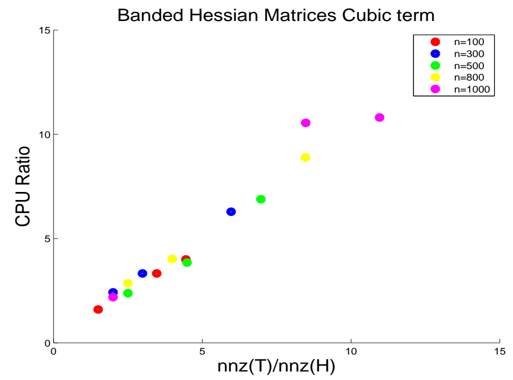

Figure 17: Banded matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the CPU ratio.

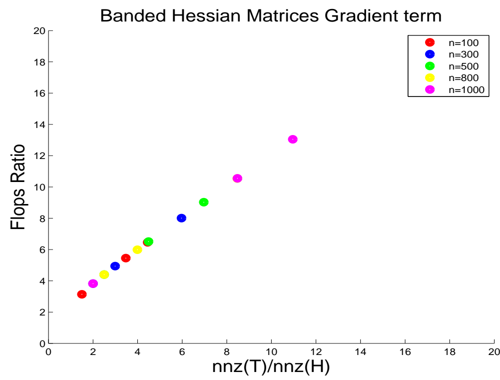

Figure 18: Banded matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the flops ratio.

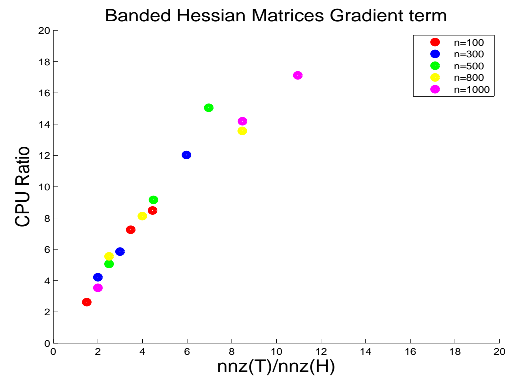

Figure 19: Banded matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the CPU ratio.

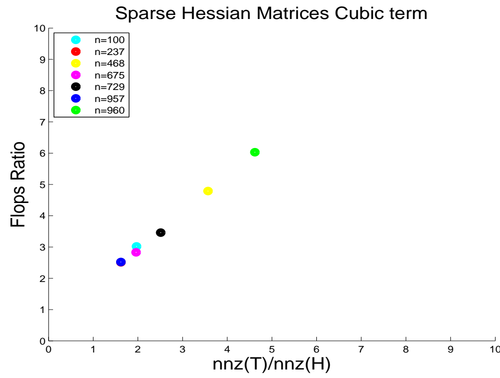

Figure 20: Sparse matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the flops ratio.

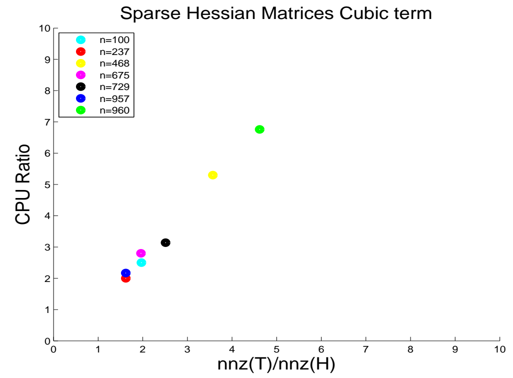

Figure 21: Sparse matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the CPU ratio.

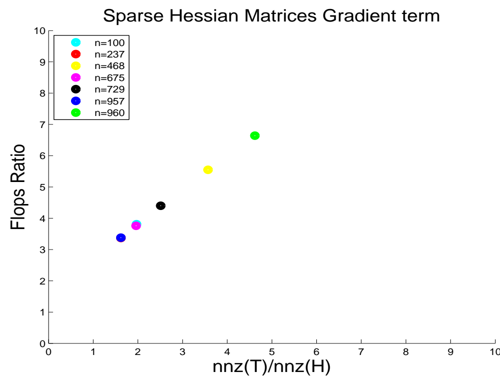

Figure 22: Sparse matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the flops ratio.

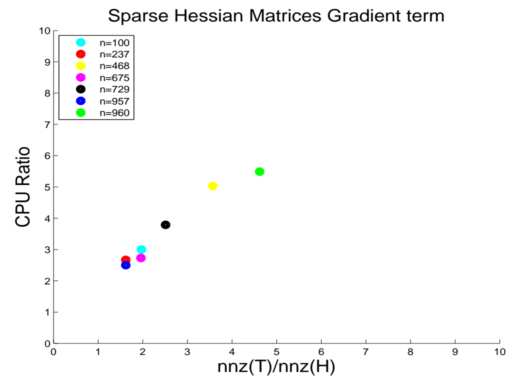

Figure 23: Sparse matrices: The ratio $nnz(\mathcal{T})/nnz(H)$ to the CPU ratio.

Conclusions and Future Work

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 trust-region algorithm with a cubic model has fewer iterations then when implemented with a quadratic model, for our test cases. Thus indicating that exact third derivatives is also useful for global methods, which is promising for further work in this area.

The concept of induced sparsity makes it possible to create data structure and algorithms for tensor operations in a straightforward manner. For banded and skyline type Hessian matrices we need not to store the index structure of the tensor, just its values. For general sparse Hessian matrices we have presented some examples of data structures for the tensor none which we considered to be satisfactorily when it comes to efficiency and memory requirements.

We have also seen that partially separable functions can reveal the structure of the Hessian matrix thus since the tensor (third derivatives) is induced by the Hessian matrix, it also reveals its structure.

Numerical results shows that the ratio $\frac{nnz(\mathcal{T})}{nnz(H)}$ is an indicator for all types of Hessian structure. The growth of the ratio $\frac{nnz(\mathcal{T})}{nnz(H)}$ for dense Hessian structure is large for increasing n. The ratio $\frac{nnz(\mathcal{T})}{nnz(H)}$ for sparse Matrix Market matrices is usually small. The Efficiency Ratio Indicator for the Halley Class for banded Hessian matrices is independent of n.

A Computing the Cubic Model

The Cubic gradient term: $(p\mathcal{T})p$

Algorithm 7 Computing $g^{(1)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(1)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = 1$ to j **do**
      $g_i^{(1)} += p_j p_k \mathcal{T}_{ijk}$
    end for
  end for
end for

Figure 24: An implementation of (15).

Algorithm 8 Computing $g^{(2)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(2)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = j+1$ to i **do**
      $g_i^{(2)} += p_j p_k \mathcal{T}_{ikj}$
    end for
  end for
end for

Figure 25: An implementation of (16) for $\mathcal{T}_{ikj}$.

Algorithm 9 Computing $g^{(2)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(2)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = 1$ to $j-1$ **do**
      $g_i^{(2)} += p_j p_k \mathcal{T}_{ijk}$
    end for
  end for
end for

Figure 26: An $1 \le k \le j \le i \le n$ approach of (16) for $\mathcal{T}_{ijk}$.

Algorithm 10 Computing $g^{(3)}$.

```
Let 𝒯 ∈ ℝ^{n×n×n} be super-symmetric.
Let g^(3) ∈ ℝ^n be a vector.
Let p ∈ ℝ^n be a vector.
for i = 1 to n do
  for j = 1 to i do
    for k = i + 1 to n do
      g_i^(3) += p_j p_k 𝒯_kij
    end for
  end for
end for
```

Figure 27: An implementation of (17) for $\mathcal{T}_{kij}$.

Algorithm 11 Computing $g^{(3)}$.

```
Let 𝒯 ∈ ℝ^{n×n×n} be super-symmetric.
Let g^(3) ∈ ℝ^n be a vector.
Let p ∈ ℝ^n be a vector.
for i = 1 to n do
  for j = 1 to i − 1 do
    for k = 1 to j do
      g_j^(3) += p_i p_k 𝒯_ijk
    end for
  end for
end for
```

Figure 28: An $1 \leq k \leq j \leq i \leq n$ approach of (17) for $\mathcal{T}_{ijk}$.

Algorithm 12 Computing $g^{(4)}$.

```
Let 𝒯 ∈ ℝ^{n×n×n} be super-symmetric.
Let g^(4) ∈ ℝ^n be a vector.
Let p ∈ ℝ^n be a vector.
for i = 1 to n do
  for j = i + 1 to n do
    for k = 1 to i do
      g_i^(4) += p_j p_k 𝒯_jik
    end for
  end for
end for
```

Figure 29: An implementation of (18) for $\mathcal{T}_{jik}$.

Algorithm 13 Computing $g^{(4)}$.

```
Let 𝒯 ∈ ℝ^{n×n×n} be super-symmetric.
Let g^(4) ∈ ℝ^n be a vector.
Let p ∈ ℝ^n be a vector.
for i = 1 to n do
  for j = 1 to i − 1 do
    for k = 1 to j do
      g_j^(4) += p_i p_k 𝒯_ijk
    end for
  end for
end for
```

Figure 30: An $1 \leq k \leq j \leq i \leq n$ approach of (18) for $\mathcal{T}_{ijk}$.

Algorithm 14 Computing $g^{(5)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(5)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.

```
for i = 1 to n do
  for j = i+1 to n do
    for k = i+1 to j do
      g_i^(5) += p_j p_k T_jki
    end for
  end for
end for
```

Figure 31: An implementation of (19) for $\mathcal{T}_{jki}$.

Algorithm 15 Computing $g^{(5)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(5)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.

```
for i = 1 to n do
  for j = 1 to i do
    for k = 1 to j-1 do
      g_k^(5) += p_i p_j T_ijk
    end for
  end for
end for
```

Figure 32: An $1 \leq k \leq j \leq i \leq n$ approach (19) for $\mathcal{T}_{ijk}$.

Algorithm 16 Computing $g^{(6)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(6)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.

```
for i = 1 to n do
  for j = i+1 to n do
    for k = j+1 to n do
      g_i^(6) += p_j p_k T_kji
    end for
  end for
end for
```

Figure 33: An implementation of (20) for $\mathcal{T}_{kji}$.

Algorithm 17 Computing $g^{(6)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $g^{(6)} \in \mathbb{R}^n$ be a vector.
Let $p \in \mathbb{R}^n$ be a vector.

```
for i = 1 to n do
  for j = 1 to i-1 do
    for k = 1 to j-1 do
      g_k^(6) += p_i p_j T_ijk
    end for
  end for
end for
```

Figure 34: An $1 \leq k \leq j \leq i \leq n$ approach of (20) for $\mathcal{T}_{ijk}$.

Note that Algorithm 8 and 9 are equivalent, Algorithm 10 and 11 equivalent, Algorithm 12 and 13 equivalent, Algorithm 14 and 15 equivalent, and Algorithm 16 and 17 equivalent in the respect that they produce the same final gradient, but the intermediate entries in the gradient are not the same.

The Cubic Hessian term: $(p\mathcal{T})$

Algorithm 18 Computing $H^{(1)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $H^{(1)} \in \mathbb{R}^{n\times n}$ be symmetric.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = 1$ to j **do**
      $H^{(1)}_{ij} += p_k \mathcal{T}_{ijk}$
    end for
  end for
end for

Figure 35: An implementation of (24).

Algorithm 19 Computing $H^{(1)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $H^{(1)} \in \mathbb{R}^{n\times n}$ be a symmetric.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = 1$ to j **do**
      $H^{(1)}_{ij} += p_k \mathcal{T}_{ijk}$
    end for
  end for
end for

Figure 36: An $1 \leq k \leq j \leq i \leq n$ approach of (24).

Algorithm 20 Computing $H^{(2)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $H^{(2)} \in \mathbb{R}^{n\times n}$ be a symmetric.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = j+1$ to i **do**
      $H^{(2)}_{ij} += p_k \mathcal{T}_{ikj}$
    end for
  end for
end for

Figure 37: An implementation of (25).

Algorithm 21 Computing $H^{(2)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $H^{(2)} \in \mathbb{R}^{n\times n}$ be a symmetric.
Let $p \in \mathbb{R}^n$ be a vector.
for $i = 1$ to n **do**
  for $j = 1$ to i **do**
    for $k = 1$ to $j-1$ **do**
      $H^{(2)}_{ik} += p_j \mathcal{T}_{ijk}$
    end for
  end for
end for

Figure 38: An $1 \leq k \leq j \leq i \leq n$ approach of (25).

Algorithm 22 Computing $H^{(3)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $H^{(3)} \in \mathbb{R}^{n\times n}$ be a symmetric.
Let $p \in \mathbb{R}^n$ be a vector.

```
for i = 1 to n do
  for j = 1 to i do
    for k = i + 1 to n do
      H(3)_ij += p_k T_kij
    end for
  end for
end for
```

Figure 39: An implementation of (26).

Algorithm 23 Computing $H^{(3)}$.

Let $\mathcal{T} \in \mathbb{R}^{n\times n\times n}$ be super-symmetric.
Let $H^{(3)} \in \mathbb{R}^{n\times n}$ be a symmetric.
Let $p \in \mathbb{R}^n$ be a vector.

```
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
```

Figure 40: An $1 \leq k \leq j \leq i \leq n$ approach of (26).

Note that Algorithm 18 and 19 are equivalent, Algorithm 20 and 21 are equivalent, and Algorithm 22 and 23 equivalent in the respect that they produce the same final matrix, but the intermediate entries in the matrices are not the same.

References

[1] M. Altman. *Iterative Methods of Higher Order*. Bulletin De LÁcademie Polonaise Des Sciences Serie des sciences math., astr. et phys.-Vol. IX, No. 2, 1961.

[2] B. W. Bader and T. G. Kolda. *MATLAB Tensor Classes for Fast Algorithm Prototyping*. Technical Report SAND 2004-5187, October 2004.

[3] E.M.L. Beale. *On an iterative method of finding a local minimum of a function of more than one variable.* Tech. Rep. No. 25, Statistical Techniques Research Group, Princeton Univ., Princeton, N.J., 1958.

[4] A. M. Cuyt. *Numerical Stability of the Halley-Iteration for the Solution of a System of Nonlinear Equations*. Mathematics of Computation. Volume 38, Number 157. January, 1982.

[5] A. M. Cuyt. *Computational Implementation of the Multivariate Halley Method for Solving Nonlinear Systems of Equations*. ACM Transactions on Mathematical Software, Vol. 11, No. 1, March 1985, Pages 20-36.

[6] N. Deng and H. Zhang. *Theoretical Efficiency of a New Inexact Method of Tangent Hyperbolas.* Optimization Methods and Software. Vol. 19, Nos. 3-4, June-August 2004, pp.247-265.

[7] J. A. Ezquerro and M. A. Hernandez. *A New Class of Third-Order Methods in Banach Spaces.* Bulletin of the Institute of Mathematics Academia Sinica. Volume 31, Number 1, March 2003.

[8] W. Gander. *On Halley's Iteration Method*. American Mathematical Monthly, Vol. 92, No. 2. (Feb., 1985), pp. 131-134.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[10] A. Griewank and Ph. L. Toint. *On the unconstrained optimization of partially separable functions.* In Michael J. D. Powell, editor, Nonlinear Optimization 1981, pages 301-312. Academic Press, New York, NY, 1982.

[11] G. Gundersen and T. Steihaug. *Data Structures In Java for Matrix Computation.* Concurrency and Computation: Practice and Experience, 16(8):735815, July 2004.

[12] G. Gundersen and T. Steihaug. *On the Effciency of Arrays in C, C# and Java.* In C. Rong, editor, Proc. of the Norwegian Informatics Conference (NIK'04), pages 1930. Tapir Academic Publisher, Nov. 2004.

[13] J. M. Gutierrez and M. A. Hernandez. *An acceleration of Newton's method: Super-Halley method.* Applied Mathematics and Computation. 25 January 2001, vol. 117, no. 2, pp. 223-239(17).

[14] D. Han. *The Convergence on a Family of Iterations with Cubic Order.* Journal of Computational Mathematics, Vol.19, No.5, 2001, 467-474.

[15] R. H. F. Jackson and G. P. McCormick. *The Polyadic Structure of Factorable Function Tensors with Applications to Higher-Order Minimization Techniques.* Journal of Optimization Theory and Applications: Vol. 51, No. 1, October 1986.

[16] R. Kalaba and A. Tishler. *A Generalized Newton Algorithm Using Higher-Order Derivatives.* Journal of Optimization Theory and Applications: Vol. 39, No. 1, January 1983.

[17] M. Luján and A. Usman and P. Hardie and T. L. Freeman and J. R. Gurd. *Storage formats for sparse matrices in Java.* In Proceedings of the 5th International Conference on Computational Science – ICCS 2005, Part I, Part I, volume 3514 of Lecture Notes in Computer Science, pages 364-371. Springer-Verlag, 2005.

[18] J.J. More, B.S. Garbow and K.E. Hillstrom. *Testing unconstrained optimization software.* ACM Transactions on Mathematical Software, 7, 17–41, 1981.

[19] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Series in Operations Research. Springer-Verlag, 1999.

[20] J. M. Ortega and W. C. Rheinboldt. *Iterative solution of nonlinear equations in several variables.* New York, Academic Press, 1970.

[21] 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.

[22] 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.

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

[24] P. Sebah and X. Gourdon. *Newton's method and higher order iterations.* numbers.computation.free.fr/Constants/constant.html

[25] J. F. Traub. *Iterative Methods for the Solution of Equations.* Prentice-Hall, Englewood Cliffs, NJ, 1964.

[26] Ph.L Toint. *Some numerical results using a sparse matrix updating formula in unconstrained optimization.* Mathematics of Computation, Volume 32, Number 143. July 1978, pages 839-851.

[27] W. Werner. *Some improvement of classical iterative methods for the solution of nonlinear equations.* In: E.L. Allgower, K. Glashoff and N.-O. Peitgen, Eds., Proc. Numerical Solution of Nonlinear Equations (Springer, Bremen, 1980) 426–440.

[28] W. Werner. *Iterative Solution of Systems of Nonlinear Equations based upon Quadratic Approximations.* Comp and Math, Appls. Vol. 12A. No. 3. pp. 331-343. 1986.

[29] T. Yamamoto. *Historical developments in convergence analysis for Newton's and Newton-like methods.* Journal of Computational and Applied Mathematics 124 (2000) 1-23.

[30] S. Zheng and D. Robbie. *A Note on the Convergence of Halley's Method for Solving Operator Equations.* J. Austral. Math. Soc. Ser. B 37(1995),16-25.