Journal of Computational and Applied Mathematics 263 (2014) 1–13

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

A recursive algorithm for optimizing differentiation

Ali Mashreghi[a,*], Hadi Sadoghi Yazdi[a,b]

[a] Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran

[b] Center of Excellence on Soft Computing and Intelligent Information Processing, Ferdowsi University of Mashhad, Mashhad, Iran

ARTICLE INFO

Article history:
Received 21 November 2013

Keywords:
Differentiation
Recursive algorithm
Optimization
Degree of derivative
Order of accuracy

ABSTRACT

In this paper a recursive algorithm will be introduced to improve the differentiation method proposed by Hasan et al. (2012). Their algorithm is based on the undetermined coefficient method and uses the Taylor series expansion and Vandermonde matrix inverse to calculate weighting coefficients with a complexity of $O(n^4)$. Our method reduces time complexity from $O(n^4)$ to $O(n^3)$. Moreover, we obtain a substantial optimality when the required degree and the order of accuracy increase. Finally, the implementation of the proposed method follows, and experimentations prove the validity of the algorithm and compare previous and new methods in terms of execution time. Besides, the combination of our optimization and parallel processing provides valuable results in real world applications. Particularly, QRS detection in ECG signal processing has been considered, and we have shown that how features of our method could be used in order to increase accuracy and speed for this application.

Contents

* Corresponding author. Tel.: +98 9361344489.
E-mail addresses: ali.mashreghi87@gmail.com (A. Mashreghi), h_sadoghi@um.ac.ir (H.S. Yazdi).

0377-0427/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cam.2013.11.022

1. Introduction

Differentiation generally is referred to the operations we use to calculate the derivative of a function. Derivative expresses the quality and quantity of change in a function with respect to its arguments. This property of a function becomes so valuable while we focus on analyzing and extracting various features from a dataset. Thus, variant methods have been proposed to design differentiator filters for a variety of purposes [1–8]. However, sometimes calculating derivatives is too expensive in terms of speed and complexity. For this reason many algorithms have been developed to do this task faster and simpler. Numerical differentiation is one of these methods which is extensively in use to approximate derivatives.

There are two main approaches for differentiating numerically [9]. The first approach attempts to calculate the derivative through developing closed form formulas, such as techniques which approximate derivatives using the Taylor series [9,1,2,7,10], while the second approach calculates the derivative using the function data. In particular, regarding the methods that use the Taylor series to calculate the derivative at a specific point, the process generally is based on forming and solving a system of linear equations using neighbor points around the desired point. Since such methods typically end up with costly operations of calculating the inverse matrix, they are not usually applicable when the number of equations increases. Furthermore, the order of accuracy is not similar for all points in the data sample because the number of the neighbor points is different for the points that are close to the center compared to those that are farther.

Thus, Hasan et al. [9] introduced a method which resolves the previously mentioned problems by developing the undetermined coefficients in terms of the Vandermonde matrix inverse. Using elementary symmetric functions as the main part of calculating undetermined coefficients, they took advantage of the closed form of the inverse of the Vandermonde matrix. Consequently, the burden of calculation for obtaining the inverse matrix is noticeably reduced and derivatives of higher degree can be calculated more easily. Moreover, they proposed a modification on the derivatives to unify the order of accuracy for all points in the data sample. Above all, their method is applicable for any required order of accuracy and degree of derivative when the distances between the points are equal.

Although the mentioned algorithm is so handy and convenient, it does some unnecessary calculations and is not optimal enough when we need to obtain derivatives faster or when the required order of accuracy and degree of derivative highly increase. In a word, this paper aims to focus on avoiding unnecessary calculations and establish a recursive relation between columns of each sigma matrix (*the term is explained in* Note 2.1) to make the desired optimization.

2. Previous method

In this section we aim to restate the proposed method in [9] and show which part of their method could be improved. For simplicity, we use the same notations as [9].

2.1. Notation and problem statement

Consider n distinct real points x_1 through x_n where $x_1 < x_2 < \cdots < x_n$ as shown in Fig. 1. The points are equally spaced which means $x_{i+1} - x_i = h$ (for $i = 1, \ldots, n-1$) and h is a constant called the sampling period. It is easy to see that the relation $x_i = x_l + (i-l) \cdot h$ holds. Given f (a differentiable and continuous function over the interval $[x_1, x_n]$), m (degree of derivative), O (order of accuracy) and h (sampling period), we want to calculate 'mth derivative of f with the order of accuracy O for every x_i ($i = 1, 2, \ldots, n$)'. This value for a specific point i is denoted by $f_i(m, O)$. Furthermore the notation f_i is a substitution of $f(x_i)$. The term *order of accuracy* somehow determines accuracy in the computations. More specifically, since the Taylor series expansion is infinite we have to ignore the remaining terms at some point. It is easy to see that using the Taylor series f_l could be obtained from the following formula:

$$f_l = \frac{f_i'}{1!}\Delta x + \frac{f_i''}{2!}(\Delta x)^2 + \cdots + \frac{f_i^{(k-1)}}{k!}(\Delta x)^{k-1} + O((\Delta x)^k), \quad \text{where } \Delta x = x_l - x_i.$$

Informally speaking, when we want to calculate $f_i^{(m)}(m < k)$, $f_i^{(m)}$ is moved to the left hand side of the equation and the error term $O((\Delta x)^k)$ is divided by $O((\Delta x)^m)$. Thus, the overall error term for obtaining $f_i^{(m)}$ is of order $O((\Delta x)^{k-m})$ and '*order of accuracy is* $k - m$'. Hence, if we need to have order of accuracy of O for the mth derivative we need to ignore the terms with $k > m + O$. Therefore, the difference between $f_i^{(m)}$ and $f_i(m, O)$ is that $f_i^{(m)}$ denotes the exact value of mth derivative while the notation $f_i(m, O)$ shows the approximated value. Obviously, the more the order of accuracy increases the more the computed answer becomes precise.

In the process of calculation, determining the weighting coefficients is the most expensive part and takes $O(n^4)$ operations. The term *weighting coefficients* is referred to the undetermined coefficients which need to be calculated by solving linear equations. When calculating $f_i(m, O)$ for the point x_i, the lth weighting coefficient is denoted by $C_{i,l}(m, O)$. At the end of Section 2.2 it will be shown that what values the weighting coefficients are depending on, and how we can use this dependency to speed up our computations.

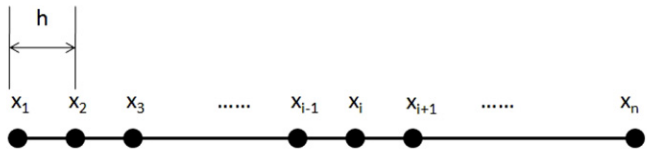

Fig. 1. The equally spaced points x_1 through x_n with a constant sampling period of h.

2.2. Derivation of the previous method

Using $C_i\,(m, O)$ as weighting coefficients, we can approximate $f_i(m, O)$ by the following equation in which $n = m + O$:

$$f_i\,(m, O) = \sum_{l=1}^{n} C_{i,l}\,(m, O) \cdot f_l, \quad i = 1, 2, \ldots, n. \tag{1}$$

We can calculate f_l in Eq. (1) by the Taylor series around the reference point x_i. The reference point is referred to the point whose neighbors are used to calculate the weighting coefficients for it. Hence, f_l becomes:

$$f_l = \sum_{k=0}^{\infty} \frac{(l-i)^k}{k!} h^k f_i^{(k)}. \tag{2}$$

By expanding sigma and substituting it into Eq. (1) we have

$$f_i\,(m, O) = \sum_{l=1}^{n} C_{i,l}\,(m, O) \left[f_i + \frac{(l-i)}{1!} h f_i' + \frac{(l-i)^2}{2!} h^2 f_i'' + \cdots + \frac{(l-i)^n}{n!} h^n f_i^{(n)} + \cdots \right]. \tag{3}$$

Considering $E_i\,(m, O)$ as the error of approximation when we ignore the terms greater than or equal to n, and distributing sigma into the series, Eq. (3) can be written in the following form:

$$\begin{aligned} f_i\,(m, O) &= f_i \left[\sum_{l=1}^{n} C_{i,l}\,(m, O) \right] + f_i' \frac{h}{1!} \left[\sum_{l=1}^{n} C_{i,l}\,(m, O) \cdot (l-i) \right] \\ &\quad + \cdots + f_i^{(n-1)} \frac{h^{(n-1)}}{(n-1)!} \left[\sum_{l=1}^{n} C_{i,l}\,(m, O) \cdot (l-i)^{(n-1)} \right] + E_i\,(m, O)\,. \end{aligned} \tag{4}$$

For simplicity, Eq. (4) can be rewritten as follows:

$$f_i\,(m, O) = \sum_{g=1}^{n} \left[f_i^{(g-1)} \frac{h^{(g-1)}}{(g-1)!} \sum_{l=1}^{n} \left(C_{i,l}\,(m, O) \cdot (l-i)^{(g-1)} \right) \right] + E_i\,(m, O)\,. \tag{5}$$

Thus, by ignoring the error term, an approximation for $f_i\,(m, O)$ is:

$$f_i\,(m, O) \approx \sum_{g=1}^{n} \left[f_i^{(g-1)} \frac{h^{(g-1)}}{(g-1)!} \sum_{l=1}^{n} \left(C_{i,l}\,(m, O) \cdot (l-i)^{(g-1)} \right) \right]. \tag{6}$$

If we consider

$$b_g = f_i^{(g-1)} \frac{h^{(g-1)}}{(g-1)!} \sum_{l=1}^{n} \left(C_{i,l}\,(m, O) \cdot (l-i)^{(g-1)} \right), \quad g = 1, 2, \ldots, n \tag{7}$$

then we have

$$b_g \frac{(g-1)!}{h^{(g-1)}} = \sum_{l=1}^{n} \left(C_{i,l}\,(m, O) \cdot (l-i)^{(g-1)} \right), \quad g = 1, 2, \ldots, n \tag{8}$$

where

$$b_g = \begin{cases} 1 & \text{if } g = m+1 \\ 0 & \text{if } g \neq m+1. \end{cases} \tag{9}$$

Accordingly

$$\sum_{l=1}^{n} \left(C_{i,l}\,(m, O) \cdot (l-i)^{(g-1)} \right) = \begin{cases} \dfrac{m!}{h^m} & \text{if } g = m+1 \\ 0 & \text{if } g \neq m+1. \end{cases} \tag{10}$$

Using a matrix representation, we get

$$\begin{bmatrix} (1-i)^0 & (2-i)^0 & \cdots & (l-i)^0 & \cdots & (n-i)^0 \\ (1-i)^1 & (2-i)^1 & \cdots & (l-i)^1 & \cdots & (n-i)^1 \\ (1-i)^2 & (2-i)^2 & \cdots & (l-i)^2 & \cdots & (n-i)^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (1-i)^{g-1} & (2-i)^{g-1} & \cdots & (l-i)^{g-1} & \cdots & (n-i)^{g-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (1-i)^{n-1} & (2-i)^{n-1} & \cdots & (l-i)^{n-1} & \cdots & (n-i)^{n-1} \end{bmatrix} \begin{bmatrix} C_{i,1}(m,O) \\ C_{i,2}(m,O) \\ C_{i,3}(m,O) \\ \vdots \\ C_{i,g}(m,O) \\ \vdots \\ C_{i,n}(m,O) \end{bmatrix} = \frac{m!}{h^m} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_g \\ \vdots \\ b_n \end{bmatrix}. \tag{11}$$

If we denote first, second and third matrix by $A_i(m, O)$, $C_i(m, O)$ and $b_i(m, O)$ respectively, then we have

$$A_i(m,O)\,C_i(m,O) = \frac{m!}{h^m} b_i(m,O). \tag{12}$$

Therefore

$$C_i(m,O) = \frac{m!}{h^m} A_i^{-1}(m,O) b_i(m,O). \tag{13}$$

$A_i(m, O)$ is the transpose of the Vandermonde matrix $V_i(m, O)$ [9]. Consequently, we can write

$$C_i(m,O) = \frac{m!}{h^m}\left[V_i^T(m,O)\right]^{-1} b_i(m,O) = \frac{m!}{h^m}\left[V_i^{-1}(m,O)\right]^T b_i(m,O). \tag{14}$$

Regarding Eq. (14), $V_i^{-1}(m, O)$ can be expressed in a closed form by elementary symmetric functions as follows [9]:

$$V_i^{-1}(m,O) = m_i(m,O) = \left[m_{l,r}\right], \quad m_{l,r} = (-1)^{n-l} \frac{\sigma^{(n)}_{n-l+1,r}}{\prod\limits_{p=1,p\neq r}^{p=n} (v_r - v_p)} \tag{15}$$

where

$$\sigma^{(n)}_{i,j} = \begin{cases} \sum\limits_{\substack{r_1=1\\ r_1\neq j}}^{r_1=n} \sum\limits_{\substack{r_2=r_1+1\\ r_2\neq j}}^{r_2=n} \sum\limits_{\substack{r_3=r_2+1\\ r_3\neq j}}^{r_3=n} \cdots \sum\limits_{\substack{r_{i-1}=r_{i-2}+1\\ r_{i-1}\neq j}}^{r_{i-1}} \prod\limits_{h=1}^{h=i-1} v_{r_h} & \text{if } i \neq 1 \\ 1 \quad \text{if } i = 1 \end{cases} \tag{16}$$

and

$$v_k = k - i, \quad k = 1, 2, 3, \ldots, n. \tag{17}$$

Note 2.1. The term *sigma matrix* denoted by $\sigma^{(n)}$, which will be frequently used by authors, is referred to an $n \times n$ matrix which contains $\sigma^{(n)}_{x,y}$ as the value of row x and column y. Also, note that a specific sigma matrix is associated with a specific vector v_k in Eq. (17). Furthermore, we say that $v_k = k - i$ creates the ith sigma matrix.

Referring Eq. (14), we have

$$\left[V_i^{-1}(m,O)\right]^T = \omega_i(m,O) = \left[\omega_{l,r}\right], \quad \omega_{l,r} = (-1)^{n-r} \frac{\sigma^{(n)}_{n-r+1,l}}{\prod\limits_{p=1,p\neq l}^{p=n} \left(v_l - v_p\right)}. \tag{18}$$

Substituting Eq. (18) into Eq. (14) we have

$$C_i(m,O) = \frac{m!}{h^m}\omega_i(m,O)\, b_i(m,O) = \frac{m!}{h^m}\left[\omega_{l,m+1}\right] \tag{19}$$

and

$$C_{i,l}(m,O) = \frac{m!}{h^m}\omega_{l,m+1} = \frac{m!}{h^m}(-1)^{n-m-1} \frac{\sigma^{(n)}_{n-m,l}}{\prod\limits_{p=1,p\neq l}^{p=n} (v_l - v_p)}. \tag{20}$$

Moreover, we can prove that:

$$\prod_{p=1,p\neq l}^{p=n} (v_l - v_p) = (-1)^{n-l}(l-1)!\,(n-l)! \tag{21}$$

Finally weighting coefficients can be expressed in a closed form as follows:

$$C_{i,l}(m,O) = (-1)^{l-m-1} \frac{m!}{h^m (l-1)!\,(n-l)!} \sigma^{(n)}_{n-m,l}. \tag{22}$$

As briefly mentioned before, determining the weighting coefficients is the most important part in calculating $f_i(m,O)$ and needs the highest number of operations. Hence, we have focused on optimizing this part of calculations. Concerning Hasan et al. [9], the weighting coefficients for $n = m+O$ points can be calculated by Eq. (22). Furthermore the main part in obtaining coefficients is computing sigma matrices. Indeed, their proposed method calculates every column of the sigma matrix independently in $O(n^2)$. But, it is easy to find out that values that are used to form a column are significantly dependent on the values of the other columns. Therefore, except the first column that requires to be calculated independently, our proposed algorithm utilizes the mentioned dependency between columns to form each column in $O(n)$. Consequently, a single sigma matrix could be computed in $O(n^2)$ which leads us to the complexity of $O(n^3)$instead of $O(n^4)$ to calculate all of the coefficients. In the next section we have explained the relationship between columns of a sigma matrix and how to use it in order to expedite the calculations.

3. Demonstration of the proposed algorithm

In this section we show how a recursive algorithm can help us to calculate the jth ($j > 1$) column of the ith sigma matrix using the first column that was calculated before. Similar to what was explained in Note 2.1 we consider that, the ith sigma matrix uses a vector $v^{(n)}$ as follows to form its values:

$$v^{(n)} = [a_1, a_2, a_3, \dots, a_n], \quad a_j = j - i. \tag{23}$$

Note 3.1. The value name i which is used in 'ith *sigma matrix*' and '$v^{(n)} = [a_1, a_2, a_3, \dots, a_n]$, $a_j = j - i$' should not be confused with $\sigma^{(n)}_{i,j}$ that comes next. The ith sigma matrix is denoted by $\sigma^{(n)}$. We only have used the same name for entries of sigma matrix for the simplicity of notation in the rest of this section. Also, from here on, when we state the jth column we refer to a column in the sigma matrix other than the first one ($j > 1$). The jth column of $\sigma^{(n)}$ can be shown as follows:

$$\sigma^{(n)}_{i,j(1\le i\le n)} = \begin{bmatrix} 1 \\ \displaystyle\sum_{\substack{1\le i1\le n\\ i1\ne j}} a_{i1} \\ \displaystyle\sum_{\substack{1\le i1<i2\le n\\ i1\ne j, i2\ne j}} a_{i1}\cdot a_{i2} \\ \displaystyle\sum_{\substack{1\le i1<i2<i3\le n\\ i1\ne j, i2\ne j, i3\ne j}} a_{i1}\cdot a_{i2}\cdot a_{i3} \\ \vdots \\ \displaystyle\sum_{\substack{1\le i1<i2<\dots<in\le n\\ i1\ne j, i2\ne j,\dots, in\ne j}} a_{i1}\cdot a_{i2}\cdot a_{i3}\cdot \dots \cdot a_{in} \end{bmatrix}. \tag{24}$$

It is straightforward to show that the value of a_j does not appear in any product term for the jth column. Thus, if we denote values which are used in the jth column of $\sigma^{(n)}$ with $v^{(n)}_j$, we have

$$v^{(n)}_j = [a_1, a_2, \dots, a_{j-1}, a_{j+1}, \dots, a_n]. \tag{25}$$

In order to form $\sigma^{(n)}$, we need to calculate each column of $\sigma^{(n)}$ using its first column. The important point in calculating the jth column is that there is no need to consider all values in $v^{(n)}_j$ because many of the elements of $v^{(n)}_j$ can be found in $v^{(n)}_1$. Therefore, we should only consider the elements that exist in $v^{(n)}_j$ and do not exist in $v^{(n)}_1$.

Note 3.2. Although the difference between the elements of $v^{(n)}_j$ and $v^{(n)}_1$ is always in one element, in the rest of this section we generalize the related proofs for the proposed method to r elements. Hence, the reduced case of $r = 1$ would be sufficient to implement the results of algorithms albeit these general proofs have meaningful interpretation in order to follow in further research.

Before continuing, we make the following definitions.

Definition 3.1. Set of elements that exist in $v^{(n)}_1$ and do not exist in $v^{(n)}_j$ is defined as $P = \{p_1, p_2, \dots, p_r\}$.

Definition 3.2. The sum of the product of elements for every subset P with the length k is denoted by L_k ($k \leq r$) and is defined as follows:

$$L_k = \begin{cases} \sum\limits_{1\leq i1<i2<\cdots<ik\leq r} p_{i1} \cdot p_{i2} \cdot \cdots \cdot p_{ik}, & 1 \leq k \leq r \\ 1, \quad k = 0. \end{cases} \tag{26}$$

Definition 3.3. Set of elements that exist in $v_j^{(n)}$ and do not exist in $v_1^{(n)}$ is defined as $Q = \{q_1, q_2, \ldots, q_r\}$.

Definition 3.4. Similarly, the sum of the product of elements for every subset of Q with the length k is denoted by M_k ($k \leq r$) and is defined as follows:

$$M_k = \begin{cases} \sum\limits_{1\leq i1<i2<\cdots<ik\leq r} q_{i1} \cdot q_{i2} \cdot \cdots \cdot q_{ik}, & 1 \leq k \leq r \\ 1, \quad k = 0\cdot \end{cases} \tag{27}$$

Definition 3.5. For the first column of $\sigma^{(n)}$, the sum of every product term in the ith row that does not contain any elements of P is denoted by B_i. Furthermore, for the jth column, the sum of every product term in the ith row that does not contain any elements of Q is again equal to B_i. For instance, if $n = 5$ and $v_1^{(n)} = [a_2, a_3, a_4, a_5]$ then the first column of $\sigma^{(n)}$ becomes:

$$\sigma^{(5)}_{i,1(1\leq i\leq 5)} = \begin{bmatrix} 1 \\ a_2 + a_3 + a_4 + a_5 \\ (a_2 \times a_3) + (a_2 \times a_4) + (a_2 \times a_5) + (a_3 \times a_4) + (a_3 \times a_5) + (a_4 \times a_5) \\ (a_2 \times a_3 \times a_4) + (a_2 \times a_3 \times a_5) + (a_2 \times a_4 \times a_5) + (a_3 \times a_4 \times a_5) \\ a_2 \times a_3 \times a_4 \times a_5 \end{bmatrix}. \tag{28}$$

Therefore, if $j = 2$ then the only element participating in the first column that does not appear in the second column is a_2, so $P = \{a_2\}$. Similarly $Q = \{a_1\}$ and we have:

$$\begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \end{bmatrix} = \begin{bmatrix} 1 \\ a_3 + a_4 + a_5 \\ (a_3 \times a_4) + (a_3 \times a_5) + (a_4 \times a_5) \\ a_3 \times a_4 \times a_5 \\ 0 \end{bmatrix}. \tag{29}$$

Definition 3.6. For the entries in the first column of $\sigma^{(n)}$, the sum of every product term that contains exactly k elements of P is denoted by $S_{i,1,k}$, where $0 \leq k \leq \min(i-1, r)$. In a similar manner, $S_{i,j,k}$ is defined using elements of Q. Note that the values of $S_{i,1,k}$ and $S_{i,j,k}$ are not known unless we know the value of j and accordingly P and Q.

Considering the previous example, we have

$$\begin{bmatrix} S_{3,1,0} \\ S_{3,1,1} \end{bmatrix} = \begin{bmatrix} (a_3 \times a_4) + (a_3 \times a_5) + (a_4 \times a_5) \\ (a_2 \times a_3) + (a_2 \times a_4) + (a_2 \times a_5) \end{bmatrix}. \tag{30}$$

Moreover, in the cell $\sigma_{i,1}^{(n)}$ we can assume that every product term has exactly zero elements of P or exactly two elements of P or …exactly $\min(i-1, r)$ elements of P. Therefore, the value of the cell can be presented in the following form:

$$\sigma_{i,1}^{(n)} = \sum_{k=0}^{\min(i-1,r)} S_{i,1,k}. \tag{31}$$

Since product terms of B_{i-k} have a length of $i - k - 1$, we have

$$S_{i,j,k} = B_{i-k} \times M_k. \tag{32}$$

And

$$S_{i,1,k} = B_{i-k} \times L_k. \tag{33}$$

For example, if $j = 2$ then Q is $\{a_1\}$. From Eq. (29) and definition of L_k and M_k we get:

$$S_{4,2,1} = (a_1 \cdot a_3 \cdot a_4) + (a_1 \cdot a_3 \cdot a_5) + (a_1 \cdot a_4 \cdot a_5) = (a_3 \cdot a_4 + a_3 \cdot a_5 + a_4 \cdot a_5) \times a_1 = B_3 \times M_1$$

$$S_{4,1,1} = (a_2 \cdot a_3 \cdot a_4) + (a_2 \cdot a_3 \cdot a_5) + (a_2 \cdot a_4 \cdot a_5) = (a_3 \cdot a_4 + a_3 \cdot a_5 + a_4 \cdot a_5) \times a_2 = B_3 \times L_1.$$

From Eq. (31), in order to update the value of the cell $\sigma_{i,1}^{(n)}$, we only need to update the values of $S_{i,1,k}$ in which $k \geq 1$ because in the case of $k = 0$, no element of P will be involved. Using this, we can update the first column to get the jth column. The updated value of $S_{i,1,k}$ is the sum of every product term that contains exactly k elements of Q and is equal to the value of $S_{i,j,k}$. Actually, $S_{i,j,k}$ only uses values of Qinstead of P; therefore, it can be used in forming values for the jth column. Similar to Eq. (31) the updated value of $\sigma_{i,1}^{(n)}$ which is equal to $\sigma_{i,j}^{(n)}$ can be expressed as follows:

$$\sigma_{i,j}^{(n)} = \sum_{k=0}^{\min(i-1,r)} S_{i,j,k}. \tag{34}$$

In order to calculate the value of $S_{i,j,k}$, we can write:

$$S_{i,j,k} = B_{i-k} \times M_k = B_{i-k} \times M_k + B_{i-k} \times L_k - B_{i-k} \times L_k = (B_{i-k} \times L_k) + (M_k - L_k) \times B_{i-k}.$$

Referring Eq. (32), we can conclude that:

$$S_{i,j,k} = S_{i,1,k} + (M_k - L_k) \times B_{i-k}. \tag{35}$$

Since $S_{i,j,0} = S_{i,1,0}$, we can start from $k = 1$ and define $G_k = M_k - L_k$, as a result the desired value of $\sigma_{i,j}^{(n)}$ can be obtained as follows:

$$\sigma_{i,j}^{(n)} = \sigma_{i,1}^{(n)} + \sum_{k=1}^{\min(i-1,r)} G_k \times B_{i-k}. \tag{36}$$

Eq. (36) is the main formula in our implemented method in Section 3.2. Before starting to update a column, we pre-calculate G_k and M_k where $1 \leq k \leq n-1$. Note that we also need to find the value of B_i to calculate the next cells. After updating a cell, similarly B_i can be calculated as follows:

$$B_i = \sigma_{i,j}^{(n)} - \sum_{k=1}^{\min(i-1,r)} M_k \times B_{i-k}. \tag{37}$$

Note 3.3. Our proposed algorithm aims to recursively calculate each column of a sigma matrix and just optimizes the previous method for the finite difference approximation of derivatives [9] in terms of time complexity. Although we could similarly obtain the first column of each sigma matrix using the first column of the previous matrix to reduce computational effort, this objective has not been developed as it is irrelevant to the main idea of the article. Therefore, we attempt to concentrate on calculating weighting coefficients in $O(n^3)$.

3.1. Time complexity analysis

As mentioned earlier, the analysis is done in a more general form for an arbitrary value of r $(1 \leq r \leq n)$. In order to update each column while forming a specific sigma matrix, there exist three main parts that can be analyzed as follows:

3.1.1. Determining P and Q

The elements of P and Q for a column can be found in linear time because we only need to scan through the vector $v_1^{(n)}$ and check whether the current element appears in $v_j^{(n)}$ or not. Therefore, this part has a complexity of $O(n)$.

3.1.2. Determining L_k, M_k and G_k

A naive algorithm needs to generate every subset of the r elements and calculate the sum of products, which has an exponential complexity of $O(2^r)$. However, there exists a relation between the sum of products with length t and the sum of products with length $t+1$. Therefore, a recursive solution can pre-calculate these values for the jth column with a complexity of $O(r^2)$. For instance, if we want to form the sum of products for a vector $v = [v_1, v_2, v_3, v_4]$ we can say that:

$$\textit{sum of products with length } 1 : X = v_1 + v_2 + v_3 + v_4$$
$$\textit{sum of products with length } 2 : X' = v_1 \times (X - v_1) + v_2 \times (X - (v_1 + v_2)) + v_3 \times (X - (v_1 + v_2 + v_3)).$$

Therefore, knowing such relationship, for a vector with r elements, the sum of products with length $t+1$ could be computed from the value for length t with $O(r)$ number of operations. And for an optimized version as in Listing 2 we would need an overall of $O(r^2)$ time and memory.

3.1.3. Updating the first column and calculating B_i

According to Eqs. (36) and (37), obtaining the value of $\sigma_{i,j}^{(n)}$ from $\sigma_{i,1}^{(n)}$ and calculating B_i for that cell has a complexity of $O(r)$, because the upper bound of sigma is $\min(i-1, r)$. Since we should perform this operation for every cell in the column, it takes $O(n \times r)$. We can obtain the first column of $\sigma^{(n)}$ with an algorithm of $O(n^2)$ as mentioned in 3.1.2. Since we need to update all $n-1$ remaining columns to form $\sigma^{(n)}$, the overall complexity is $O(n^2 + (n-1) \times (n + r^2 + n \times r))$. Knowing the fact that $r = 1$, calculating the whole sigma matrix can be done in $O(n^2)$.

3.2. Implementation

In order to compare the proposed method with the former method, we bring the equivalent C++ code of the Matlab code of [9] for calculating weighting coefficients. Note that in both algorithms the calculated coefficients are scaled by

$\lambda = h^m (n-1)!/m!$ as utilized in [9], so h is not needed in either of algorithms. The reason for scaling was to make the process of validation easier and more reliable in the next section. In Listing 1, the function *Sig* creates the sigma matrix for every i from 1 to n and function *Ceoff* generates the weighting coefficients.

```cpp
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;

typedef long long ll;

#define sz(a) ((int)a.size())
#define VV vector < vector <ll> > // is used as a matrix
#define V vector <ll> //is used as a vector

/*
using a factorial array to avoid recalculation
NOTE: factorials bigger than 12 cannot fit into 'long long'
*/
ll fact[200];

V minus_vec(V &v, int x){
        V ret = v;
        for(int i = 0; i < sz(v); i++)
                ret[i] -= x;
        return ret;
}

//===== calculating sigma matrix - O(n ^ 3) ======
VV sig(V &v){
        VV S(sz(v), V(sz(v), 1));
        VV s = S;
        for(int k = 0; k < sz(S); k++){
                V vv = v;
                vv[k] = v[0];
                for(int i = 1; i < sz(S); i++){
                        s[i][i] = s[i - 1][i - 1] * vv[i];
                        if(i > 1){
                                for(int j = i - 1; j >= 1; j--){
                                        s[j][i] = s[j - 1][i - 1] * vv[i] + s[j][i - 1];
                                }
                        }
                }
                for(int i = 0; i < sz(S); i++){
                        S[i][k] = s[i][sz(S) - 1];
                }
        }
        return S;
}

//==== generating weighting coefficients - O(n ^ 4) =====
VV coeff(int m, int O){
        int n = m + O;
        V k, v;
        VV sig_matrix;
        VV coeff_matrix(n, V(n, 0));
        for(int i = 1; i <= n; i++) k.push_back(i);
        for(int i = 1; i <= n; i++){
                v = minus_vec(k, i);
                sig_matrix = sig(v);
                for(int l = 1; l <= n; l++){
                        coeff_matrix[i - 1][l - 1] = ((((l - m - 1) & 1) ? -1 : +1)
                                * sig_matrix[n - m - 1][l - 1] * fact[n - 1]) / (fact[l - 1] * fact[n - l]);
                }
        }
        return coeff_matrix;
}
```

Listing 1. Equivalent C++ code of the algorithm used in [1]

Listing 2 indicates our proposed algorithm. The function *new_sig* receives $v^{(n)}$ and returns $\sigma^{(n)}$; it uses *cal_sums* to calculate the sum of products of every subset of $v_1^{(n)}$ with lengths from 0 to $n-1$. As explained earlier we assume that r is equal to 1. Therefore, we always have $P = \{a_j\}$ where $j > 1$ and $Q = \{a_1\}$. Accordingly, we can obtain M and G with a simple initialization and finally function *my_coeff* calculates weighting coefficients.

```cpp
// determines sum of products
V cal_sums(V &a){
	int r = sz(a);
	VV sum(r, V(r, 0));
	V ret(r);
	for(int i = 0; i < r; i++){
		ll cur_sum = (i - 1 >= 0 ? ret[i - 1] : 1);
		ll cur_ans = 0;
		for(int j = 0; j < r; j++){
			cur_sum -= (i - 1 >= 0 ? sum[i - 1][j] : 0);
			sum[i][j] = a[j] * cur_sum;
			cur_ans += sum[i][j];
		}
		ret[i] = cur_ans;
	}
	ret.insert(ret.begin(), 1);
	return ret;
}


// Eq(37) and Eq(38)
VV new_sig(V &v){
	VV S(sz(v), V(sz(v), 0));
	int n = sz(v);
	V B(n, 0), first_vec;
	//calculating first column
	for(int i = 1; i < n; i++)
		first_vec.push_back(v[i]);
	V first_col = cal_sums(first_vec);
	//initializing all of n columns with first column
	for(int j = 0; j < n; j++){
		for(int i = 0; i < n; i++){
			S[i][j] = first_col[i];
		}
	}
	//calculating n - 1 remaining columns
	for(int j = 1; j < n; j++){
		int M = v[0];
		int G = v[0] - v[j];
		for(int i = 0; i < n; i++){
			S[i][j] += G * (i - 1 >= 0 ? B[i - 1] : 0);
			B[i] = S[i][j] - M * (i - 1 >= 0 ? B[i - 1] : 0);
		}
	}
	return S;
}

VV my_coeff(int m, int O){
	int n = m + O;
	V k, v;
	VV sig_matrix;
	VV coeff_matrix(n, V(n, 0));
	for(int i = 1; i <= n; i++) k.push_back(i);
	for(int i = 1; i <= n; i++){
		v = minus_vec(k, i);
		sig_matrix = new_sig(v);
		for(int l = 1; l <= n; l++){
			coeff_matrix[i - 1][l - 1] = ((((l - m - 1) & 1) ? -1 : +1)
				* sig_matrix[n - m - 1][l - 1] * fact[n - 1]) / (fact[l - 1] * fact[n - l]);
		}
	}
	return coeff_matrix;
}

int main(){
	fact[0] = 1;
	for(int i = 1; i < 200; i++)
		fact[i] = (i > 12 ? 1 : i * fact[i - 1]);
	int m, o;
	cin >> m >> o;
	clock_t t1 = clock();
	coeff(m, o);
	cout << "Previous Method : " << clock() - t1 << endl;
	//=====================================
	clock_t t2 = clock();
	my_coeff(m, o);
	cout << "New Method : "  << clock() - t2 << endl;
	return 0;
}
```

Listing 2. Implementation of the proposed algorithm

Table 1
PC configuration.

Processor	Installed memory (RAM)	System type
Intel Core i3—CPU 2.10 GHz	4.00 GB	64-bit operating system

Table 2
Calculated scaled coefficients for $m = 4$ and $O = 5$, $C'_{i,l} = \lambda \times C_{i,l}$.

	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 6$	$l = 7$	$l = 8$	$l = 9$
$i = 1$	22 449	−147 392	428 092	−720 384	769 510	−534 464	235 452	−60 032	6769
$i = 2$	6769	−38 472	96 292	−140 504	132 510	−83 384	34 132	−8232	889
$i = 3$	889	−1232	−6468	21 616	−28 490	20 496	−8708	2128	−231
$i = 4$	−231	2968	−9548	12 936	−7490	616	1092	−392	49
$i = 5$	49	−672	4732	−13 664	19 110	−13 664	4732	−672	49
$i = 6$	49	−392	1092	616	−7490	12 936	−9548	2968	−231
$i = 7$	−231	2128	−8708	20 496	−28 490	21 616	−6468	−1232	889
$i = 8$	889	−8232	34 132	−83 384	132 510	−140 504	96 292	−38 472	6769
$i = 9$	6769	−60 032	235 452	−534 464	769 510	−720 384	428 092	−147 392	22 449

4. Numerical experimentation and code validation

In this section we aim to prove the validity and optimality of the proposed method. All of experimentations are done with Microsoft Visual C++ 2010 express, under release mode. The configuration of the PC is also shown in Table 1. To check the validity of the implemented code we use $m = 4$ and $O = 5$. Table 2 shows the results of the proposed algorithm that are exactly the same as calculated in [9].

In the next step, the concentration is on the optimality of the algorithm. Note that the zero values in Table 3 show that the execution time has been less than a millisecond. For relatively small values of m and O such as rows 1–3, our method does not work faster than the previous method. The reason is that the proposed method uses some costly operations to reduce time complexity rather than simple calculations of the previous method. Therefore, large values are needed to show that our method is better. But the growth of values in the sigma matrix and weighting coefficients is of the factorial order. Although large numbers and their operations can be implemented using arrays to store values, they cannot fit into currently available primitive data types (such as integers and long integers). For this reason, in this part we ignore the validity of the algorithm for the large values of n, and factorials larger than $12!$ are considered 1, to simultaneously prevent them from being overflowed during arithmetic operations and also avoid division by zero error. In conclusion, we only evaluate both algorithms in terms of execution time. Note that the algorithm is still valid for $n \leq 12$.

5. Applications

In this section we want to point out some of the major advantages of our method (other than mere better time complexity) and state a related specific application in the real world. In the previous sections we indicated that our method is faster than the previous one. But, here we want to show that even for small values of n our method is preferable. In the old method the complexity for calculating a sigma matrix was cubic not just because it used three successive loops. A more important point is that in that algorithm in addition to add operations, the number of multiplication operations was also of cubic order. In a more real approach, a multiplication costs quite more than an addition, so in this sense our method is better. Hence, if we only optimize our implementation for the case $r = 1$ and avoid calling *cal_sums* subroutine the proposed algorithm always works faster and has less computational effort.

Another point is that this filter can be used indirectly to improve accuracy. Compared to the old method, the quickness of the new one allows us to obtain higher order of accuracy for derivates. Suppose that we want to calculate the derivative of degree m and order of accuracy O, so here we have $n = m + O$. Since the new method is faster, instead of O we can use a remarkably higher order of accuracy O' $\left(O' > O\right)$, and now size of input is $n' = m + O'$ which is strictly larger than n and of course more operations are required to complete the calculations. Hence, we can adjust the value of O' in a way that the number of operations for n in the old method is equal to the number of operations for n' in the new method. Consequently, with the same amount of operations, we can obtain a higher order of accuracy.

Having the two mentioned points in mind, the proposed method of differentiating could be used in order to obtain derivatives with a higher order of accuracy without imposing much computational effort. In the next two sections we show that how *other features of our algorithm* make it an excellent choice for differentiating when we simultaneously need accuracy and speed while working with DSPs in the real applications.

5.1. Locating signal peaks

In this part we want to use a short example (borrowed from [11]) to explain how signal derivatives can be used for locating peaks which are hard to realize due to strong influence of noise on the signal. This part is a prerequisite to better

Table 3
Comparison of previous and new methods in terms of execution time.

Row#	m	O	$n = m+O$	Previous method (ms)	New method (ms)
1	5	5	10	0	1
2	8	9	17	0	1
3	10	10	20	1	1
4	9	18	27	4	1
5	6	22	28	5	2
6	14	15	29	7	2
7	25	25	50	39	7
8	47	7	54	50	8
9	37	32	69	132	15
10	39	31	70	136	16
11	16	66	82	261	25
12	31	52	83	271	26
13	16	67	83	268	26
14	77	7	84	279	27
15	44	45	89	352	31
16	55	41	96	474	39
17	24	74	98	522	42
18	35	64	99	574	44
19	50	50	100	561	44
20	65	44	109	786	57
21	61	53	114	936	64
22	8	114	122	1227	78
23	73	52	125	1397	88
24	50	79	129	1527	93
25	6	144	150	2769	143
26	64	116	180	5751	265
27	100	100	200	8865	344

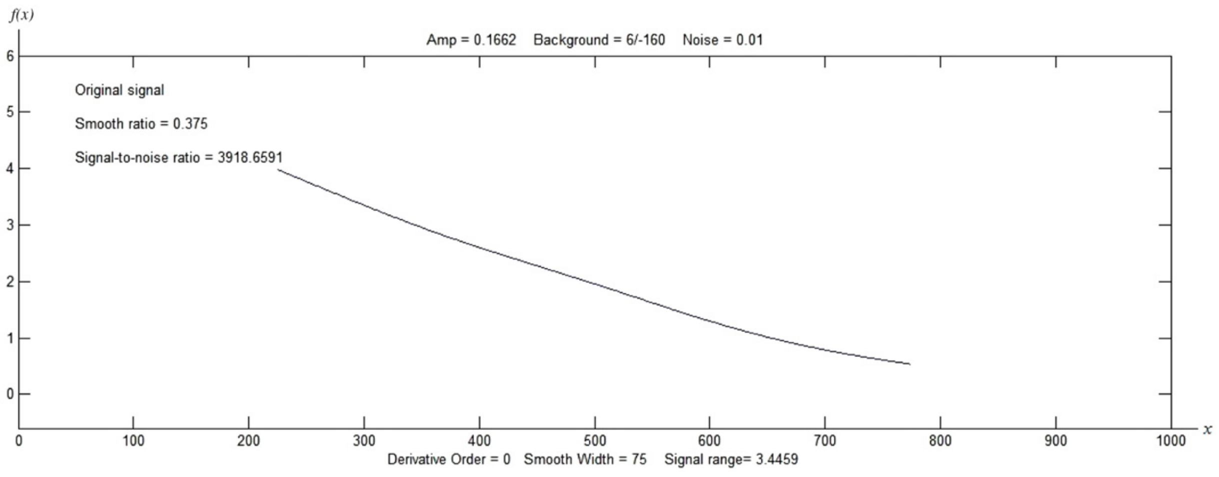

Fig. 2. Original signal, the peak at $x = 500$ is barely visible.

describe the application in the next section. Consider a signal as shown in Fig. 2 where a peak has been buried at $x = 500$ as a result of strong background signal. Here the signal which is a peak-type has been smoothed in order to increase the signal-to-noise ratio. After computing the 4th derivative of the signal we get Fig. 3. If we scale Fig. 3 we get a signal in which the location of the peak is noticeably obvious (Fig. 4). Hence, using high order derivatives of a noisy signal we can extract the desired features more easily.

5.2. ECG signal processing

Nowadays electrocardiogram (ECG) is extensively in use in order to diagnose heart diseases. In fact, the ECG signal contains a lot of valuable information which play a great role in primary medical decisions. However, due to various sources of noise such as those caused by electromyogram because of the motion of the muscles, power line interferences, or motions of instruments and even patients, it is not easy to obtain an ECG with high quality [12]. To indicate the ECG signal usually a wave of the form PQRST is used as shown in Fig. 5. The QRS part of the ECG signal is the most visually apparent part in the middle and is of significant importance because of the information it provides about right and left ventricle depolarization.

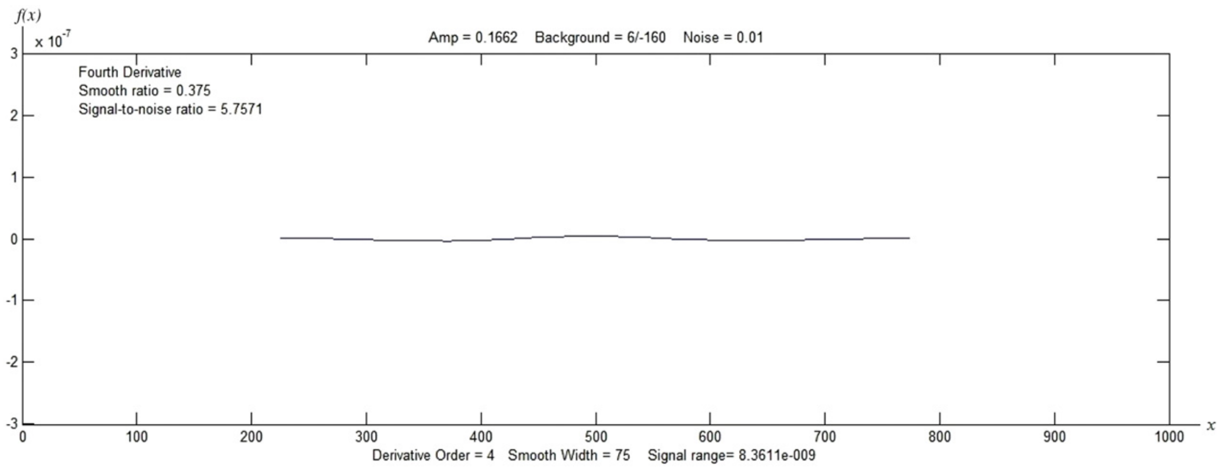

Fig. 3. 4th derivative of the original signal.

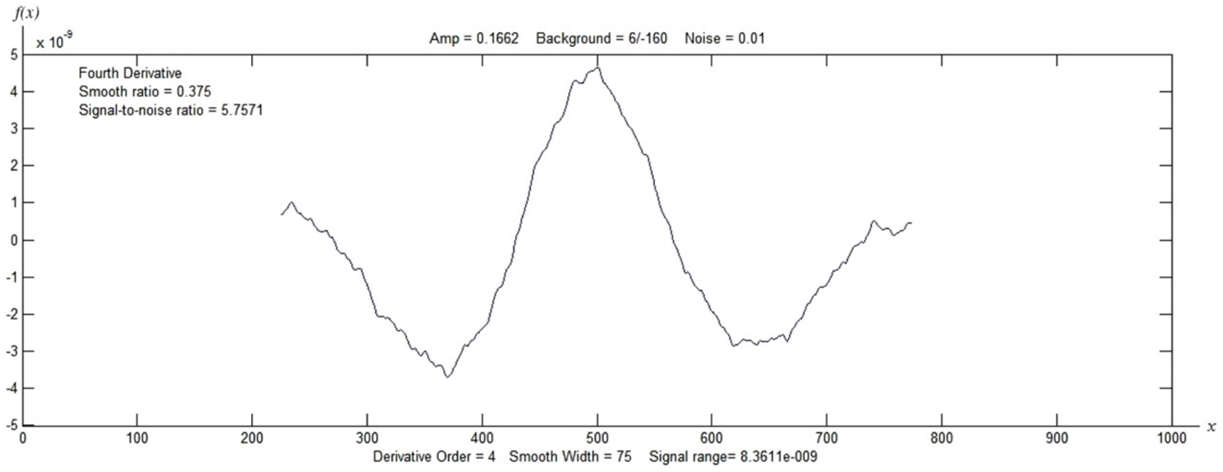

Fig. 4. 4th derivative of the original signal after scaling.

Therefore, the task of detecting QRS quickly and accurately is of great importance in ECG instruments. One of the needs in locating QRS complex precisely is the slope of the R wave which could be calculated using derivative algorithms. Apparently, the nature of ECG implies real time signal processing to keep track of information and changes in the signal. This is why QRS detectors (e.g. Pan–Tompkins Algorithm [13]) have a real time approach.

Recently, parallel processing techniques and multiprocessor DSPs have significantly accelerated computations by taking advantage of parallel algorithms. Having that in mind, we aim to show another interesting characteristic of our algorithm which makes it effective in such approaches. After what was explained in Section 3, it is easy to notice that unlike the old method which needs all previously calculated values, our method only depends on the first column in order to compute a sigma matrix. Hence, after determining the first column, we can divide the task of computing remaining columns among processors and obtain a significant speed. In a word, our method is applicable in parallel computing because it forms the solution of a large problem by solving smaller independent problems. Particularly, if we are using k processors (or have simulated the work of k processors), forming each sigma matrix takes $O(n^2/k)$ instead of $O(n^3)$. Thus, we can provide the coefficients needed for calculating derivatives a lot faster. Thus, in an application like ECG signal processing which needs both fast real time computations and accuracy, the combination of our method and parallel processing provides satisfactory results.

For another advantage, methods which use Taylor series [7,14] are more efficient in differentiating a noisy signal. Actually, differentiator filters which are based on finite difference method are very sensitive to noise and fail to estimate high order derivatives with sufficient accuracy. Moreover, if we are interested in obtaining highest possible accuracy and possess powerful DSPs we cannot be limited to primitive data types, but can take advantage of massive numbers where operations are done using string processing, such as BigInteger in JAVA. As a result, the size of input can freely increase and reveal the true power of the algorithm. As shown in the execution-time comparison table, when n is near 70 our method is about 10 times faster. However, one could argue that an algorithm which reveals its true speed at such a large n is not practical

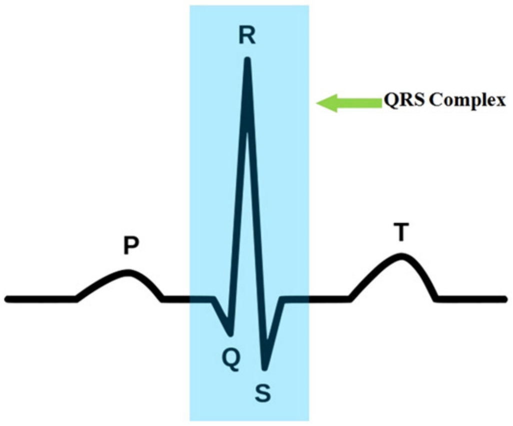

Fig. 5. QRS complex in the ECG signal.

enough, because the number of applications which need derivatives of degree near 70 or higher is apparently limited. But, the same argument is not acceptable for the order of accuracy, due to the fact that we would always like higher order of accuracy even if the degree of derivative is not large. Obviously, the proposed optimization becomes valuable in such cases and allows us to take advantage of the algorithm when precision is the most crucial factor in our application.

Finally, it would be helpful to mention that some alternatives like pre-calculating are not as effective as they seem. It might seem that as long as the values of the inverse of Vandermonde matrices are known pre-calculating these values and using them when needed is a good option because the only remaining unknown parameter is h (sampling period). But such pre-calculation requires $O(n^4)$ time and $O(n^3)$ memory, instead of $O(n^3)$ and $O(n^2)$. Obviously enough, to use this approach we should be able to determine an upper bound for n in our applications. And if such upper bound is large we would have a significant burden of calculation. On the other hand, if the estimated upper bound is never achieved we would have a significant waste of memory.

6. Conclusions

All considered, in this paper we provided a recursive algorithm to improve the process of calculating weighting coefficients as the main part of calculating derivatives in the work of Hasan et al. [9]. The proposed method was completely implemented and its validity was confirmed through proofs and experimentations. The complexity analysis showed that the proposed algorithm calculates the coefficients with the complexity of $O(n^3)$ or $O(n^3/k)$ when we use k processors. Hence, the previous method can easily be substituted in the cases in which more optimality and high precision are needed.

References

[1] I.R. Khan, R. Ohba, Digital differentiators based on taylor series, IEICE Trans. Fundam. E82-A (12) (1999).
[2] I.R. Khan, R. Ohba, New design of full band differentiators based on Taylor series, IEE Proc. Vis. Image Signal Process. 146 (4) (1999).
[3] O. Vainio, M. Renfors, T. Saramalu, Recursive implementation of FIR differentiators with optimum noise attenuation, IEEE Trans. Instrum. Meas. 46 (5) (1997).
[4] M.A. Al-Alaoui, Linear phase low-pass IIR digital differentiators, IEEE Trans. Signal Process. 55 (2) (2007).
[5] M.A. Al-Alaoui, Novel approach to designing digital differentiators, Electron. Lett. 28 (15) (1992).
[6] I.W. Selesnick, C.S. Burrus, Exchange algorithms for the design of linear phase FIR filters and differentiators having flat monotonic passbands and equiripple stopbands, IEEE Trans. Circuits Syst. II 43 (9) (1996).
[7] J.A. de la O Serna, M.A. Platas-Garza, Maximally flat differentiators through WLS Taylor decomposition, Digit. Signal Process. 21 (2011) 183–194.
[8] V.V. Sondur, et al., Design of a fifth-order FIR digital differentiator using modified weighted least-squares technique, Digit. Signal Process. 20 (2010) 249–262.
[9] H.Z. Hasan, A.A. Mohamad, G.E. Atteia, An algorithm for the finite difference approximation of derivatives with arbitrary degree and order of accuracy, J. Comput. Appl. Math. 236 (2012) 2622–2631.
[10] I.R. Khan, R. Ohba, Closed-form expressions for the finite diference approximations of First and higher derivatives based on Taylor series, J. Comput. Appl. Math. 107 (1999) 179–193.
[11] C. Thomas, O'Haver Homepage, University of Maryland, Date of access: 16 May 2013. http://terpconnect.umd.edu/~toh/spectrum/Differentiation.html.
[12] Udai Indu, P.R. Lekshmi, K Mathews Sherin, Maria Daie Tinu, T.S. Manu, ECG signal processing using DSK TMS320C6713, IOSR J. Eng. 2 (10) (2012) Oct. edition, 02950005.
[13] J. Pan, W.J. Tompkins, A real-time QRS detection algorithm, IEEE Trans. Biomed. Eng. (3) (1985) 230–236.
[14] Ronald L. Allen, Duncan Mills, Signal Analysis: Time, Frequency, Scale, and Structure, Wiley–IEEE Press, 2004.