Proceedings of the Federated Conference on Computer Science and Information Systems pp. 507–511
DOI: 10.15439/2017F428
ISSN 2300-5963 ACSIS, Vol. 11

An algorithm for Gaussian Recursive Filters in a Multicore Architecture

Ardelio Galletti, Giulio Giunta, Livia Marcellino, Diego Parlato
University of Naples "Parthenope"
Department of Science and Technology
Centro Direzionale, Isola C4, 80143, Naples, Italy
Email: {ardelio.galletti, giulio.giunta,livia.marcellino,diego.parlato}@uniparthenope.it

***Abstract*—Recursive Filters (RFs) are a well-known way to approximate the Gaussian convolution and, due to their computational efficiency, are intensively used in several technical and scientific fields. The accuracy of the RFs can be improved by means of the repeated application of the filter, which gives rise to the so-called K-iterated Gaussian recursive filter. In this work we propose a parallel algorithm for the implementation of the K-iterated first-order Gaussian RF for multicore architectures. This algorithm is based on a domain decomposition with overlapping strategy. The presented implementation is tailored for multicore architectures and makes use of the Pthreads library. We will show through extensive numerical tests that our parallel implementation is very efficient for large one-dimensional signals and guarantees the same accuracy level of the sequential K-iterated first-order Gaussian RF.**

I. Introduction

NOWADAYS, the recursive filters (RFs) have become a useful computational tool in several fields. For example, Gaussian RFs are usually involved in image processing [1], [2], in data assimilation for solving three-dimensional variational analysis schemes [3], [4] and in advanced signal processing such as other class of RFs has been recently constructed specifically for the electrocardiogram (ECG) denoising [6], [7], [8]. The idea of a recursive filter is to provide a more efficient approximation either to a given filter operator, or to the convolution with the impulse response of the filter. Gaussian RFs are basically designed to approximate Gaussian-based convolutions and can be built in many ways (see [9] and the references therein). It is well-known that Gaussian RFs, when applied to signals with support in a finite domain, generate distortions and artifacts, mostly localized at the boundaries. This issue is known as *edge effect* and heuristic and theoretical tools, namely *boundary conditions*, have been proposed in literature to remove it [9], [10]. These tools can be used even in the more general case in which a Gaussian RF is repeatedly applied, i.e. the so-called K-iterated Gaussian RFs [3] where K denotes the number of filter iterations. In this paper, we consider K-iterated first-order Gaussian recursive filters. The analysis of such filters has been recently provided in terms of accuracy [3], [5].

Although Gaussian RFs have low computational complexity, they may become inapplicable in practice when the size of the input signal is very large, so that there is the need of their parallel implementation. A thorough survey on parallel implementations of RFs is in [11]. The aim of our work is to introduce a new parallel algorithm for the K-iterated first-order Gaussian RF for 1D signals, based on a suitable domain decomposition with overlapping. Our approach is specifically designed for multicore architectures and is based on the Pthreads library. The paper is organized as follows. In Section II we briefly recall some mathematical preliminaries about the K-iterated first-order Gaussian RFs. Section III deals with the structure of the parallel algorithm and the underlying domain decomposition strategy. In Section IV we give some details about the implementation of the parallel algorithm in a multicore environment. Moreover, we discuss results of numerical tests that show that our parallel algorithm reaches the same accuracy of the sequential one, and we provide evidence of the gain obtained by the parallel implementation in terms of performance. Finally conclusions and future work are drawn in Section V.

II. Preliminaries

In the following we recall some preliminaries, notations and results presented in [5], [9]. We limit our description to the arguments needed for the understanding of the parallel approach proposed in next section. We first introduce the K-iterated n-order Gaussian RFs and in particular exhibit a sequential algorithm for the first-order case. Let:

$$s^{(0)} = \left\{s_j^{(0)}\right\}_{j\in\mathbf{Z}} = \left(\ldots, s_{-2}^{(0)}, s_{-1}^{(0)}, s_0^{(0)}, s_1^{(0)}, s_2^{(0)}, \ldots\right)$$

be an input signal. $s^{(0)}$ can be thought of as a complex function defined on the set of integers, that is an element of the set of sequences of complex numbers $\mathbf{C}^{\mathbf{Z}}$. Let g denote the Gaussian function with zero mean and standard deviation σ:

$$g(t) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{t^2}{2\sigma^2}\right). \tag{1}$$

Let $\delta = \left\{\delta_j\right\}_{j\in\mathbf{Z}}$ be the *unit-sample*:

$$\delta_j = \begin{cases} 1 & \text{if } \ j = 0 \\ 0 & \text{if } \ j \neq 0 \end{cases} \tag{2}$$

The Gaussian filter is a filter whose response to the unit-sample (i.e. the impulse response) is the restriction of the Gaussian

function g on $\mathbf{Z}$. Since the Gaussian filter is linear and time-invariant, its response $s^{(g)}$ to the input $s^{(0)}$ can be simply expressed by the discrete Gaussian convolution:

$$s_j^{(g)} = \left(g * s^{(0)}\right)_j = \sum_{t\in\mathbf{Z}} g_{j-t} s_t^{(0)}, \qquad \forall\, j \in \mathbf{Z}, \tag{3}$$

where $g_t \equiv g(t)$. Gaussian RFs and K-iterated Gaussian RFs are efficient tools for approximating the entries $s_j^{(g)}$ of $s^{(g)}$. A K-iterated n-order Gaussian RF filter generates an output signal $s^{(K)}$, the so-called K-iterate approximation of $s^{(g)}$, whose entries solve the $2K$ recurrence relations:

$$p_j^{(k)} = \beta s_j^{(k-1)} + \sum_{t=1}^{n} \alpha_t p_{j-t}^{(k)}, \quad \forall\, j \in \mathbf{Z},\ k = 1, 2, \ldots, K, \tag{4}$$

$$s_j^{(k)} = \beta p_j^{(k)} + \sum_{t=1}^{n} \alpha_t s_{j+t}^{(k)}, \qquad \forall\, j \in \mathbf{Z},\ k = 1, 2, \ldots, K. \tag{5}$$

For $K = 1$, the filter merely becomes an n-order Gaussian RF filter. As $K \to \infty$ the filter converges to the Gaussian filter[20]. Relations (4) and (5) are conveniently referred to as the advancing and backing filters, respectively. The values α_t and β are called *smoothing coefficients* and verify:

$$\beta = 1 - \sum_{t=1}^{n} \alpha_t.$$

In a general setting they depend on σ, n and K. In the following we consider only the first-order Gaussian RF, so that relations (4) and (5) take the simplified form [9]

$$p_j^{(k)} = \beta s_j^{(k-1)} + \alpha p_{j-1}^{(k)}, \qquad \forall\, j \in \mathbf{Z}, \tag{6}$$

$$s_j^{(k)} = \beta p_j^{(k)} + \alpha s_{j+1}^{(k)}, \qquad \forall\, j \in \mathbf{Z}. \tag{7}$$

Now, the smoothing coefficients are:

$$\alpha = 1 + E_\sigma - \sqrt{E_\sigma(E_\sigma + 2)}, \tag{8}$$

$$\beta = \sqrt{E_\sigma(E_\sigma + 2)} - E_\sigma, \tag{9}$$

with $E_\sigma = K\sigma^{-2}$. If we consider an input signal $s^{(0)}$ with support in the grid $\{0, 1, 2, \ldots, N-1\}$ then, in order to implement such a Gaussian RF as an algorithm, the index j must be treated in increasing order in the advancing filter (from 0 to $N-1$) and in decreasing order in the backing filter (from $N-1$ to 0) [9]. Such a scheme requires to set values $p_0^{(k)}$ and $s_{N-1}^{(k)}$ for priming the advancing and backing filters, respectively. A common choice is to set these values at zero or introduce the so-called *boundary conditions* that simulate the effect of the neglected filter equations in the algorithm. For first-order filters, the boundary conditions are [9]:

$$p_0^{(k)} = \frac{1}{1+\alpha} s_0^{(k-1)}, \qquad s_{N-1}^{(k)} = \frac{1}{1+\alpha} p_{N-1}^{(k)}.$$

Both approaches suffer from a well-known edge effect, that is a large perturbation on the boundary entries of the finite output signal. As shown in [9], provided that the input support is in $[0, N-1]$, this effect can be mitigated with a suitable *extending-resizing* strategy that allows an effective implementation of the K-iterated first-order Gaussian RF. This idea consists in three steps:

(i) *extending* the given input signal $s^{(0)}$, with support in $\{0, 1, 2, \ldots, N-1\}$, by adding artificial zero entries at the left and right boundaries. More specifically, we introduce the extended signal:

$$s^{(0),m} = \left(0, \ldots, 0, s_0^{(0)}, \ldots, s_{N-1}^{(0)}, 0, \ldots, 0\right), \tag{10}$$

which is obtained by inserting m zeros before $s_0^{(0)}$ and m zeros after $s_{N-1}^{(0)}$;

(ii) applying the K-iterated first-order Gaussian RF to $s^{(0),m}$, in order to obtain $s^{(K),m}$;

(iii) *resizing* the signal $s^{(K),m}$, by removing its first and last m entries, so that the (approximated) output signal $s^{(K)}$ is recovered.

The underlying idea of this scheme is to shift the edge effects on the artificially added entries. Step (i)-(iii) are shown in Figure 1.

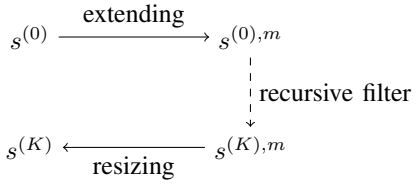

Fig. 1. *extending-resizing* strategy

Algorithm 1 implements this strategy, while **Algorithm 2** is a K-iterated first-order Gaussian RF straight implementation.

Algorithm 1 Extending-resizing strategy for the K-iterated first-order recursive filter

Input: $s^{(0)}$, σ, m, K

Output: $s^{(K)}$

1: extend $s^{(0)}$ to $s^{(0),m}$ as described in step (i)
2: apply **Algorithm 2** to $s^{(0),m}$ with parameters σ, K as described in step (ii), to obtain $s^{(0),m}$
3: resize $s^{(K),m}$ as described in step (iii), to obtain $s^{(K)}$

III. The parallel algorithm

In this section we describe the strategy underlying our parallel algorithm for a K-iterated first-order Gaussian RF and highlight the main feature of our parallel software for multicore environment. A multicore processor is a single computing component with two or more independent actual processing units, called "cores" (see Figure 2). The instructions are ordinary CPU instructions (such as add, move data, and branch), but the multiple cores can run multiple instructions

Algorithm 2 K-iterated first-order RF with boundary conditions

Input: $s^{(0)}$, σ, K
Output: $s^{(K)}$

set β, α as in (8) and (9); $M := 1/(1+\alpha)$
set $N := \mathrm{size}(s^{(0)})$
for $k = 1, 2, \ldots, K$ % filter loop
  compute $p_0^{(k)} := M s_0^{(k-1)}$ % left end conditions
  if $k = 1$ then
    $p_0^{(k)} := \beta s_0^{(k-1)}$
  end
  for $j = 1, \ldots, N-1$ % advancing filter
    $p_j^{(k)} := \beta s_j^{(k-1)} + \alpha p_{j-1}^{(k)}$
  endfor
  compute $s_{N-1}^{(k)} := M p_{N-1}^{(k)}$ % right end conditions
  for $j = N-2, \ldots, 0$ % backing filter
    $s_j^{(k)} := \beta p_j^{(k)} + \alpha s_{j+1}^{(k)}$
  endfor
endfor

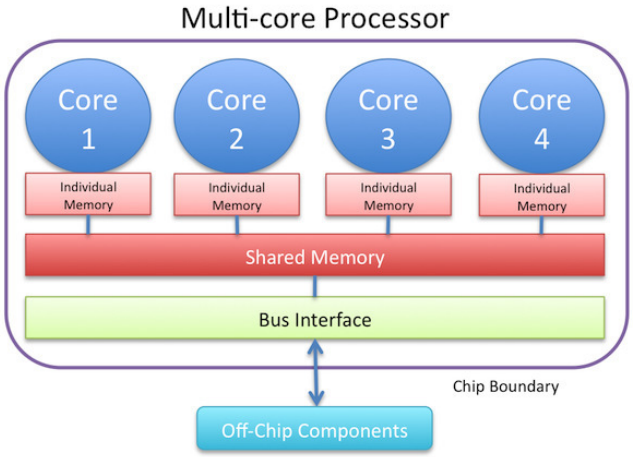

Fig. 2. Multicore architecture scheme.

concurrently, the so-called concurrent threads, increasing the overall efficiency.

Multicore processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU).

Possible gains in efficiency are limited by the fraction of the software that can run in parallel simultaneously on multiple cores. In the best case, the so-called embarrassingly parallel problems, one can reach a speedup factor close to the number of cores, or even more, if problems' data are decomposed so that they fit within each core cache, thus avoiding the use of the slower main memory.

To implement the algorithm in this environment, our starting idea is to divide in local blocks the original signal and to apply the strategy shown in previous section to each block. Finally, after collecting local outputs, the output signal is recovered. This is a typical domain decomposition approach in which the starting signal $s^{(0)}$ is first partitioned in t signal blocks, with t number of threads:

$$s_{\mathbf{0}}^{(0)},\ s_{\mathbf{1}}^{(0)}, \ldots,\ s_{\mathbf{t-1}}^{(0)}. \tag{11}$$

More in particular, denoting by $d = \lfloor \frac{N}{t} \rfloor$, $r = mod(N, t)$, the $\mathbf{j}$-th block ($\mathbf{j} = 1, \ldots, t$) has components:

$$\left(s_{\mathbf{j}}^{(0)}\right)_i = \begin{cases} s_{\mathbf{j}d+\mathbf{j}+i}^{(0)}, & i = 0, \ldots, d \quad \text{if } \mathbf{j} < \mathrm{r} \\ s_{\mathbf{j}d+r+i}^{(0)}, & i = 0, \ldots, d-1 \quad \text{if } \mathbf{j} \geq \mathrm{r} \end{cases} \tag{12}$$

We highlight that such a domain decomposition strategy relies in a sort of new method in which the output pieces are an approximation of the Gaussian convolution of the local inputs. In other words, the global output, i.e. the approximation of the Gaussian convolution of the global input, is obtained by collecting approximated local Gaussian convolutions of the local inputs that neglect the farthest entries.

A naive choice would be to apply the *extending-resizing* strategy to each block $s_{\mathbf{j}}^{(0)}$, so that each thread extends (with $2m$ zeros) its local input, computes the local output by means of **Algorithm 2**, and resizes the local output. Finally the global signal output can be restored by gathering all resized outputs. However this strategy suffers a serious drawback, that is a low accuracy in the output entries close to the cut points of the domain decomposition. This can be explained observing that each thread uses as extended local input a block with support in $[m+1, m+d+1]$ (or $[m+1, m+d]$) instead of the input entries of $s^{(0)}$. This can be seen as a perturbation in the local input signal boundary entries which causes a significant distortion in the output entries. In order to overcome the above drawback we devised a more appropriate *extending-resizing* strategy that, for extending the local inputs, uses the known entries of the signal input instead of zero values.

(i') *domain decomposition with overlapping*. The starting signal $s^{(0)}$ is partitioned in t input blocks $s_{\mathbf{j}}^{(0)}$ as in (11), (12), and each block is assigned to a thread. Each thread *extends* its block by adding m actual input entries at the left boundary and m actual input entries at the right boundary. When not available these entries are set at zero. Formally, the extended input signal components are:

$$\left(s_{\mathbf{j}}^{(0),m}\right)_i = \begin{cases} s_{\mathbf{j}d+\mathbf{j}+i-m}^{(0)}, & i = 0, \ldots, d+2m \quad \text{if } \mathbf{j} < \mathrm{r} \\ s_{\mathbf{j}d+r+i-m}^{(0)}, & i = 0, \ldots, d+2m-1 \quad \text{if } \mathbf{j} \geq \mathrm{r} \end{cases}$$

where conventionally $s_l^{(0)} = 0$ for $l < 0$ and $l \geq N$, so that zeros are considered only when the input entries do not exist. This kind of decomposition is illustrated in Figure 3;

(ii') each thread $\mathbf{j}$ applies the K-iterated first-order Gaussian RF to $s_{\mathbf{j}}^{(0),m}$, in order to obtain $s_{\mathbf{j}}^{(K),m}$;

(iii') *resizing with collecting*. Each thread $\mathbf{j}$ *resizes* the signal $s_{\mathbf{j}}^{(K),m}$, by removing its first and last m entries and generates the local output $s_{\mathbf{j}}^{(K)}$. Finally, the local outputs are collected in the global output signal.

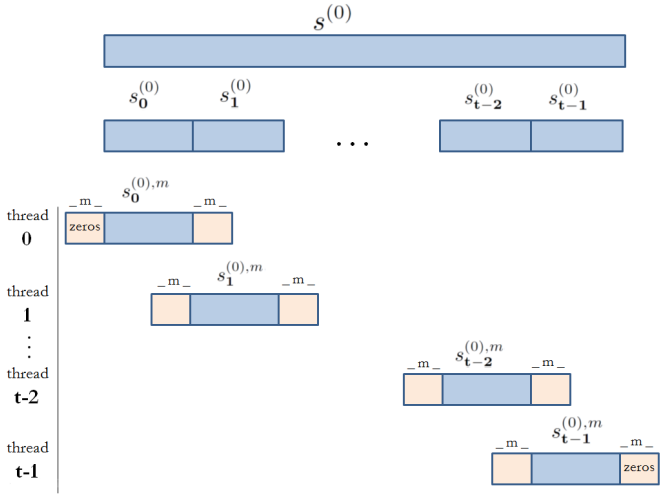

Fig. 3. Domain decomposition with overlapping scheme.

We observe that steps *(i')-(iii')* can be carried out by all threads in a fully-parallel way. In other words, our strategy can be considered embarrassingly parallel and can be summarized in **Algorithm 3**.

Algorithm 3 Parallel K-iterated first-order Gaussian recursive filter based on domain decomposition with overlapping

Input: $s^{(0)}$, σ, m, K, t
Output: $s^{(K)}$

1: FOR ALL THREAD **j**
2: save in the private memory of thread the extended input signal $s_{\mathbf{j}}^{(0),m}$ as described in step (i') (domain decomposition with overlapping)
3: apply **Algorithm 2** to $s_{\mathbf{j}}^{(0),m}$ with parameters σ, K as described in step (ii'), to obtain $s_{\mathbf{j}}^{(K),m}$
4: resize $s_{\mathbf{j}}^{(K),m}$ to recover $s_{\mathbf{j}}^{(K)}$ and copy it in the shared memory in order to obtain the global output $s^{(K)}$ as described in step (iii')
5: ENDFOR ALL THREAD **j**

IV. Implementation details and numerical tests

Our parallel algorithm is developed and tested on a CPU Intel Core i7 (2.8GHz), with 8 cores, 8GB of RAM memory, 4GB GDDR5 of memory size and 173 GB/s of memory bandwidth. In order to take full advantage of the capabilities provided by a multicore processor, a standardized interface is needed for programming the threads. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard. Implementations adhering to this standard are referred to as POSIX threads, or Pthreads[1]. The Pthreads library is a set of C language programming types and procedures for managing the synchronization and concurrency in a shared memory environment. Most hardware vendors offer Pthreads in addition to their proprietary API's.

[1]https://computing.llnl.gov/tutorials/pthreads/

TABLE I
$\sigma = 4, N = 2000$

K	$m = \sigma$	$m = 2\sigma$	$m = 3\sigma$	$m = 4\sigma$	$m = 5\sigma$
5	0.055592	0.056120	0.056124	0.056124	0.056124
15	0.021831	0.022343	0.022345	0.022345	0.022345
30	0.013884	0.013859	0.013861	0.013861	0.013861
50	0.011184	0.010460	0.010462	0.010462	0.010462
100	0.009679	0.007908	0.007909	0.007909	0.007909

TABLE II
$\sigma = 4, N = 2000$, Number of threads = 2

K	$m = \sigma$	$m = 2\sigma$	$m = 3\sigma$	$m = 4\sigma$	$m = 5\sigma$
5	0.176246	0.062215	0.056333	0.056138	0.056126
15	0.197547	0.033730	0.022521	0.022351	0.022346
30	0.212070	0.029099	0.014043	0.013864	0.013861
50	0.220339	0.028026	0.010657	0.010465	0.010462
100	0.228025	0.027667	0.008127	0.007912	0.007909

A. Accuracy

Here we are interested in comparing, in terms of provided accuracy, the sequential implementation (**Algorithm 1**) and the parallel implementation (**Algorithm 3**) of the K-iterated first-order Gaussian RF. The accuracy is measured by the 2-norm $\|s^{(g)} - s^{(K)}\|_2$, where $s^{(g)}$ is the output of the standard Gaussian convolution, and $s^{(K)}$ is either the output of **Algorithm 1** or the output of **Algorithm 3** depending on the context.

In Table I we report the results obtained applying **Algorithm 1** to the random input signal used in [9], for several values of K and m with fixed σ and N. Following [9] we notice that the larger m the better the accuracy, but the choice $m = 2\sigma$ guarantees a good trade-off between accuracy and size of the extended signal ($N+2m$). In Table II-IV we report the results obtained applying **Algorithm 3** to same input signal of Table I, with $t = 2, 4, 8$ threads, respectively. We observe that, regardless of the number of threads, the parallel algorithm can obtain the same accuracy level of **Algorithm 1** with a slightly larger value of m ($m = 4\sigma$).

B. Performance analysis

Here we are interested in the performance of the parallel algorithm (**Algorithm 3**). The performance are measured in

TABLE III
$\sigma = 4, N = 2000$, Number of threads = 4

K	$m = \sigma$	$m = 2\sigma$	$m = 3\sigma$	$m = 4\sigma$	$m = 5\sigma$
5	0.279605	0.070410	0.056524	0.056145	0.056126
15	0.324447	0.046540	0.022704	0.022353	0.022346
30	0.349988	0.043858	0.014254	0.013866	0.013861
50	0.364236	0.043664	0.010899	0.010466	0.010462
100	0.377388	0.044023	0.008412	0.007913	0.007909

TABLE IV
NUMBER OF THREADS = 8

K	$m = \sigma$	$m = 2\sigma$	$m = 3\sigma$	$m = 4\sigma$	$m = 5\sigma$
5	0.418137	0.083128	0.056548	0.056123	0.056124
15	0.493410	0.064770	0.022829	0.022345	0.022345
30	0.533700	0.064082	0.014463	0.013860	0.013861
50	0.555979	0.064849	0.011178	0.010462	0.010462
100	0.576482	0.066054	0.008779	0.007909	0.007909

TABLE V
EXECUTION TIME IN SECONDS, FOR $K = 100$

t	$N = 2\,000$	$20\,000$	$200\,000$
1	3.4e-03	2.3e-02	2.4e-01
2	3.1e-03	2.3e-02	2.3e-01
3	2.6e-03	1.7e-02	1.7e-01
4	2.5e-03	1.3e-02	1.3e-01
5	2.3e-03	1.2e-02	1.2e-01
6	2.2e-03	1.0e-02	1.0e-01
7	2.0e-03	9.5e-03	9.0e-02
8	2.0e-03	8.3e-03	8.0e-02

terms of execution time. In Table V we report the execution times applying this algorithm to random input signals of increasing size ($N = 2\,000$, $N = 20\,000$, $N = 200\,000$) and varying the number of threads. The values of the other parameters are fixed as follows: $\sigma = 4$, $m = 4\sigma$ and $K = 100$. Table V shows that execution times decrease as the number of threads grows. In particular, an appreciable gain in time, expressed in percentage, reached with 8 threads and $N = 200\,000$, is:

$$\frac{0.24 - 0.08}{0.24} \cdot 100\% = 66.7\%.$$

V. CONCLUSIONS

In this work, we have presented a new parallel algorithm for the approximation of the one-dimensional Gaussian convolution, based on K-iterated Gaussian recursive filters. The algorithm has been implemented on a multicore architecture. We also provided preliminary results that show the accuracy and the efficiency of our algorithm. This is a first step towards the development of algorithms and softwares, for HPC many core environments, for an efficient computation of multidimensional Gaussian convolutions that appear across several technical and scientific fields as data assimilation [12], reputation systems [13], [14], classical [15], [16] and multidimensional interpolation [17], [18], [19], image processing and data mining.

REFERENCES

[1] van Vliet, L.J., Young, I.T., Verbeek, P.W.. - *Recursive Gaussian derivative filters*. The 14 th International Conference on Pattern Recognition, pp. 509-514, DOI: 10.1109/ICPR.1998.711192, 1998.

[2] Young, I.T., van Vliet L.J.. - *Recursive implementation of the Gaussian filter*. Signal Processing 44, pp 139-151, 1995.

[3] Cuomo, S., Farina, R., Galletti, A., Marcellino, L.. *-An error estimate of Gaussian recursive filter in 3Dvar problem* Federated Conference on Computer Science and Information Systems, FedCSIS 2014, art. no. 6933068, pp. 587-595, 2014. DOI: 10.15439/2014F279

[4] Cuomo, S., Galletti, A., Giunta, G., Marcellino, L. . *-Numerical Effects of the Gaussian Recursive Filters in Solving Linear Systems in the 3Dvar Case Study* (2017) Numerical Mathematics, 10 (3), pp. 520-540. DOI: 10.4208/nmtma.2017.m1528

[5] Galletti, A., Giunta, G.. *-Error analysis for the first-order Gaussian recursive filter operator*, 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), Gdansk, 2016, pp. 673-678. DOI: 10.15439/2016F455

[6] Cuomo, S., De Pietro, G., Farina, R., Galletti, A., Sannino, G.. - *A novel O(n) numerical scheme for ECG signal denoising* Procedia Computer Science, 51 (1), pp. 775-784, 2015. DOI: 10.1016/j.procs.2015.05.198

[7] Cuomo, S., De Pietro, G., Farina, R., Galletti, A., Sannino, G.. - *A framework for ECG denoising for mobile devices* PETRA 2015 ACM. ISBN 978-1-4503-3452-5/15/07, DOI: 10.1145/2769493.2769560, 2015.

[8] Cuomo, S., De Pietro, G., Farina, R., Galletti, A., Sannino, G.. - *A revised scheme for real time ECG Signal denoising based on recursive filtering*, Biomedical Signal Processing and Control, 27, pp. 134-144, 2016. DOI: 10.1016/j.bspc.2016.02.007

[9] Cuomo, S., Farina, R., Galletti, A., Marcellino, L.. - *A K-iterated scheme for the first-order Gaussian Recursive Filter with boundary conditions* Federated Conference on Computer Science and Information Systems, FedCSIS 2015, pp.641-647, 2015. DOI: 10.15439/2015F286

[10] Triggs, B., Sdika M.. - *Boundary conditions for Young-van Vliet recursive filtering*. IEEE Transactions on Signal Processing, 54 (6 I), pp. 2365-2367, 2006.

[11] Chaurasia, G., Kelley, J.R., Paris, S., Drettakis, G., Durand, F., - *Compiling High Performance Recursive Filters*. Proceedings of the 7th Conference on High-Performance Graphics, pp 8594, 2015.

[12] Montella, R., Agrillo, G., Mastrangelo, D., Menna, M. *A globus toolkit 4 based instrument service for environmental data acquisition and distribution* (2008) High Performance Distributed Computing - Proceedings of the 3rd International Workshop on Use of P2P, Grid and Agents for the Development of Content Networks 2008, UPGRADE'08, pp. 21-27. DOI: 10.1145/1384209.1384214

[13] Galletti, A., Giunta G., and Schmid G., *A mathematical model of collaborative reputation systems* International Journal of Computer Mathematics 89.17 (2012): 2315-2332.

[14] Cuomo, S., Michele, P.D., Piccialli, F., Galletti, A., Jung, J.E. *IoT-based collaborative reputation system for associating visitors and artworks in a cultural scenario* (2017) Expert Systems with Applications, 79, pp. 101-111. DOI: 10.1016/j.eswa.2017.02.034

[15] S. Cuomo, A. Galletti, G. Giunta, L. Marcellino. A class of piecewise interpolating functions based on barycentric coordinates. *Ricerche di Matematica*, Springer, (2014).

[16] S. Cuomo, A. Galletti, G. Giunta, L. Marcellino. Piecewise Hermite interpolation via barycentric coordinates: In memory of Prof. Carlo Ciliberto. *Ricerche di Matematica*, Springer, (2015).

[17] S. Cuomo, A. Galletti, G. Giunta, A. Starace - *Surface Reconstruction from Scattered Point via RBF Interpolation on GPU* In *Federated Conference on Computer Science and Information Systems, FedCSIS 2013*, pp. 433–440 (2013)

[18] S. Cuomo, A. Galletti, G. Giunta, L. Marcellino - *Reconstruction of implicit curves and surfaces via RBF interpolation* In *Appl. Numer. Math. (2016)*, http://dx.doi.org/10.1016/j.apnum.2016.10.016

[19] S. Cuomo, A. Galletti, G. Giunta, L. Marcellino. A novel triangle-based method for scattered data interpolation. *Applied Mathematical Sciences*,8 (133-136), pp. 6717-6724 (2014).

[20] Wells, William M. *Efficient synthesis of Gaussian filters by cascaded uniform filters.* IEEE Transactions on Pattern Analysis and Machine Intelligence 2 (1986): 234-239.