Journal of Applied Computer Science Methods

Published by University of Social Sciences

ACSM

Volume 5 Number 2 2013

University of Social Sciences, IT Institute

Społeczna Akademia Nauk
Łódź

Instytut Technologii Informatycznych iti

ISSN 1689-9636

International Journal of Applied Computer Science Methods

Journal of Applied Computer Science Methods

Published by University of Social Sciences

ACSM

Volume 5 Number 2 2013

University of Social Sciences, IT Institute

Społeczna Akademia Nauk
Łódź

Instytut Technologii Informatycznych iti

INTERNATIONAL JOURNAL OF APPLIED COMPUTER SCIENCE METHODS (JACSM)
is a semi-annual periodical published by the University of Social Sciences (SAN)
in Lodz, Poland.

PUBLISHING AND EDITORIAL OFFICE:
University of Social Sciences (SAN)
Information Technology Institute (ITI)
Sienkiewicza 9
90-113 Lodz
Tel.: +48 42 6646654
Fax.: +48 42 6366251
E-mail: acsm@swspiz.pl
URL: http://acsm.swspiz.pl

Print: Mazowieckie Centrum Poligrafii, ul. Duża 1, 05-270 Marki, www.c-p.com.pl, biuro@c-p.com.pl

AIMS AND SCOPE:

The **International Journal of Applied Computer Science Methods is** a semi-annual, refereed periodical, publishes articles describing recent contributions in theory, practice and applications of computer science. The broad scope of the journal includes, but is not limited to, the following subject areas:

Knowledge Engineering and Information Management: *Knowledge Processing, Knowledge Representation, Data Mining, Machine Learning, Knowledge-based Systems, Knowledge Elicitation, Knowledge Acquisition, E-learning, Web-intelligence, Collective Intelligence, Language Processing, Approximate Reasoning, Information Archive and Processing, Distributed Information Systems.*

Intelligent Systems: *Intelligent Database Systems, Expert Systems, Decision Support Systems, Intelligent Agent Systems, Artificial Neural Networks, Fuzzy Sets and Systems, Evolutionary Methods and Systems, Hybrid Intelligent Systems, Cognitive Systems, Intelligent Systems and Internet, Complex Adaptive Systems.*

Image Understanding and Processing: *Computer Vision, Image Processing, Computer Graphics, Pattern Recognition, Virtual Reality, Multimedia Systems.*

Computer Modeling, Simulation and Soft Computing: *Applied Computer Modeling and Simulation, Intelligent Computing and Applications, Soft Computing Methods, Intelligent Data Analysis, Parallel Computing, Engineering Algorithms.*

Applied Computer Methods and Computer Technology: *Programming Technology, Database Systems, Computer Networks Technology, Human-computer Interface, Computer Hardware Engineering, Internet Technology, Biocybernetics.*

DISTRIBUTION:

Apart from the standard way of distribution (in the conventional paper format), on-line dissemination of the JACSM is possible for interested readers.

CONTENTS

MULTIPLICATIVE ALGORITHM FOR CORRENTROPY-BASED NONNEGATIVE MATRIX FACTORIZATION

Ehsan Hosseini Asl[1], Jacek M. Zurada[1,2]

[1] Department of Electrical and Computer Engineering
University of Louisville, Louisville, KY, USA
ehsan.hosseiniasl@louisville.edu, jacek.zurada@louisville.edu

[2]IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Łódź, Poland

Abstract

Nonnegative matrix factorization (NMF) is a popular dimension reduction technique used for clustering by extracting latent features from high-dimensional data and is widely used for text mining. Several optimization algorithms have been developed for NMF with different cost functions. In this paper we evaluate the correntropy similarity cost function. Correntropy is a nonlinear localized similarity measure which measures the similarity between two random variables using entropy-based criterion, and is especially robust to outliers. Some algorithms based on gradient descent have been used for correntropy cost function, but their convergence is highly dependent on proper initialization and step size and other parameter selection. The proposed general multiplicative factorization algorithm uses the gradient descent algorithm with adaptive step size to maximize the correntropy similarity between the data matrix and its factorization. After devising the algorithm, its performance has been evaluated for document clustering. Results were compared with constrained gradient descent method using steepest descent and L-BFGS methods. The simulations show that the performance of steepest descent and L-BFGS convergence are highly dependent on gradient descent step size which depends on σ parameter of correntropy cost function. However, the multiplicative algorithm is shown to be less sensitive to σ parameterand yields better clustering results than other algorithms. The results demonstrate that clustering performance measured by entropy and purity improve the clustering. The multiplicative correntropy-based algorithm also shows less variation in accuracy of document clusters for variable number of clusters. The convergence of each algorithm is also investigated, and the experiments have shown that the multiplicative algorithm converges faster than L-BFGS and steepest descent method.

Key words: Nonnegative Matrix Factorization (NMF), Correntropy, Multiplicative Algorithm, Document Clustering

1 Introduction

Large size of data is one of the central issues in data analysis research. Processing these large amounts of data opens new issues related to data representation, disambiguation, and dimensionality reduction. A useful representation typically makes latent structure in the data explicit, and often reduces the dimensionality of the data so that additional computational methods can be applied. In this process it is important to reduce the data size without losing its most essential features. Therefore, a common ground in the various approaches of data mining is to replace the original data with a lower dimensional representation obtained via subspace approximation [1, 2, 4].

There are several methods to reduce the dimensionality of large data such as Principal Component Analysis (PCA), Singular Value Decomposition (SVD) and Independent Component Analysis (ICA). Often the data to be analyzed is nonnegative, and the low-rank data are further required to be comprised of nonnegative values in order to avoid contradicting physical realities. However, these classical tools cannot guarantee to maintain the nonnegativity [1]. Therefore, an approach of finding reduced rank nonnegative factors to approximate a given nonnegative data matrix becomes a natural choice. The Nonnegative Matrix Factorization (NMF) approach allows to create a lower rank data out of original data, while maintaining nonnegativity of matrices entries [1, 2, 3].

The NMF technique approximates a data matrix A with the product of low rank matrices W and H, such that $A \approx WH$ and the elements of W and H are nonnegative [1,2]. If columns of A would be data samples, then the columns of W can be interpreted as basis or parts from which data samples are formed, while the columns of H give the contribution of each basis which when combined form the corresponding data sample. In application of NMF to clustering, it is common to define clusters based on each basis vector, and assigning each data sample to a cluster based on basis contribution intensity which is found from matrix H.

Several cost functions have been used in the literature to implement the NMF for various types of applications and data type. Euclidean distance is the most common cost function used for many applications including text mining [1]. Kullback-Leibler divergence (KL-divergence) [1, 2], β-divergence [21, 22] are among other methods also used for different applications. However, the main issue is to find the matrix factors (W, H) that minimize the chosen cost function. There are several optimization algorithms in the literature to perform this optimum decomposition [3, 4, 8, 10, 11, 12]. Correntropy similarity function is a recently proposed cost function which has been used for different tasks of pattern recognition [23]. It has been introduced to NMF only recently in [24, 25, 26]. In this paper, a multiplicative algorithm for corren-

tropy-based NMF (MACB-NMF) has been developed and its performance has been investigated in comparison to general gradient descent method for document clustering application using several metrics.

This paper is organized as follows. Section 2 introduces the correntropy cost function. Section 3 discusses some developed optimization algorithms for NMF. In section 4, a multiplicative update algorithm for correntropy cost function (MACB) is presented. Experiments on real data set are presented in Section 5. The discussion and conclusions are presented in Section 6.

2 Correntropy Similarity Function

Given a data matrix $A \in \mathbb{R}^{m\times n}$ and a positive integer $k < \min\{m.n\}$, find nonnegative factorization into matrices $W \in \mathbb{R}^{m\times k}$ and $H \in \mathbb{R}^{k\times n}$ as

$$\min_{W,H} D(A|WH) \; subject\; to\; W \geq 0, H \geq 0 \tag{1}$$

where:

$A \geq 0$expresses nonnegativity of the entries of A(and not semidefinite positiveness),

$D(A|WH)$isa measure for goodness of fit such that

$$D(A|WH) = \sum_{i=1}^{m}\sum_{j=1}^{n} d([A]_{ij}|[WH]_{ij}) \tag{2}$$

where:

d(x|y) is a scalar cost function [22].

Several cost function are used and most of them use the Bregman divergence [7]. Generally, a divergence function is defined as follows

$$D_{\alpha}(a,b) = \begin{cases} a\dfrac{a^{\alpha}-b^{\alpha}}{\alpha} + b^{\alpha}(b-a) & : \alpha \in (0,1] \\ a(loga - logb) + (b-a) & : \alpha = 0 \end{cases} \tag{3}$$

where:

α is chosen to define the type of the divergence function.

Obviously, $D_1(a,b) = (a-b)^2$ is the Euclidean distance function, and $D_0(a,b)$ defines KL-divergence [13]. The most common function found in literature is shown below

$$D_{Euclidean}(A|WH) = \sum_{i=1}^{m}\sum_{j=1}^{n}\frac{1}{2}\left(A_{ij} - (WH)_{ij}\right)^2 \quad (4)$$

Using the above notation, the correntropy cost function is defined as

$$d_{correntropy}(a|b) = -exp\left(\frac{-(a-b)^2}{2\sigma^2}\right) \quad (5)$$

$$D_{correntropy}(A|WH) = -\sum_{i=1}^{m}\sum_{j=1}^{n}\exp\left(\frac{-\left(A_{ij} - (WH)_{ij}\right)^2}{2\sigma^2}\right) \quad (6)$$

where:

$\sigma$is a parameter of correntropy measure.

The optimization algorithms try to minimize the correntropy, since it is a similarity instead of distance between two elements. The algorithm for minimizing these cost functions is introduced in the next section.

3 Optimization Algorithms

A key issue of NMF factorization is to minimize the cost function while keeping elements of W and H matrices nonnegative. Another challenge is the existence of local minima due to non-convexity of $D(A|WH)$ in both $W$and H. Moreover, a unique solution to NMF problem does not exist, since for any invertible matrix B whose inverse is B^{-1}, a term $WBB^{-1}H$ could also be nonnegative. This is most probably the main reason for non-convexity of $D(A|WH)$ function [13].

Several algorithms exist for minimizing cost functions in the NMF context. Lee and Seung [1, 2] developed a multiplicative algorithm for solving Euclidean and KL-divergence in 2001. Sparse Coding and sparseness constraint which impose sparsity on H matrix was proposed by Hoyer in 2002 and 2004 [3, 5]. Alternating Least Square (ALS) [12], ALS using projected gradient descent (ALSPGRAD) [14], gradient descent with constrained least square (GD-CLS) [9], Quasi Newton method [11], Alternating Nonnegative Constrained Least Squares (ANLS) using active set and block principal pivoting [17, 20], Hierarchical Alternating Least Square (HALS) [19] was proposed for Euclidean cost function. Fevotte et al proposed several algorithms for minimizing β-divergence cost function [21, 22]. In 2012, Li et al convert general

Bregman divergence to Euclidean distance function using Taylor expansion and solve the corresponding function using HALS algorithm [25]. Du et al proposed a half-quadratic optimization algorithm to solve NMF based on correntropy cost function and developed a multiplicative algorithm for resulting weighted NMF [26].

In 2012, Ensari et al used general algorithms of Constrained Gradient Descent (CGD) method for solving the correntropy function [18] and compared the results with projected gradient descent method of Euclidean cost function [24, 25]. The major disadvantage of CGD is its dependency on σ parameter of correntropy cost function. As will be shown in the next section, the update rate of CGD algorithm is based on this parameter. In the next section, we derive the CGD algorithm based on multiplicative update rule which has adaptive update learning rate and less sensitivity to variation of σ parameter.

4 Multiplicative Algorithm for Correntropy-based NMF

This section proposes a multiplicative algorithm for correntropy cost function (MACB). To minimize (6) using gradient descent algorithm, its gradient should be taken with respect to W and H matrices' elements which are parameters of cost function. The gradients $\nabla_W(D_\varphi), \nabla_H(D_\varphi)$are calculated as follows,

$$\nabla_W(D_\varphi(A\|WH)) = {}^1\!/_{\sigma^2}\left[\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\odot(WH-A)\right]H^T \quad (7)$$

$$\nabla_H(D_\varphi(A\|WH)) = {}^1\!/_{\sigma^2}\,W^T\left[(WH-A)\odot\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right] \quad (8)$$

where:

$\odot$is the element-wise product of two matrices.

As can be seen from Equations(7) and (8), the gradient formula involves the step size in the direction of gradient that is proportional to $1/\sigma^2$ parameter. Therefore, the gradient step variation could cause the solution to deviate from the limit points of the feasible region. This may result in unsatisfactory solution for $W$and H.

The multiplicative gradient descent approach is equivalent to updating each parameter by multiplying its value at previous iteration by the ratio of the negative and positive parts of the gradient of the cost function with regard to this parameter [2, 11]. Suppose there is a function $f(\theta)$ which should be minimized over θ. Gradient descent using multiplicative algorithm is equivalent to,

$$\theta \leftarrow \theta \frac{[\nabla f(\theta)]_-}{[\nabla f(\theta)]_+} \qquad (1)$$

where:

$$\nabla f(\theta) = [\nabla f(\theta)]_+ - [\nabla f(\theta)]_- \qquad (10)$$

and the summands are both nonnegative.This ensures nonnegativity of the parameter updates, provided initializationis with a nonnegative value. A fixed point $\theta^\star$ of the algorithm implies either $\nabla f(\theta_-) = 0$ or $\theta^\star = 0$[21, 22]. We apply this algorithm on Correntropy function gradients, Equations (7) and (8), and derive the update formula for W and H matrices respectively as follows,

$$W \leftarrow W \frac{[\nabla_W (D_\varphi(A\|WH))]_-}{[\nabla_W (D_\varphi(A\|WH))]_+} \qquad (11)$$

$$W \leftarrow W \odot \frac{\left[\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right) \odot A\right] H^T}{\left[\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right) \odot (WH)\right] H^T} \qquad (12)$$

$$H \leftarrow H \frac{[\nabla_H (D_\varphi(A\|WH))]_-}{[\nabla_H (D_\varphi(A\|WH))]_+} \qquad (13)$$

$$H \leftarrow H \odot \frac{W^T \left[A \odot \exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right]}{W^T \left[(WH) \odot \exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right]} \qquad (14)$$

As can be seen from Equations (12) and (14), the σ parameter is in numerator and denominator of update algorithm, which reduce the effect of variation of this parameter to the update algorithm. Although, we do not prove the non-increasing property of multiplicative update algorithm with Correntropy criterion analytically, the experimental results show that it is monotonic and non-increasing. It also give better results in comparison to other gradient descent methods. Therefore, MACB algorithm for NMF is as follows:

MACB-NMF Algorithm:

(1) Initialize W and H with nonnegative values, and scale the columns of W to unit norm.
(2) Iterate until convergence or for l iterations:

(a) $$W_{ij} \leftarrow W_{ij} \frac{\left(\left[\left(\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right)\odot A\right]H^T\right)_{ij}}{\left(\left[\left(\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right)\odot (WH)\right]H^T\right)_{ij}+\epsilon} \quad for\ i\ and\ j\ \ [\epsilon = 10^{-9}]$$

(b) $$H_{ij} \leftarrow H_{ij} \frac{\left(W^T\left[A\odot\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right]\right)_{ij}}{\left(W^T\left[(WH)\odot\exp\left(\frac{-(A-WH)^2}{2\sigma^2}\right)\right]\right)_{ij}+\epsilon} foriandj\ \ [\epsilon = 10^{-9}]$$

5 Experiments

This section outlines the design procedure of an experiment to test MACB algorithm. We employ Reuters Documents Corpus for document clustering. This original dataset contains 21578 documents and 135 topics or document clusters created manually. Each document in the corpus is been assigned one or more topics or category labels based on its content. The size of each cluster which is the number of documents it contains, range from less than ten to four thousand. For this experiment, documents associated with only one topic are used and topics which contain less than five documents are discarded [9]. Therefore, 8293 documents with 48 topics were left at the end. In order to evaluate the performance of the MACB for increasing complexity, i.e., the number of clusters to be created or the k parameter, ten different k values of $[2, 4, 6, 8, 10, 15, 20, 30, 40, 48]$ are chosen.

After creating clusters using NMF, the cluster is assigned to a most related document topic. For this purpose, a matrix which shows the distribution of all documents between each created cluster and dataset topics is created. The matrix's dimension is $k \times l$, which k is the number of clusters and l is the number of topics. This matrix is called Document Distribution Matrix (DDM). The maximum value at each column of DDM is found first. Then, the corresponding document topic related to this column is assigned to the NMF cluster related to the row number. At the end of this process, there may be some NMF clusters which are not assigned to any topic. Some of these clusters may contain large number of documents, and omitting them may reduce the accuracy metric. To assign these NMF clusters to a topic, the maximum value found in a row of DDM related to any of these NMF clusters is used for the topic assignment. It turns out that the related column of the founded value indicates the topic to be assigned. This method may results in assigning some of NMF clusters to more than one topic.

We evaluate the clustering performance with Accuracy, Root Mean Square Residual (RMSR), Entropy, Purity, and computational time metrics. Accuracy of clustering is assessed using the metric AC used by [4] is defined

$$AC = \sum_{i=1}^{n} \delta(d_i)/n \tag{15}$$

where:

$\delta(d_i)$ is set to 1 if d_i has the same topic label for both NMF cluster and the original topic, and otherwise set to 0,

n is the total number of documents in the collection.

The RMSR between A and $W$and H matrix is defined as:

$$RMSR = \sqrt{\frac{\sum_{ij}\left(A_{ij} - WH_{ij}\right)^2}{m * n}} \tag{16}$$

Total entropy for a set of clusters is calculated as the weighted mean of the entropies of each cluster weighted by the size of each cluster [8]. Using DDM, we compute p_{ij} for topic j, the probability that a member of cluster i belongs to topic j as $p_{ij} = n_{ij}/n_i$, where n_i is the number of objects in cluster i and n_{ij} is the number of documents of topic j in cluster i. Entropy of each cluster is defined as:

$$e_i = -\sum_{j=1}^{l} p_{ij} \log_2\left(p_{ij}\right) \tag{17}$$

where:

l is the number of topics.

Entropy of the full data set as the sum of the entropies of each cluster weighted by the size of each cluster:

$$e = \sum_{i=1}^{k} \frac{n_i}{n} e_i \tag{18}$$

where:

k is the number of NMF clusters,

n is the total number of documents.

Purity measures the extent to which each NMF cluster contained documents from primarily one topic [16]. *Purity* of a NMF clustering is obtained as a weighted sum of individual NMF cluster Purity values and is given by

$$P(S_i) = \frac{1}{n_i} max_j\left(n_i^j\right) \tag{19}$$

$$Purity = \sum_{i=1}^{K} \frac{n_i}{n} P(S_i) \tag{20}$$

where:

$S_i$is a particular NMF cluster of size n_i,
$n_i^j$is the number of documents of the $i-th$ topic that were assigned to the $j-th$ NMF cluster,
$k$is the number of clusters,
$n$is the total number of documents.

In general, the larger the Purity value, the better the clustering solution. We also compute the computational time taken by each minimization algorithms in terms of CPU time measured in second.

For performance evaluation of MACB, the results of this algorithm were compared to Steepest Descent (SD) and L-BFGS methods of gradient descent algorithm implemented in MATLAB [18], and robust Correntropy Induced Metric (rCIM) [26]. For each algorithm, three clustering experiments were executed based on normalization of W and H matrices. As mentioned before, NMF does not have a unique solution, and it is better to normalize either W or H to have a consistent factorization of a particular dataset when using different algorithms. This procedure is also taken to investigate the effect of normalization of these W and H matrices on the clustering result. Therefore, we implement three experiments for each algorithm, one without normalization, another using normalization of W matrix's columns, and the last one with normalization on each row of H matrix.

Since σ value has an effect on update learning rate of SD, L-BFGS and rCIM algorithms, improper selection of σcould result in poor clustering. However, σ value have a small effect on MACB update algorithm, because the effect of σ is significantly decreased by the division in formula of MACB algorithm. Moreover, the learning rate is adaptive and is proportional to W and H matrices in each step of MACB algorithm. By implementing several experiments, we realize that the best value which yields the highest AC, lowest Entropy and highest Purity in clustering for each algorithm is $\sigma = 1$. We continue the experiment with three methods of normalization for MACB algo-

rithm and compare them to W-normalized case (normalization on each column of W matrix) for SD, L-BFGS, and rCIM algorithms with $\sigma = 1$ for three algorithms of optimization. AC, Entropy and Purity of clustering are shown in Figure 1-3 respectively,

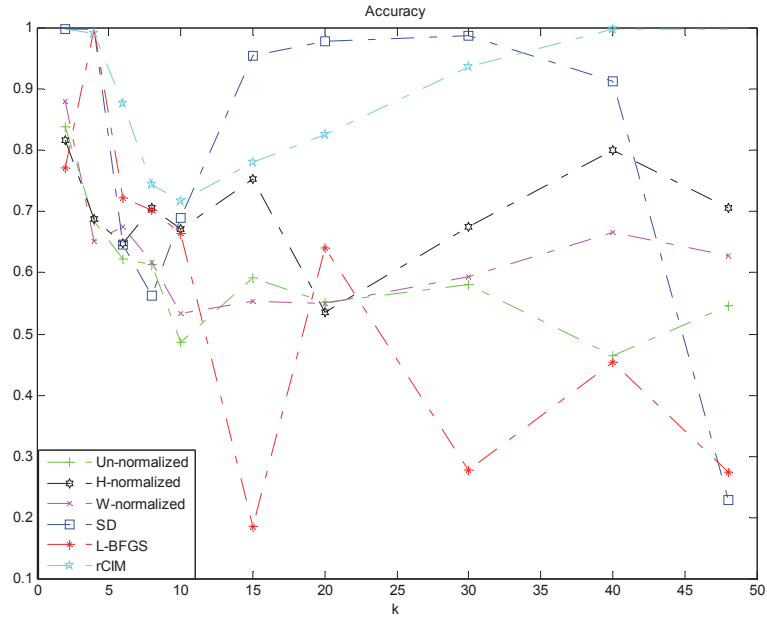

Figure 1. Accuracy of SD, L-BFGS, rCIM, and MACB algorithm

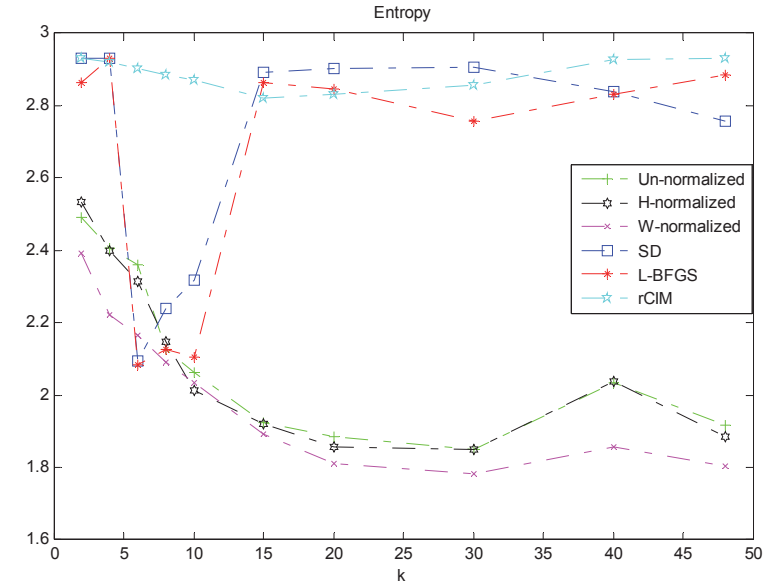

Figure 2. Entropy of SD, L-BFGS, and MACB algorithm

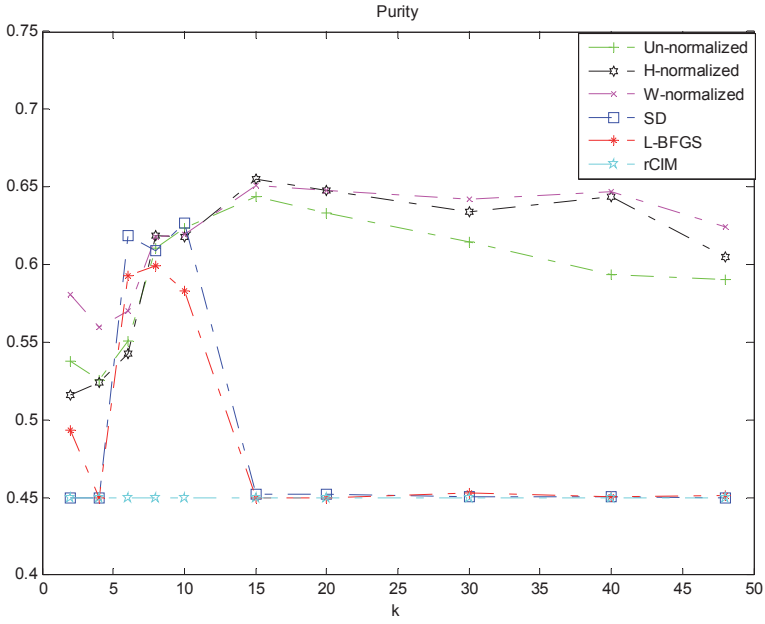

Figure 3. *Purity* of SD, L-BFGS, and MACB algorithm

It is clear that MACB algorithm yields smaller Entropy and higher Purity for all values of k. However, SD, L-BFGS, and rCIM algorithms have low Entropy and high Purity only for $k = [6,8,10]$. On the other hand, MACB have a consistent change in AC, Entropy, and Purity for different values of k. Moreover, as k increase, the quality of clustering improves for MACB. To have a good comparison between all algorithms, we select two values of k which results in highest AC, lowest Entropy and highest Purity. According to Fig.1-3, these metrics occurs in $k = [15, 19]$. Therefore we tabulate the clustering result of each algorithm for corresponding k values in Table 1 and 2.

Tables 1 and 2 indicate that MACB algorithm give better Entropy and Purity in comparison to the other algorithms. The RMSR metric is also small for MACB algorithm, while this metric is too large for SD, L-BFGS and rCIM. This indicates a large error between WH and A. One may notice that the computational time of MACBand rCIM algorithms is higher than SD and L-BFGS algorithms. The reason is that in each step of algorithm, there are two multiplications and divisions for updating W and H in MACB and rCIM algorithms, which do not exist in SD and L-BFGS algorithms. The multiplication and division of these large matrices are highly computational and time consuming.

As a result, we can conclude that the computed W and H matrices using MACB algorithm offer the best approximation of documents dataset among other correntropy-based NMF. The minimization of correntropy cost function for 40 iterations is shown in Fig.4 for all algorithms. It demonstrates that MACB algorithm has a faster convergence than SD, L-BFGS and rCIM algorithms. Gradient minimization curve for $k = 20,30,40,48$ is shown in Figure 5. It indicates that as the value of k increases, the gradient minimizes more slowly. This implies that the algorithm reaches the limit point of feasible region, and the constraint of nonnegativity does not allow the optimization algorithm to converge. We propose that other algorithms like alternating least square method with nonnegativity constraint and hierarchical ALS could be investigated on this case for future work.

Table 1. Comparison between performance of different NMF algorithms, k=15

Algorithm	**RMSR**	**Accuracy**	**Entropy**	**Purity**	**CPU time (sec)**
SD	1983	0.9401	2.8834	0.4582	552
L-BFGS	2517	0.1469	2.8634	0.4496	602
MACB (W-normalized)	0.3328	0.5530	1.8920	0.6514	2353
MACB (H-normalized)	0.3328	0.7528	1.9191	0.6551	2353

Table 2. Comparison between performance of different NMF algorithms, k=20

Algorithm	RMSR	Accuracy	Entropy	Purity	CPU time (sec)
SD	53594	0.8961	2.8616	0.4527	535
L-BFGS	17.75	0.6274	2.8399	0.4496	605
Multiplicative (W-normalized)	0.9776	0.5507	1.8094	0.6475	2513
Multiplicative (H-normalized)	0.9776	0.5360	1.8567	0.6479	2513

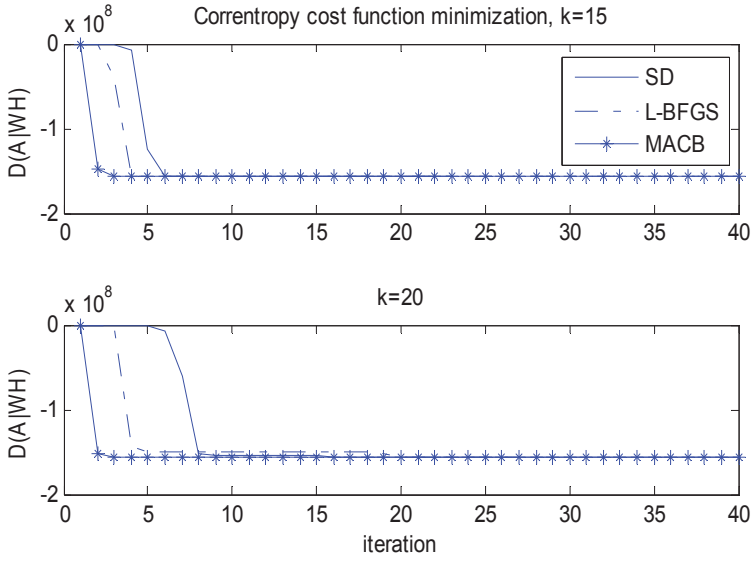

Figure 4. Correntropy cost function minimization curve

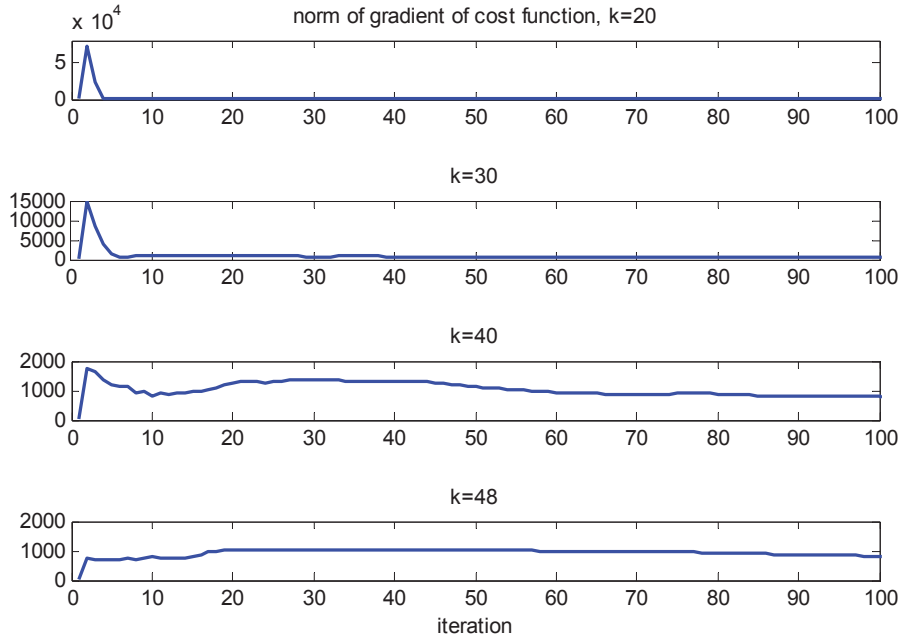

Figure 5. History of norm of cost function'sgradient

6 Conclusion

In this paper, a multiplicative algorithm for NMF based on correntropy cost function is developed. Its performance was tested on the Reuters Document Corpus for document clustering. The clustering result is also compared to gradient descent algorithm using SD and L-BFGS algorithms using common clustering evaluation measures. The minimization curve and curve of gradient's norm of cost function are also investigated. The result proves that MACB algorithm gives better clustering performance in terms of Entropy and Purity and also faster convergence than other two methods. However, it shows that by increasing the number of NMF clusters (k value), gradient curve of cost function does not converge appropriately. For future work, we propose that other minimization algorithms like ALS, ANLS, and HALS could be used for improving this problem.

References

1. Lee D.D.,Seung H.S., 1999,*Learning the parts of objects by non-negative matrix factorization*, Nature, 401, 6755, pp. 788-791.
2. Seung D.,Lee L., 2001,*Algorithms for non-negative matrix factorization,* Advances in neural information processing systems, 13, pp. 556-562.
3. Hoyer P.O., 2002, *Non-negative sparse coding,* Proc. of 12th IEEE Workshop on Neural Networks for Signal Processing, pp. 557-565.
4. Xu W., Liu X., Gong Y., 2003, *Document clustering based on non-negative matrix factorization*, Proc. of the 26th Annual Int. ACM SIGIR Conf. on Research and development in informaion retrieval, pp. 267-273.
5. Hoyer P.O., 2004,*Non-negative matrix factorization with sparseness constraints*, The Journal of Machine Learning Research, 5, pp. 1457-1469.
6. Pauca V.P., Shahnaz F., Berry M.W., PlemmonsR.J., 2004,*Text mining using non-negative matrix factorizations,* Proc. SIAM Int. Conf. on Data Mining, Orlando FL, pp. 22-24.
7. Sra S., DhillonI.S., 2005, *Generalized nonnegative matrix approximations with Bregman divergences*, Advances in neural information processing systems, pp. 283-290.
8. Tan P.N., Steinbach M., Kumar V., 2006,*Introduction to Data Mining*, Pearson Addison Wesley.
9. Shahnaz F., Berry M.W., Pauca V.P., PlemmonsR.J., 2006, *Document clustering using nonnegative matrix factorization*, Information Processing & Management, 42, 2, pp. 373-386.
10. Liu W., PokharelP.P., Principe J.C., 2006,*Correntropy: A localized similarity measure*, Int. Joint Conf. on Neural Networks, pp. 4919-4924.
11. Zdunek R., Cichocki A., 2006,*Non-negative matrix factorization with quasi-Newton optimization*, Int. Conf. on Artificial Intelligence and Soft Computing, Springer Berlin Heidelberg, 4029, pp. 870-879.
12. Berry M.W., Browne M., LangvilleA.N., Pauca V.P., PlemmonsR.J., 2007, *Algorithms and applications for approximate nonnegative matrix factorization*, Computational Statistics & Data Analysis, 52, 1, pp. 155-173.
13. KompassR., 2007,*A generalized divergence measure for nonnegative matrix factorization*, Neural computation, 19, 3, pp. 780-791.
14. Lin C.J., 2007, *Projected gradient methods for nonnegative matrix factorization*, Neural computation, 19, 10, pp. 2756-2779.
15. Liu W., PokharelP.P., Príncipe J.C., 2007, *Correntropy: properties and applications in non-Gaussian signal processing*, IEEE Trans. on Signal Processing, 55, 11, pp. 5286-5298.
16. Ding C., Li T., Peng W., Park H., 2006, *Orthogonal nonnegative matrix t-factorizations for clustering*, Proc. of the 12th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining, ACM, pp. 126-135.
17. Kim H.,Park H., 2008,*Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method*, SIAM Journal on Matrix Analysis and Applications, 30, 2, pp. 713-730.
18. Matlab Software by Mark Schmidt, www.di.ens.fr/~mschmidt/Software/minConf.html

19. Cichocki A.,Anh-HuyP., 2009, *Fast local algorithms for large scale nonnegative matrix and tensor factorizations*, IEICE Trans. on fundamentals of electronics, communications and computer sciences,92, 3, pp. 708-721.
20. Kim J.,Park H., 2011, *Fast nonnegative matrix factorization: An active-set-like method and comparisons*, SIAM Journal on Scientific Computing, 33, 6, pp. 3261-3281.
21. FévotteC., BertinN., DurrieuJ.L., 2011,*Nonnegative matrix factorization with the itakura-saito divergence: With application to music analysis*, Neural computation, 21, 3, pp. 793-830.
22. FévotteC., IdierJ., 2011, *Algorithms for nonnegative matrix factorization with the β-divergence*, Neural Computation, 23, 9, pp. 2421-2456.
23. He R., Zheng W.S., Hu B.G., 2011, *Maximum correntropy criterion for robust face recognition*, IEEE Trans. on Pattern Analysis and Machine Intelligence, 33, 8, pp. 1561-1576.
24. Ensari T., Chorowski J., Zurada J.M., 2012, *Correntropy-Based document clustering via nonnegative matrix factorization*, Artificial Neural Networks and Machine Learning–ICANN 2012, Springer Berlin Heidelberg, pp. 347-354.
25. Ensari T., ChorowskiJ., Zurada J.M., 2012,*Occluded Face Recognition Using Correntropy-Based Nonnegative Matrix Factorization*, 11th International Conference on Machine Learning and Applications (ICMLA), 1, pp. 606-609.
26. Du L., Li X., Shen Y.D., 2012, *Robust Nonnegative Matrix Factorization via Half-Quadratic Minimization*, IEEE 12th International Conference on Data Mining (ICDM), pp. 201-210.

COMPUTER MODELING OF SUPERCAPACITOR WITH COLE-COLE RELAXATION MODEL

Marek Orzyłowski[1], Mirosław Lewandowski[2]

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Łódź, Poland
marek.orzylowski@gmail.com

[2] Department of Electrical Engineering
Warsaw University of Technology
miroslaw.lewandowski@ee.pw.edu.pl

Abstract

Electric energy stored insupercapacitors is associated with ion movement between the porous electrodes . This phenomenon can be described by dielectric relaxation model. Cole-Davidson relaxation model application reported in publications is difficult to use for control purposes. In the paper for impedance of the supercapacitors description Cole-Cole relaxation model is applied. For impedance parameters identification Nedler-Mead simplex method is used. Supercapacitor impedance model simplification based on physical properties is presented. Such model can be easy used for calculations in Matlab environment with FOTF toolbox designed to fractional calculus. The example of modeling of dynamic system with supercapacitor impedance model is described. The effects of the simulation show that fractional model of superapacitors is important tool for exact description of its dynamics.

Key words: Supercapacitor modeling, Cole-Cole relaxation model, fractional calculus, control systems

1 Introduction

Supercapacitors are electronic elements having the properties between electrolytic capacitors and accumulators. Capacitance of the supercapacitors reaches several thousands of farads. They can reach energy and power densities of more than 10 Wh/kg and 10 kW/kg respectively. The possibility of large electric charge storage is obtained due to porous electrodes made of active carbon, graphene, carbon nanotubes or aerogel. Supercapacitors are used in many applications: for protection of computers from input power in-

terruptions, as power supply of robots, toys, electric toothbrushes etc. Recently they are increasingly used in electric vehicles for braking energy storage and its delivery during acceleration.

Electric energy stored insupercapacitors is associated with ion movement between the porous electrodes of large surface and relatively large resistance. This phenomenon causes that the typical equivalent models of capacitors that contain one or two lumped parameter RC circuits are not sufficient for accurate representation of dynamic properties of the supercapacitors. In the result, for this purpose, the complex equivalent schemes with many connected RC elements [1] or fractional differential equations [2, 3] are used.

In the paper, for impedance of the supercapacitors description fractional order calculus and model of dielectric relaxation are applied. Dielectric relaxation can be described by few models [4]. It was reported that Cole-Davidson model application is well for exact modeling of the supercapacitors [4, 5, 6] but its application in automation is difficult. The paper presents Cole-Cole model application for such purposes.

2 Cole-Cole and Cole-Davidson models of supercapacitor impedance

Classic Debye model of ideal dielectric relaxation is in practice replaced by its empiric modifications [4]. Such modification is presented by Havriliak-Negami model of complex dielectric constant, expressed as equation

$$\varepsilon_{HN}(j\omega) = \varepsilon_\infty + \frac{\varepsilon_S - \varepsilon_\infty}{\left[1+(j\omega T)^\delta\right]^\gamma}, \quad 0 < \gamma \leq 1 \qquad 0<\delta \leq 1, \tag{1}$$

where

ε_∞ – infinite frequency dielectric constant,
ε_s – static frequency dielectric constant,
T–characteristicrelaxation time of the medium.

For γ=1 equation (1) becomes Cole-Cole equation

$$\varepsilon_{CC}(j\omega) = \varepsilon_\infty + \frac{\varepsilon_S - \varepsilon_\infty}{1+(j\omega T)^\delta}, \qquad \text{where} \quad 0 < \delta \leq 1 \tag{2}$$

and for δ=1 it becomes Cole-Davidson equation

$$\varepsilon_{CD}(j\omega) = \varepsilon_\infty + \frac{\varepsilon_S - \varepsilon_\infty}{(1+j\omega T)^\gamma}, \qquad \text{where} \quad 0 < \gamma \leq 1 \tag{3}$$

Parameters δ and γ are determined experimentally.

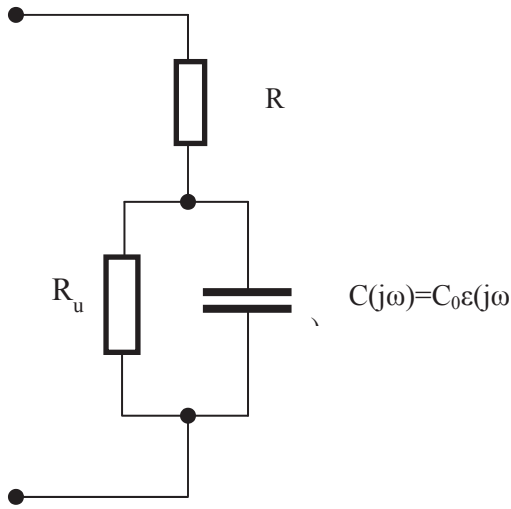

Figure1. Equivalent circuit of supercapacitor

The expression of the real supercapacitor impedance can be based on one of above equations of complex dielectric constants but it should also contain parallel leakage resistance R_u and serial equivalent resistance R_c (Figure 1) [5, 6]. As a result supercapacitor impedance is given by equation

$$Z(j\omega) = R_c + \frac{R_u \frac{1}{j\omega C(j\omega)}}{R_u + \frac{1}{j\omega C(j\omega)}} \tag{4}$$

where capacitance C(jω) is proportional to complex dielectric constant (1). Additionally, for the supercapacitors, can by assumed that

$$\varepsilon_\infty \ll \varepsilon_S \tag{5}$$

Let us replace Fourier transform with Laplace transform. Impedance of supercapacitor Z(s) can be treated as fractional transfer function G(s) with current input signal transform I(s) and voltage output signal transform V(s). On the basis of Cole-Davidson model (3), equations (4) and (5) one obtains the expression of supercapacitor impedance [5, 6]

$$G_{CD}(\mathrm{s}) = \frac{\mathrm{V(s)}}{\mathrm{I(s)}} = \mathrm{R_c} + \frac{R_u \dfrac{(1+\mathrm{sT})^{\gamma}}{sC}}{R_u + \dfrac{(1+\mathrm{sT})^{\gamma}}{sC}} = \frac{\left(1+\dfrac{R_c}{R_u}\right)(1+Ts)^{\gamma} + sR_cC}{\dfrac{1}{R_u}(1+sT)^{\gamma} + sC} \tag{6}$$

Transfer function is commonly in automation presented as [2, 3]

$$G(s) = \frac{b_0 s^{\beta_0} + b_1 s^{\beta_1} + \cdots + b_{m-1} s^{\beta_{m-1}} + b_m s^{\beta_m}}{a_0 s^{\propto_0} + a_1 s^{\propto_1} + \cdots + a_{n-1} s^{\beta_{n-1}} + a_n s^{\propto_n}} \tag{7}$$

Such a form of fractional transfer function can be directly used for calculation e.g. applying numerical computing environment Matlab with FOTF toolbox [7, 8] designed for fractional calculus.

Unfortunately equation (6) can't be directly expressed in form (7) because of presence binomial to a fractional power γ [6]. The same complications are connected with Havriliak-Negami model.

To avoid that issue one can apply Cole-Cole model of dielectric relaxation given by expression (2). Using the same transformation as for Cole-Davidson model, one can obtain equation

$$G_{CC}(\mathrm{s}) = \frac{\left(1+\dfrac{R_c}{R_u}\right) + s^{\delta}\left(1+\dfrac{R_c}{R_u}\right)T^{\delta} + sR_cC}{\dfrac{1}{R_u} + s^{\delta}\dfrac{T^{\delta}}{R_u} + sC} \tag{8}$$

Taking into consideration parameters of the supercapacitor equation (8) can be simplified. At the beginning it is worth to notice that serial resistance Rc is several order of magnitude lower than parallel leakage resistance Ru

$$\frac{R_c}{R_u} \ll 1 \tag{9}$$

This inequality leads to expression

$$G_{CC}(\mathrm{s}) = \frac{\left(1+\dfrac{R_c}{R_u}\right) + s^{\delta}\left(1+\dfrac{R_c}{R_u}\right)T^{\delta} + sR_cC}{\dfrac{1}{R_u} + s^{\delta}\dfrac{T^{\delta}}{R_u} + sC} \cong \frac{1 + s^{\delta}T^{\delta} + sR_cC}{\dfrac{1}{R_u} + s^{\delta}\dfrac{T^{\delta}}{R_u} + sC} \tag{10}$$

Generally transfer function (10) can be written in form

$$G_{CC}(\mathrm{s}) = \frac{1 + s^{\delta}T^{\delta} + sR_cC}{\frac{1}{R_u} + s^{\delta}\frac{T^{\delta}}{R_u} + sC} = \frac{1 + b_1 s^{\delta} + b_2 s}{a_0 + a_1 s^{\delta} + a_2 s} \tag{11}$$

which corresponds to (7).

$R_u$value can be determined from supercapacitor self-discharge curve. As a result the value of a0 coefficient is known

$$a_0 = \frac{1}{R_u} \tag{12}$$

Taking into account the value of a_0 and the following equality

$$b_1 = T^{\delta} \tag{13}$$

it can be written that

$$a_1 = \frac{T^{\delta}}{R_u} = a_0 b_1 \tag{14}$$

Summarizing, one can find that omitting R_c for determination of model (11) only 4 parameters should be identified:a_2, b_1, b_2 and δ. This identification can be based on the measurements of complex impedance values for the appropriate frequency range.

Identification of model (11) parameters can be performed on basis of minimization of performance index

$$J_f = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\left|G_{CC}(j\omega_i) - G_p(j\omega_i)\right|}{\left|G_p(j\omega_i)\right|}\right)^2 \tag{15}$$

where

G_{CC} – transfer function (11),
G_p – measured frequency response of the supercapacitor,
ω_i – frequency of measured point.

Chosen performance index corresponds to the variance of moduli of relative errors of the frequency response points, related to appropriate points of approximation function (11). For minimization purpose Nelder-Mead simplex method was used. This optimization problem is multi-modal so proper start point should be chosen. Fortunately the coefficients in expression (11) can be roughly estimated on the basis of estimation of supercapacitor physical parameters.

Measured frequency responses of supercapacitors presented in the paper, are based on data published in [5, 9, 10]. The example of transfer function calculated for 2700 F supercapacitor using data [10] is

$$G_{CC}(s) = \frac{1 + 0.869s^{0.846} + 0.632s}{0.00200 + 0.00174s^{0.846} + 2020s} \tag{16}$$

The result of the approximation of the frequency response (16) is presented in Figure 2. Another example is the impedance of the supercapacitorof 0.047 F capacitance [5]. Its transfer function is

$$G_{CC}(s) = 1000\frac{1 + 2.44s^{0.735} + 1.65s}{0.010 + 0.024s^{0.735} + 58.7s} \tag{17}$$

The frequency diagram of (17) is shown in Figure 3.

The basis for comparison of the accuracy of approximation for different supercapacitors can be performance index J_f (15).The square root of J_f corresponds to standard deviation of the error. For supercapacitors taken into consideration standard deviation of error is equal a few percent.

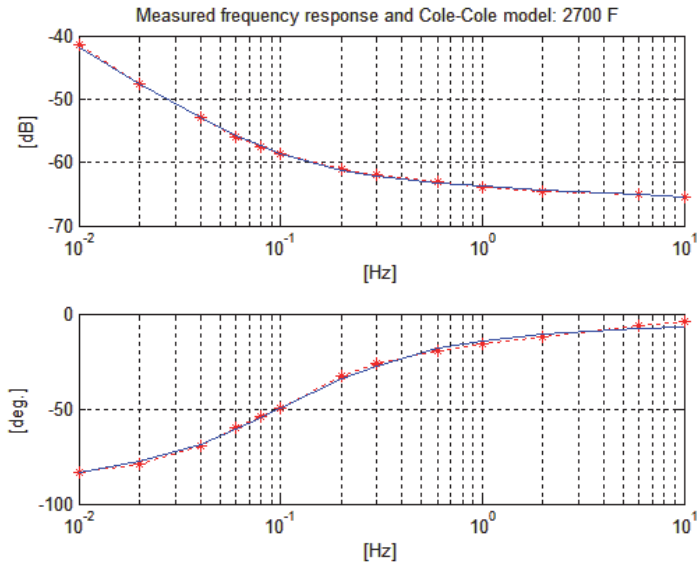

Figure 2. Measured frequency response points (asterisks) and approximating function (16) for 2700 F supercpacitor

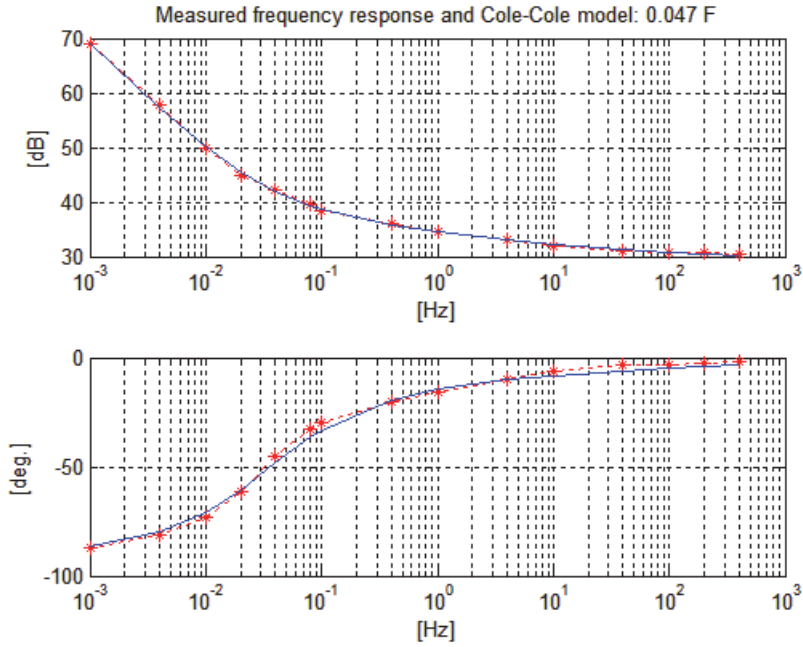

Figure 3. Measured frequency response points (asterisks) and approximating function (17) for 47 mF supercapacitor

3 Cole-Cole model simplification and time response

On the basis of the results of the impedance approximation of supercapacitors of capacitance between 0.047 F and 2700 F it can be stated that for all those examples model (11) can be simplified. The denominator of expression (11) can be written as

$$G_{CCd} = G_{CCd1} + G_{CCd2} \tag{18}$$

where

$$G_{CCd1} = a_0 + a_2 s \tag{19a}$$

$$G_{CCd2} = a_1 s^{\delta} \tag{19b}$$

It was proved that the ratio of

$$S(\omega) = \frac{|G_{CCd2}(\omega)|}{|G_{CC1}(\omega)|} \ll 1 \tag{20}$$

which means that the term G_{CCd2} practically has no influence on frequency response of the supercapacitor. In Figure 4 are shown graphs of S(ω) for various supercapacitors which frequency responses are presented in [5, 10].

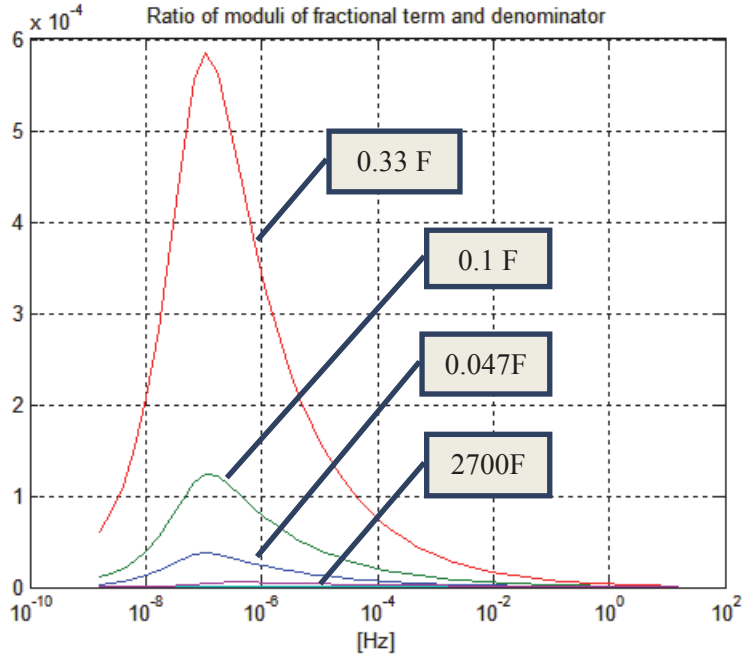

Figure 4. Frequency dependence of ratio S (18) for various supercapacitors

It can be mentioned that S(ω) strongly depends on exponent δ value. Typical value of δ for the capacitors is between 0.5 and 0.9. Graph of S(ω) for 0.6 F supercapacitor [10] is presented in Figure 5. Identified value of δ for this supercapacitor is 0.82. Other plots were calculated for hypothetical cases with lower values of δ.

Basing on current analysis one can determine the simpler model of the impedance of the supercapacitor. Omitting term G_{CCd2} the simplified expression is given as

$$G_{CC}(\mathrm{s}) = \frac{1 + b_1 s^{\delta} + b_2 s}{a_0 + a_2 s} \tag{21}$$

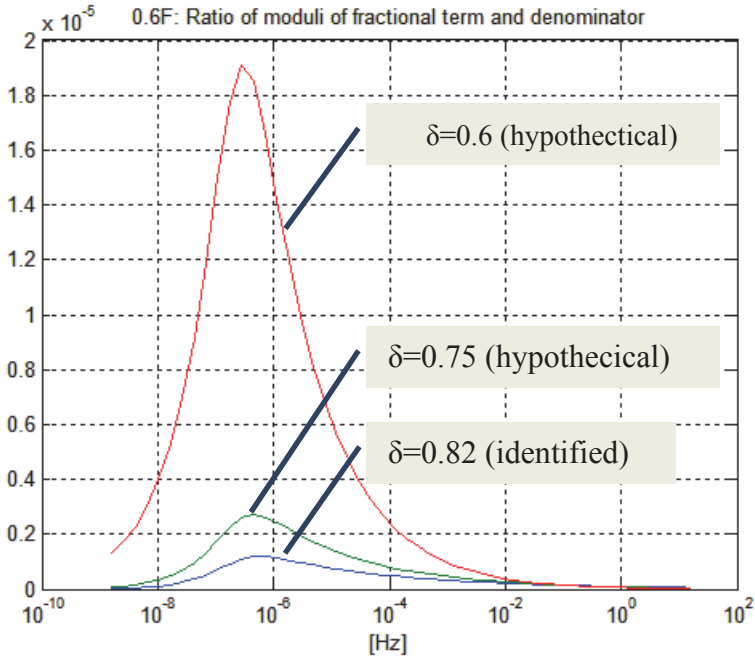

Figure 5. Ratio S(ω) for supercapacitor 0.6F

Consequently the impedance of e.g. 0.33 F supercapacitor [5] can be written as

$$\mathrm{G_{CC}(s)} = \frac{1 + 13.5\mathrm{s}^{0.670} + 0.632\mathrm{s}}{1.65\mathrm{e} - 07 + 0.340\mathrm{s}} \tag{22}$$

For the further analysis expression (21) can be decomposed into three simple fractions

$$G_{cc}(s) = \frac{1 + b_1 s^{\delta} + b_2 s}{a_0 + a_2 s} = G_{cc1}(s) + G_{cc2}(s) + G_{cc3}(s) \tag{23}$$

where

$$G_{cc1}(s) = R_c \tag{24a}$$

$$G_{cc2}(s) = \frac{R_u}{1 + sR_uC} \tag{24b}$$

$$G_{cc3}(s) = \frac{s^{\partial} T^{\partial} R_u}{1 + sR_u C} \tag{24c}$$

In Figure 6 are shown moduli of frequency responses of each term of (23) and modulus of $G_{cc.}$ The terms are asymptotes of $G_{cc}(s)$. The slope of logarithmic plots for G_{cc2} is -20 dB per decade of frequency and the slope of G_{cc3} is -20*(1-δ) dB per decade of frequency.

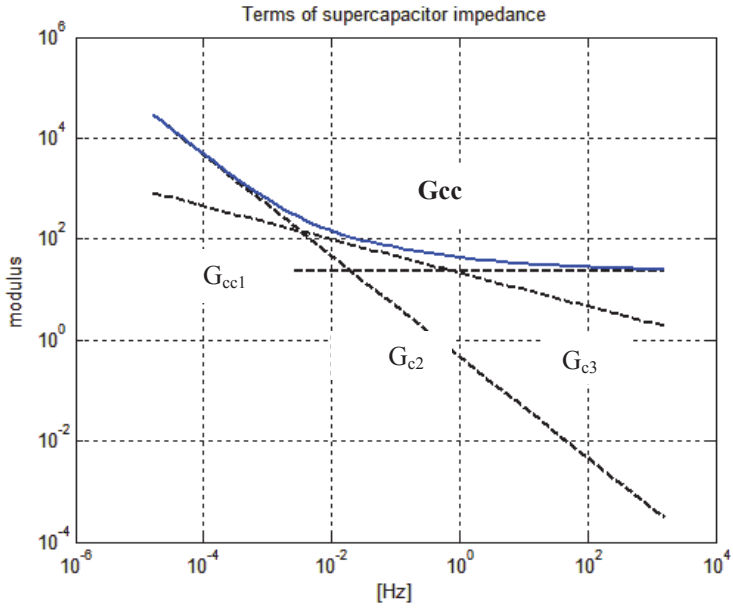

Figure6. Moduli of terms of equation (22) and modulus Gcc for supercapacitor 0.33 F

Voltage response of impedance (23) to current step is a sum of responses of mentioned 3 terms: proportional step, exponential response of large time constant R_uC, and response dependent on fractional order term. The voltage response for I_0 magnitude of current step can be written as

$$v_{cc}(t) = \mathcal{L}^{-1}\left\{\frac{I_0}{s}\left[R_c + \frac{R_u}{1 + sR_u C} + \frac{s^{\partial} T^{\partial} R_u}{1 + sR_u C}\right]\right\} = v_{cc1}(t) + v_{cc2}(t) + v_{cc3}(t) \tag{25}$$

where

$$v_{cc1}(t) = \mathcal{L}^{-1}\left\{\frac{I_0}{s}R_c\right\} = I_0 R_c \tag{26a}$$

$$v_{cc2}(t) = \mathcal{L}^{-1}\left\{\frac{I_0}{s}\left[\frac{R_u}{1+sR_uC}\right]\right\} = I_0 R_u\left[1 - exp\left(\frac{-t}{R_uC}\right)\right] \tag{26b}$$

$$v_{cc3}(t) = I\mathcal{L}^{-1}\left\{\frac{I_0}{s}\left[\frac{s^{\partial}T^{\partial}R_u}{1+sR_uC}\right]\right\}_0 = f_T(t, R_u, C, T, \partial) \tag{26c}$$

For time t<<R_uC the two first terms causes step summed with quasi-linear increase. The third term is responsible for initial non-linearity – Figure 7.

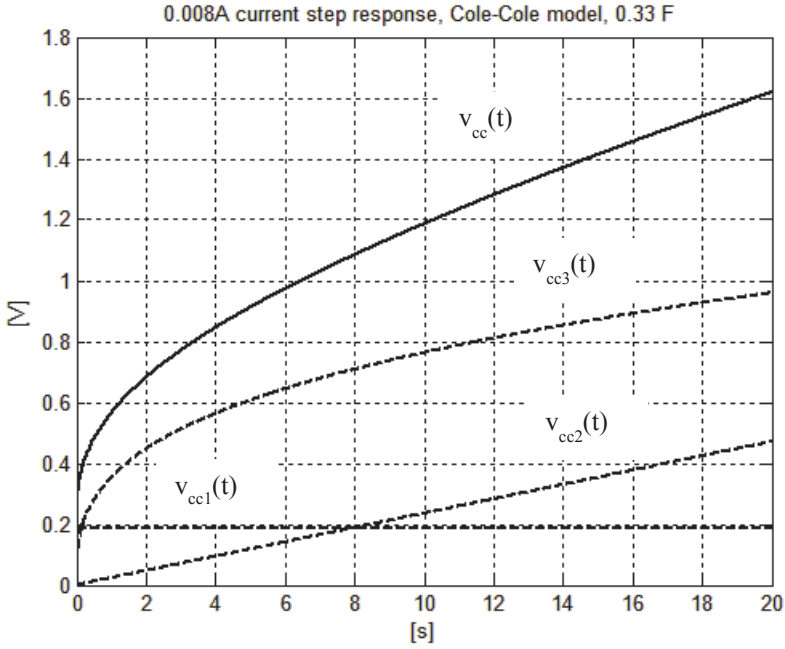

Figure 7. Current step response of supercapacitor of 0.33 F

4 Cole-Cole model application in control systems analysis

It has been mentioned that for fractional calculus the numerical computing environment Matlab with FOTF toolbox [7] can be applied. Matlab environment is well known and widely used tool for modeling and simulation of

physical systems. Using Control Toolbox one can study and design control systems. FOTF toolbox enables fractional calculus providing functions for:

- fractional transfer function object creation,
- presentation of Bode and Nyquist plots of this transfer function,
- calculation of time response on basis of transfer function and time input signal,
- addition, subtraction, multiplication and inversion of created models,
- feedback connection of such models,
- determination whether system is stable.

Presented example of FOTF toolbox application is design of resistor /capacitor voltage divider of inertial properties consisting of resistor R_0=5kΩ and supercapacitor of C=0.1F (Figure 8).

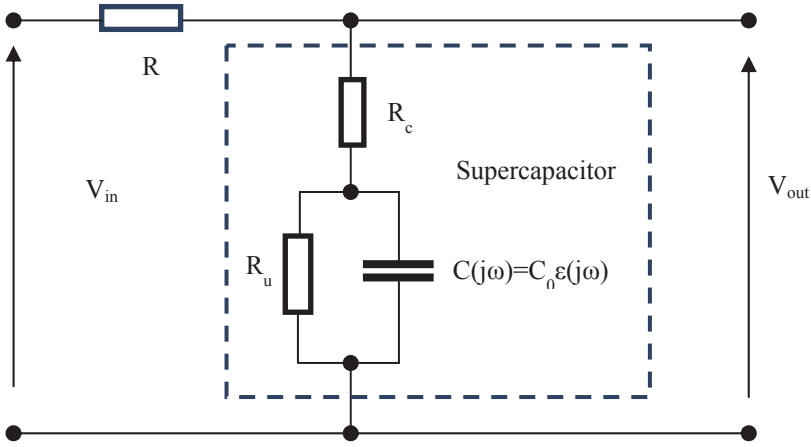

Figure 8. Scheme of the voltage divider

This divider shown in Figure 8 is described by equation

$$G_d(s) = \frac{G_{CC}(s)}{R_0 + G_{CC}(s)} = G_{CC}(s)\big(R_0 + G_{CC}(s)\big)^{-1} \tag{27}$$

where

$$\mathrm{G_{CC}(s)} = \frac{1 + 4.67\mathrm{s}^{0.705} + 5.01\mathrm{s}}{5\mathrm{e} - 08 + 0.1\mathrm{s}} \tag{28}$$

One can specify FOFT object for (28) and enter it into MATLAB. Vectors formulated according to form (7) are input parameters of such an object. They

contain coefficients a_i, b_i and exponents of s defined in (21). For (28) these vectors (in the reverse order) are equal

$$wa = [a_2 \quad a_0] = [0.1 \quad 5e-08] \tag{29a}$$

$$pa = [1 \quad 0] \tag{29b}$$

$$wb = [b_2 \quad b_1 \quad 1] = [5.01 \quad 4.67 \quad 1] \tag{29c}$$

$$pb = [1 \quad \delta \quad 0] = [1 \quad 0.705 \quad 0] \tag{29d}$$

In the next step the FOFT objects of (28) and R_o should be created

```
Gcc=fotf(wa,pa,wb,pb);
R0=fotf([[1],[0],[5000],[0]);
```

Then according to (26) these objects should be added

```
G1=plus(R0,Gcc);
```

inverted

```
G1i=inv(G1);
```

and multiplied

```
Gd=mtimes(Gcc,G1i);
```

The calculated transfer function of considered divider is equal to

$$G_d(s) = \frac{5e-08+2.34e-07s^{0.705}+0.1s+0.470s^{1.705}+0.504s^2}{5e-08+2.34e-07s^{0.705}+0.1s+0.470s^{1.705}+51s^2} \tag{30}$$

This transfer function has been compared with transfer function of divider with capacitor in which the relaxation phenomenon can be neglected. Impedance of such idealized capacitor of capacitance C_i=0.1 F is similar to (10)

$$G_i(s) = R_c + \frac{R_u \frac{1}{sC_i}}{R_{u+}\frac{1}{sC_i}} \cong \frac{1+sR_cC_i}{\frac{1}{R_u}+sC_i} = \frac{1+5\mathrm{s}}{5\mathrm{e}-08+0.1\mathrm{s}} \tag{31}$$

Transfer function of the divider with this capacitor is

$$G_{id}(s) = \frac{R_0}{R_0+G_i(s)} = \frac{R_0(1+SR_uC_i)}{R_0(1+SR_uC_i)+R_u(1+SR_cC_i)} = \frac{1+5s}{1+505s} \tag{32}$$

Bode plots of $G_d(s)$ and $G_i(s)$ are compared in Figure 9. The influence of relaxation phenomenon on frequency response is distinct for higher frequencies.

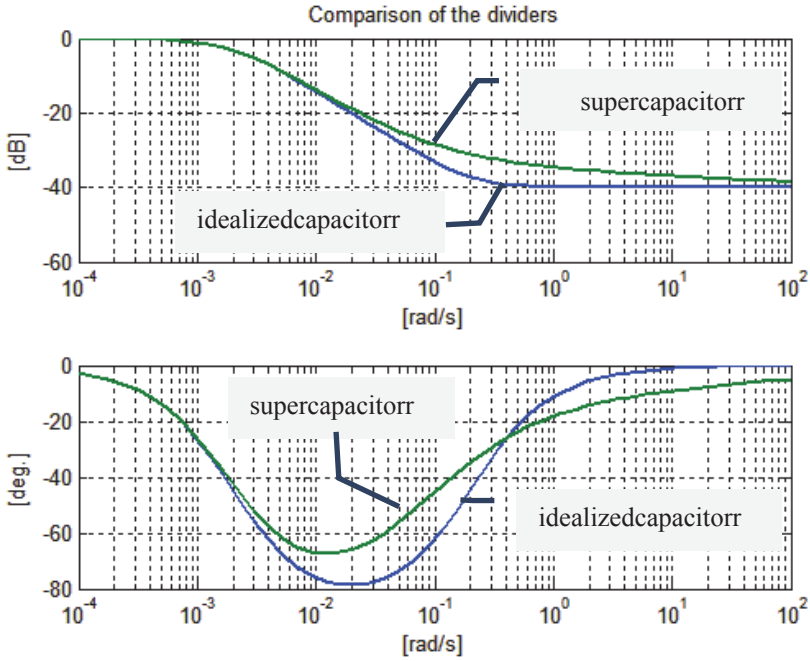

Figure 9. Bode plots of transfer functions of the dividers with supercapacitor and idealized capacitor

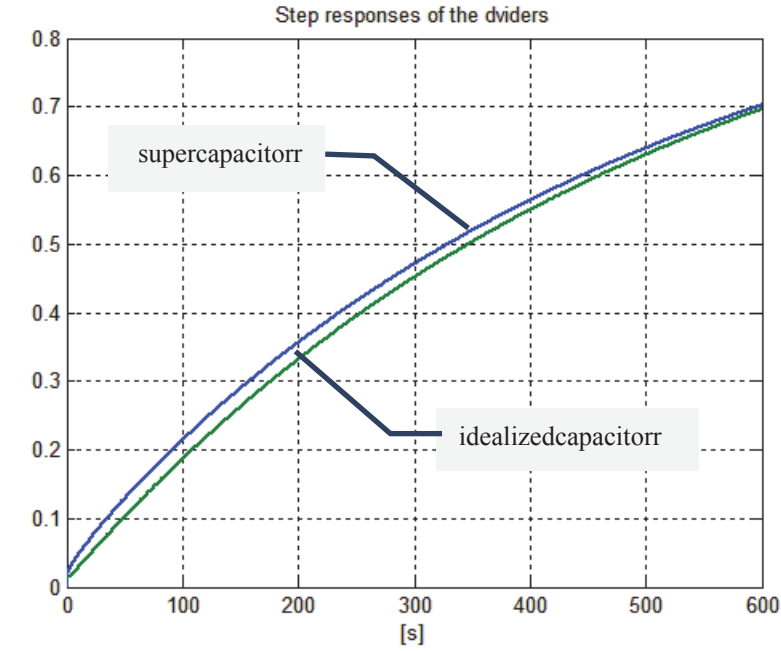

Figure 10. Step responses of the dividers

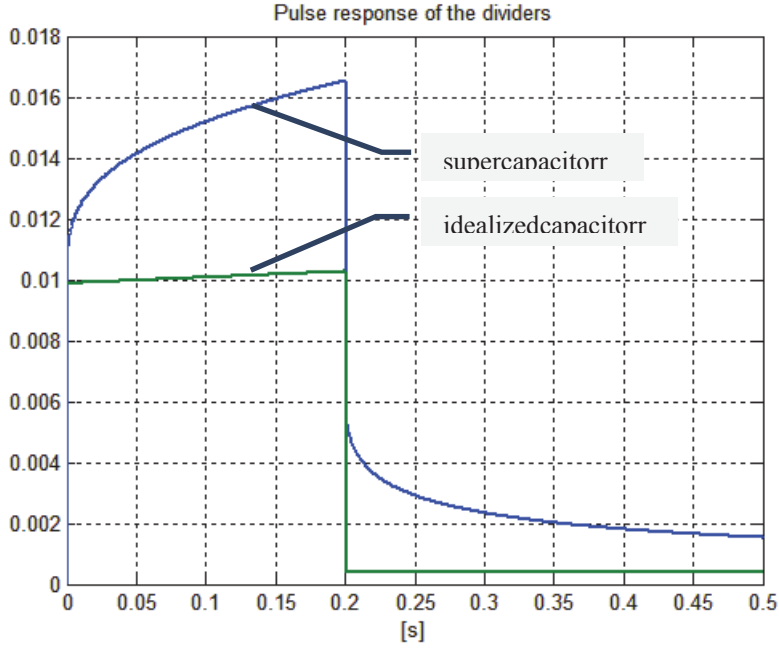

Figure 11. Pulse responses of the dividers

The step responses of both dividers are presented in Figure 10. They are convergent with the time rise. The essential difference at the beginning of the time responses is presented by the plots of time responses to short input pulse of 0.2 s duration (Figure 11).

Taking into account the difference between time and frequency responses of models of fractional and lumped parameters one can state that the fractional model of superapacitors can be important for exact description of its dynamics.

5 Conclusions

The technical literature mostly concerns the supercapacitor models with Cole-Davidson relaxation model application. In the paper the computer model of supercapacitor impedance based on Cole-Cole relaxation modelis presented. Consequently the impedance has polynomial form commonly used in automation. It enables the analysis of various control systems containing supercapacitors. For this purpose Matlab environment with FOTF toolbox designed to fractional calculus can be applied.

In the studied examples Cole-Davidson model in general is a bit more accurate for frequency and time responses of real supercapacitor approximation but advantages connected with easy analysis and simulation of control systems is essential. The comparison of practical effects of both relaxation models application in control systems analysis will be subject of the next publications.

In general it can be stated that the fractional model of superapacitors can be important tool for exact description of its dynamics.

References

1. Shi L., Crow M. L., *Comparison of Ultracapacitor Electric Circuit*,Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century, IEEEXplore,2008
2. Monje, C.A., Chen, Y., Vinagre, B.M., Xue, D., Feliu-Batlle, V.,*Fractional-Order Systems and Controls: Fundamentals and Applications*, Springer, 2010
3. Ostalczyk P., *Zarys rachunku różniczkowo-całkowego ułamkowych rzędów, Teoria i zastosowania w automatyce*, Wydawnictwo Politechniki Łódzkiej, 2008
4. Déjardin, J-L., Jadzyn J., *Determination of the nonlinear dielectric increment in the Cole-Davidson model*, The Journal of Chemical Physics 125, 114503, 2006.
5. Dzieliński A., Sierociuk D., Sarwas G., *Some Applications of Fractional Order Calculus*, Automatics, Bulletin of the Polish Academy of Sciences, Technical Sciences, vol. 58, No. 4, 2010

6. Dzieliński A., Sierociuk D., Sarwas G., *Time domain validation of ultracapacitor fractional order model,*49th IEEE Conference on Decision and Control, December 15-17, 2010, Hilton Atlanta Hotel, Atlanta, GA, USA
7. YangQuan Chen, Ivo Petras and DingyuXue, *Fractional Order Control – A Tutorial*, 2009 American Control Conference Hyatt Regency Riverfront, St. Louis, MO, USA, June 10-12, 2009
8. Matsu R., *Matlab Toolboxes for Fractional Order Control: an Overview*, Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium, Volume 22, No. 1, 2011
9. *HS206 supercapacitor Datasheet*, Rev 1.1, CAP-XX, 2008
10. *Ultracapacitors Data Sheets and technical information for PC2500TM*, Maxwell

IMPLEMENTATION OF THE WAVELET TRANSFORM WITH SSE EXTENSIONS

Tadeusz Łyszkowski[1], Tomasz Wiechno[2], Mykhaylo Yatsymirskyy[2]

[1]Higher Vocational State School in Wloclawek
tadeusz.lyszkowski@pwsz.wloclawek.pl

[2]Institute of Information Technology, Lodz University of Technology
tomasz.wiechn@p.lodz.pl, mykhaylo.yatsymirskyy@p.lodz.pl

Abstract

It has been shown that application of assembly implementation of Streaming SIMD Extensions (SSE) shortens the time needed to apply filtration in two-channel filter bank by tenfold, comparing to non-optimized version, written in Microsoft Visual C++ 2010 Express, without assembler extensions.

The implementation described in this paper can be applied to computation of Discrete Wavelet Transform on general-purpose processors..

Key words: Orthogonal Filters, Discrete Wavelet Transform, SSE extensions

1 Introduction

Discrete Wavelet Transform (DWT) is applied to data compression, system identification, signal approximation and interpolation, image processing and recognition as well as synthesis of digital watermarking [1-4].

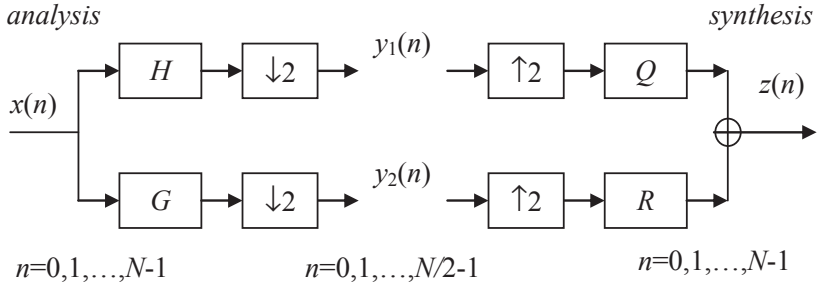

Figure 1. Diagram of one stage of analysis and synthesis of Discrete Wavelet Transform.

Because of such wide and profound applications, there is a lot of research on the improvements of Fast Computational Algorithms for the Discrete Wavelet Transform [5-10]. The construction of the algorithm is based on parallel or pyramidal repetition of basic analysis stage for forward transform and a basic synthesis stage for inverse transform. The two channel biorthogonal filter banks shown on Figure 1. [11] are a classic model of such a transform.

Blocks H, G, Q and R, are linear filters with finite impulse response $H = h_0,h_1,...,h_{K-1}$, $G = g_0,g_1,...,g_{K-1}$, $Q = q_0,q_1,...,q_{K-1}$ i $R = r_0,r_1,...,r_{K-1}$, where the length of the filter K is an even natural number. Blocks ↓2 and ↑2 denote, respectively, the operations of decimation in time of input sequence (downsampling) and upsampling by a factor of 2, i.e. inserting zeroes between each sample of a input sequence.The results of analysis stage of (forward) DWT can be expressed as two convolutions with decimation [12]

$$\begin{aligned} y_{2i} &= \sum_{k=0}^{K-1} h_{K,K-1-k}\, x_{2i+k}, \\ y_{2i+1} &= \sum_{k=0}^{K-1} g_{K,K-1-k}\, x_{2i+k} \qquad i = 0,1,...,N/2\text{-}1, \end{aligned} \tag{1}$$

where $h_{K,k}$ $g_{K,k}$ for $k = 0,1,...,K\text{-}1$ are impulse responses of filters H_K, G_K, and N is the length of input sequence.

If coefficients of impulse responses of filters H_K and G_K are written in reversed order:

$$h1_{K,k} = h_{K,K-1-k}, \quad g1_{K,k} = g_{K,K-1-k} \quad k = 0,1,...,K\text{-}1$$

formulas (1) can be rewritten in the form (2) that is more convenient for implementation

$$\begin{aligned} y_{2i} &= \sum_{k=0}^{K-1} h1_{K,k}\, x_{2i+k}, \\ y_{2i+1} &= \sum_{k=0}^{K-1} g1_{K,k}\, x_{2i+k} \qquad \text{for } i = 0,1,...,N/2\text{-}1. \end{aligned} \tag{2}$$

From (2) it is clear that the time needed for computation of DWT expressed as a convolution, depends on the effectiveness of floating point multiplications and additions. Exploiting Data Level Parallelism this can be enhanced by the usage of Streaming SIMD Extensions (SSE) available on contemporary general-purpose processors.

The paper describes construction of assembler implementations of DWT algorithms (2) that make use of SSE. The algorithms are given for a number of filter lengths $K = 6,8,10$ and 12 and the results are compared with the reference algorithms written in pure C++.

The problem solved in the paper is important as the majority of personal computers in use, is equipped with processors that are compliant with SSE rather than newer AVX extensions, introduced in 2011 [13].

2 SSE in IA-32 architecture

Beginning from the Pentium III processor the Streaming SIMD Extensions (SSE) were introduced to the IA-32 architecture. The SSE expands the SIMD execution model introduced with the Intel (Multimedia Extension) MMX technology by providing a new set of eight 128-bit registers *xmm*0, *xmm*1 ... *xmm*7 and the ability to perform (single-instruction, multiple-data) SIMD operations on four 32-bit packed single-precision floating-point values [13].The same operation can be performed at the same instruction cycle on four *float* elements stored in *xmm* register or in four array elements kept in memory.

Because of this parallelism in data processing, application of SSE Extensions can yield even fourfold performance gain comparing to a code that is non SSE aware. It is worth noting that data level parallelism reduces up to four times the number of instructions needed to write the algorithm.

3 Implementation of one stage of forward DWT computed as a convolution

Figure 2 shows the reference C++ implementation of DWT written according to the formula (2).

```
// DWT in C++
for (int i=0;i<N;i+=2)
{
     float t1=h1[0]*x[i], t2=g1[0]*x[i];
     for (int k=1;k<K;k++)
     {
       t1+=h1[k]*x[i+k];  t2+=g1[k]*x[i+k];
     }
     y[i]=t1;  y[i+1]=t2;
}
```

Figure 2. Algorithm of the one stage of forward DWT computed as a convolution in C++.

The algorithm needs *K* floating-point multiplications and *K-1* floating-point additions to compute one output element. However because of the data level parallelism it is possible to significantly shorten the time of this computation by the application of SSE extensions.To maximize performance gain, the whole algorithm has been programmed in assembly language.Furthermore, the inner loop that computes the sum of the product of input values times coefficients of impulse response (in reversed order), has been unfolded and optimized for the selected filter lengths, to shorten the most computation intensive part. The outer loop that contains mainly instructions for reading samples and writing output coefficients has been left intact.

Hence, further discussion in this section will concern major parts of the two assembler implementations of forward DWT for *N* being divisible by 4, namely: version A, for filers of length *K*=6 and 8, version B, for *K*=10 and 12 as well as some elements of version C, for *N* being even and *K*=6.

3.1 Version A. Implementation of DWT using assembler with SSE extensions

The implementation of this version, for filter length *K*=8 is shown on Figure 3. For the sake of clarity and speed of computation it has been assumed that the number of input samples *N* is divisible by 4. It is not really a constraint as, in majority of DWT applications, the length of input sequence is power of 2 with the exponent greater than 1. However, this makes it possible to compute and keep four output coefficients in *xmm* register as well as store them into the memory on every iteration of the loop.

In the discussed implementation there are eight steps. The first step shown on part a) of Figure 3. loads four input samples x_3, x_2, x_1, x_0 into the register *xmm0* and next four samples x_7, x_6, x_5, x_4 into the register *xmm1*. It is illustrated by the comments to the code, where four parts of the relevant register are shown for every instruction. In part b) registers *xmm*4, *xmm*5, *xmm*6 i *xmm*7 are loaded with coefficients of impulse responses *h1* and *g1* in reversed order.

```
mov ecx,0              ;(i=0)ecx=0
movaps xmm0,x[ecx]  ; xmm0=x3|x2|x1|x0
movaps xmm1,x[ecx+16]     ; xmm1=x7|x6|x5|x4
movaps buf1,xmm1    ; buf1=x7|x6|x5|x4
```

– Loading input data to the *xmm* registers

```
movaps xmm4,h1       ; xmm4=h13|h12|h11|h10
movaps xmm5,h1[16]   ; xmm5=h17|h16|h15|h14
movaps xmm6,g1       ; xmm6=g13|g12|g11|g10
```

```
  movaps xmm7,g1[16]  ; xmm7=g1_7|g1_6|g1_5|g1_4
```

– Loading parameters h1 and g1 to the xmm registers

```
iloop:
movaps xmm2,xmm0    ; xmm2=x_{i+3}|x_{i+2}|x_{i+1}|x_{i+0}
mulps xmm2,xmm4     ; xmm2=x_{i+3}*h1_3|x_{i+2}*h1_2|
                    ; x_{i+1}*h1_1|x_{i+0}*h1_0
movaps xmm3,xmm1    ; xmm3=x_{i+7}|x_{i+6}|x_{i+5}|x_{i+4}
mulps xmm3,xmm5     ; xmm3=x_{i+7}*h1_7|x_{i+6}*h1_6|
                    ; x_{i+5}*h1_5|x_{i+4}*h1_4
addps xmm2,xmm3     ; xmm2=x_{i+7}*h1_7+x_{i+3}*h1_3|
                    ; x_{i+6}*h1_6+x_{i+2}*h1_2|
                    ;x_{i+5}*h1_5+x_{i+1}*h1_1|x_{i+4}*h1_4+x_{i+0}*h1_0
movaps xmm3,zeros   ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3    ; xmm2=0.0|0.0|
                    ;x_{i+7}*h1_7+x_{i+3}*h1_3+x_{i+6}*h1_6+x_{i+2}*h1_2|
                    ; x_{i+5}*h1_5+x_{i+1}*h1_1+x_{i+4}*h1_4+x_{i+0}*h1_0
haddps xmm2,xmm3    ; xmm2=0.0|0.0|0.0|
                    ; x_{i+7}*h1_7+x_{i+6}*h1_6+x_{i+5}*h1_5+x_{i+4}*h1_4+
                    ; x_{i+3}*h1_3+x_{i+2}*h1_2+x_{i+1}*h1_1+x_{i+0}*h1_0
movaps buf5,xmm2    ; buf5=0.0|0.0|0.0|t1
```

– Computation of coefficient t1 of DWT (at that moment it is yi+0)

```
movaps xmm2,xmm0       ; xmm2=x_{i+3}|x_{i+2}|x_{i+1}|x_{i+0}
mulps xmm2,xmm6        ; xmm2=x_{i+3}*g1_3|x_{i+2}*g1_2|
                       ; x_{i+1}*g1_1|x_{i+0}*g1_0
movaps xmm3,xmm1       ; xmm3=x_{i+7}|x_{i+6}|x_{i+5}|x_{i+4}
mulps xmm3,xmm7        ; xmm3=x_{i+7}*g1_7|x_{i+6}*g1_6|
                       ; x_{i+5}*g1_5|x_{i+4}*g1_4
addps xmm2,xmm3        ; xmm2=x_{i+7}*g1_7+xi_{+3}*g1_3|
                       ; x_{i+6}*g1_6+x_{i+2}*g1_2|
                       ; x_{i+5}*g1_5+x_{i+1}*g1_1|
                       ; x_{i+4}*g1_4+x_{i+0}*g1_0
movaps xmm3,zeros      ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3       ; xmm2=0.0|0.0|
                       ; x_{i+7}*g1_7+x_{i+3}*g1_3+
                       ; x_{i+6}*g1_6+x_{i+2}*g1_2|
                       ; x_{i+5}*g1_5+x_{i+1}*g1_1+
                       ; x_{i+4}*g1_4+x_{i+0}*g1_0
haddps xmm2,xmm3       ; xmm2=0.0|0.0|0.0|
                       ; x_{i+7}*g1_7+x_{i+6}*g1_6+
```

```
                          ; x_{i+5}*g1_5+x_{i+4}*g1_4+x_{i+3}*g1_3+
                          ; x_{i+2}*g1_2+x_{i+1}*g1_1+x_{i+0}*g1_0
shufps xmm2,xmm2,11110011b ; xmm2=0.0|0.0|t2|0.0
addps xmm2,buf5           ; xmm2=0.0|0.0|t2|t1
movaps buf5,xmm2          ; buf5=0.0|0.0|y_{i+1}|y_{i+0}
```

– Computation of coefficient t2 of DWT (at that moment it is yi+1)

```
movaps xmm2,x[ecx+32]       ; xmm2=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
movaps buf0,xmm2            ; buf0=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
shufps xmm0,xmm1,01001110b ; xmm0=x_{i+5}|x_{i+4}|x_{i+3}|x_{i+2}
shufps xmm1,xmm2,01001110b ; xmm1=x_{i+9}|x_{i+8}|x_{i+7}|x_{i+6}
```

– Loading new input data to the xmm registers

```
movaps xmm2,xmm0          ; xmm2=x_{i+5}|x_{i+4}|x_{i+3}|x_{i+2}
mulps xmm2,xmm4           ; xmm2=x_{i+5}*h1_3|x_{i+4}*h1_2|
                          ; x_{i+3}*h1_1|x_{i+2}*h1_0
movaps xmm3,xmm1          ; xmm3=x_{i+9}|x_{i+8}|x_{i+7}|x_{i+6}
mulps xmm3,xmm5           ; xmm3=x_{i+9}*h1_7|x_{i+8}*h1_6|
                          ; x_{i+7}*h1_5|x_{i+6}*h1_4
addps xmm2,xmm3           ; xmm2=x_{i+9}*h1_7+x_{i+5}*h1_3|
                          ; x_{i+8}*h1_6+x_{i+4}*h1_2|
                          ; x_{i+7}*h1_5+x_{i+3}*h1_1|
                          ; x_{i+6}*h1_4+x_{i+2}*h1_0
movaps xmm3,zeros         ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3          ; xmm2=0.0|0.0|
                          ; x_{i+9}*h1_7+x_{i+5}*h1_3+
                          ; x_{i+8}*h1_6+x_{i+4}*h1_2|
                          ; x_{i+7}*h1_5+x_{i+3}*h1_1+
                          ; x_{i+6}*h1_4+x_{i+2}*h1_0
haddps xmm3,xmm2          ; xmm3=0.0|
                          ; x_{i+9}*h1_7+x_{i+8}*h1_6+
                          ; x_{i+7}*h1_5+x_{i+6}*h1_4+
                          ; x_{i+5}*h1_3+x_{i+4}*h1_2+
                          ; x_{i+3}*h1_1+x_{i+2}*h1_0|
                          ; 0.0|0.0
movaps buf6,xmm3          ; buf6=0.0|t1|0.0|0.0
```

– Computation of coefficient t1 of DWT (at that moment it is yi+2)

```
movaps xmm2,xmm0          ; xmm2=x_{i+5}|x_{i+4}|x_{i+3}|x_{i+2}
```

```
mulps xmm2,xmm6          ; xmm2=x_{i+5}*g1_3|x_{i+4}*g1_2|x_{i+3}*g1_1|
                         ; x_{i+2}*g1_0
movaps xmm3,xmm1         ; xmm3=x_{i+9}|x_{i+8}|x_{i+7}|x_{i+6}
mulps xmm3,xmm7          ; xmm3=x_{i+9}*g1_7|x_{i+8}*g1_6|x_{i+7}*g1_5|
                         ; x_{i+6}*g1_4
addps xmm2,xmm3          ; xmm2=x_{i+9}*g1_7+x_{i+5}*g1_3|
                         ; x_{i+8}*g1_6+x_{i+4}*g1_2|
                         ; x_{i+7}*g1_5+x_{i+3}*g1_1|
                         ; x_{i+6}*g1_4+x_{i+2}*g1_0
movaps xmm3,zeros        ; xmm3=0.0|0.0|0.0|0.0
haddps xmm2,xmm3         ; xmm2=0.0|0.0|
                         ; x_{i+9}*g1_7+x_{i+5}*g1_3+
                         ; x_{i+8}*g1_6+x_{i+4}*g1_2|
                         ; x_{i+7}*g1_5+x_{i+3}*g1_1+
                         ; x_{i+6}*g1_4+x_{i+2}*g1_0
haddps xmm3,xmm2         ; xmm3=0.0|x_{i+9}*g1_7+x_{i+8}*g1_6+
                         ; x_{i+7}*g1_5+x_{i+6}*g1_4+
                         ; x_{i+5}*g1_3+x_{i+4}*g1_2+
                         ; x_{i+3}*g1_1+x_{i+2}*g1_0|0.0|0.0
shufps xmm3,xmm3,10000000b ; xmm3=t2|0.0|0.0|0.0
addps xmm3,buf6          ; xmm3=t2|t1|0.0|0.0
addps xmm3,buf5          ; xmm3=y_{i+3}|y_{i+2}|y_{i+1}|y_{i+0}
movapsy[ecx],xmm3        ; y[ecx]=y_{i+3}|y_{i+2}|y_{i+1}|y_{i+0}
```

– Computation of coefficient t2 of DWT (at that moment it is yi+3), assembling yi+3, yi+2, yi+1, yi+0, in xmm register and storing its content into the memory

```
add ecx,16               ;(i=i+4) i.e. ecx=ecx+16
movaps xmm0,buf1         ; xmm0=x_{i+7}|x_{i+6}|x_{i+5}|x_{i+4}
movaps xmm1,buf0         ; xmm1=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
movaps buf1,xmm1         ; buf1=x_{i+11}|x_{i+10}|x_{i+9}|x_{i+8}
cmpecx,NN                ; Test the end of loop
                         ; condition(ecx = NN),
                         ;where NN=(N/4)*16
jneiloop                 ; Jump to the label iloop
                         ; mentioned in step c)
                         ; if  ecx ≠ NN
```

– updating xmm0 i xmm1 before the next iteration of the loop and exit from the loop.

Figure 3. Steps of computation of the coefficients y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} for i = 0,4,8,...,N-4 in Version A of the implementation of DWT using assembler with SSE extensions

After the initialization steps a) - b) the algorithm enters the loop shown in steps c) - h). The step c) shows how the output coefficient y_{i+0} is being computed and put into the least significant part of the *xmm* register. Two SSE multiplications and three SSE additions are performed in this phase that is equivalent to eight floating point multiplications and seven additions. The results are saved to appropriate parts of the *xmm* register.

The next phase of the loop, namely step d) is devoted to computation of output coefficient y_{i+1} and saving it in the subsequent, more significant part of the *xmm* register. After this step, the register contains coefficients y_{i+1}, y_{i+0}in its lower part and floating zeros in the upper part.

This step is almost identical to the preceding one, with the exception of replacing impulse response $h1$with $g1$.

In step e) registers *xmm*0 and *xmm*1 are loaded with input samples shifted by two positions, in relation to their previous content. Namely,*xmm0* contains samples x_5, x_4 ,x_3 ,x_2, and *xmm1*samples x_9, x_8, x_7, x_6. These data will be used to compute y_{i+3}, y_{i+2}.

Step f) show the details of computation of coefficient y_{i+2} and points out that its value is stored in the *xmm* register, next to already computed y_{i+1}, y_{i+0}. Again, this phase is very similar to step c), the only difference is the position in *xmm* register where the value of y_{i+2} is being saved.

Similarly, the step g) concerns computation of the forth coefficient y_{i+3}. It is stored in the most significant part of the register *xmm*. The final quadruplet y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} is saved from the *xmm* register into the memory.

The last phase of loop shown as step h) increments loop counter in *ecx* register by 16 (i.e. the size of *xmm* register in bytes) This value will be used to address input samples *x* and output coefficients *y*.

In the following lines of code, registers *xmm*0 and *xmm*1 are being prepared for computation of output coefficients y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} in the next iteration of the loop (with *i* incremented by 4).

The loop concludes with a test condition $ecx \neq NN$, where $NN=(N/4)*16$.

If this condition is met the code jumps to the label `iloop` discussed in step c), i.e. the beginning of the loop.

The code for case K=8 can be also used for K=6, provided the two most significant coefficients of reversed order impulse responses are set to zero, i.e. $h1_7=h1_6=g1_7=g1_6=0$. However, coefficients $h1_5 \ldots h1_0$ and $g1_5 \ldots g1_0$ need to be initialized with corrected values, appropriate for K=6.

3.2 Version B. Implementation of DWT using assembler with SSE extensions

The implementation for filter length K=12 is similar to Version A. Again it has been assumed that the number of input samples N is divisible by 4, which makes it possible to process, in one iteration of the loop, four output coefficient in the register *xmm*.

The implementation can be logically divided into 8 steps. At first, the registers *xmm0* are loaded with values x_3, x_2, x_1, x_0, *xmm1* with x_7, x_6, x_5, x_4 and *xmm2* with x_{11}, x_{10}, x_9, x_8. The reversed coefficients of impulse responses $h1_7$, $h1_6$, $h1_5$, $h1_4$and $h1_3$, $h1_2$, $h1_1$, $h1_0$ are sent to *xmm*5 and *xmm*4, while $g1_7$, $g1_6$, $g1_5$, $g1_4$and $g1_3$, $g1_2$, $g1_1$, $g1_0$ are sent to *xmm*7 and *xmm*6. However, because of the limited number of *xmm* registers the most significant parts of $h1_{11}$, $h1_{10}$, $h1_9$, $h1_8$ and $g1_{11}$, $g1_{10}$, $g1_9$, $g1_8$ will be fetched from memory.

Following the above initialization code there are six steps in the loop, as they were in version A.The first step shows how the output coefficient y_{i+0} is being computed and put into the least significant part of the *xmm* register. Three SSE multiplications and four SSE additions are performed in this phase which is equivalent to twelve floating point multiplications and eleven additions. The results are saved to appropriate parts of the *xmm* register.

The next phase of the loop is devoted to computation of output coefficient y_{i+1} and saving it into the subsequent, more significant part of the *xmm* register. After this step, the register contains coefficients y_{i+1}, y_{i+0}in its lower part and floating zeros in the upper part. Thatphase of the algorithm is almost identical to the preceding one, with the exception of replacing impulse response $h1$with $g1$.

In the next step registers *xmm*0, *xmm*1 and *xmm2* are loaded with input samples shifted by two positions, in relation to their previous content. Namely,*xmm0* contains samples x_5, x_4 ,x_3 ,x_2,, *xmm1*samples x_9, x_8, x_7, x_6, and*xmm*2 samples x_{13}, x_{12}, x_{11}, x_{10}. These data will be used to compute y_{i+3}, y_{i+2}.

In the subsequent step, coefficient y_{i+2} is being computed and stored in the *xmm* register, next to the already computed y_{i+1}, y_{i+0}. Again, this phase is very similar to the computation of y_{i+0}, the only difference is the position in *xmm* register where the value of y_{i+2} is being saved.

Similarly, the following step, concerns computation of the last coefficient y_{i+3}. It is stored in the most significant part of the register *xmm*. The final quadruplet y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} is transferred from the *xmm* register into the memory.

The last phase of loop increments loop counter in *ecx* register by 16 (i.e. the size of *xmm* register in bytes). This value will be used to address input samples x and output coefficients y.

In the following lines of code, registers *xmm* are being prepared for computation of output coefficients y_{i+3}, y_{i+2}, y_{i+1}, y_{i+0} in the next iteration of the

loop (with i incremented by 4). The loop concludes with a test condition $ecx \neq NN$, where $NN=(N/4)*16$.If this condition is met the code jumps to the label `iloop`, i.e. computation of y_{i+0}.

The code that computes DWT for K=12 can be also used for K=10, provided the most significant coefficients of reversed impulse responses are set to zero, i.e. $h1_{11}$,=$h1_{10}$,=$g1_{11}$,=$g1_{10}$=0. However, coefficients $h1_9$… $h1_0$ and $g1_9$… $g1_0$ need to be initialized with corrected values, appropriate for K=10.

3.3 Version C. Implementation of DWT using assembler with SSE extensions

This version of DWT implementation assumes that K=6, but the number of input samples $N$is even. In that case, on every odd iteration of the loop, the computations are performed in the same way as in version A (see Figure 3. Step c), and the pair of output coefficients y_{i+1}, y_{i+0} is saved on the least significant positions od 128-bit long buffer *buf*. On every even iteration, a following pair of output coefficients is being computed and saved in memory, together with a preceding pair, as a quadruplet of properly ordered coefficients.

If N is not divisible by 4, the last save operation concerns only the last pair of y_{i+1}, y_{i+0}.The code for computation of output coefficients is identical in versions A and C of the algorithm, so is the time of computation for K=6.

4 Test environment

All DWT implementations presented in the paper were written as C++ inline assembly (with SSE extensions) and compiled with Microsoft Visual C++ 2010 Express Version 10.0.40219.1 SP1Rel. The compiled code was executed on MS Windows 7 Home Premium PC with Intel® Core™ i5 CPU 650 3.20GHz and 4GB of RAM on board. Further, to neglect impact of concurrent operations of the processor on the computation time, all tests were run pN times, and the minimum time of execution, obtained with 64-bit clock cycle counter (measuring the number of clock cycles of the very code responsible for the computation of DWT), has been taken as a actual result of the measurement.

5 Experimental results

In order to compare effectiveness of the proposed implementation of DWT using assembler with SSE extensions, it was compared against the reference program written in pure C++, for the selected lengths of filter K, and a few

lengths of a input sequence N being divisible by 4. The resulting measurements expressed in cycles are gathered in Table 1.

Table 1. Results of measurement for pN = 100 000 000

K	Implementation	*N*=64	*N*=256	*N*=1024	*N*=4096
6	C++	3 465	13 983	55 902	223 611
	Assembler with SSE (version A)	459	1 845	7 395	29 490
8	C++	4 479	18 246	72 927	291 693
	Assembler with SSE (version A)	459	1 845	7 389	29 862
10	C++	5 577	22 374	89 193	357 006
	Assembler with SSE (version B)	594	2 403	9 621	38 880
12	C++	6 579	26 628	106 965	425 748
	Assembler with SSE (version B)	597	2 424	9 657	38 616

As can be seen from the table above, for filters of length K=8 and all tested values of N, implementation of DWT in assembler, with SSE extensions is performed almost 10 times faster than pure C++ version. For K=12 the optimized code is almost 11 times faster. This speedup may be attributed to manually optimized assembler implementation with parallel processing of data using SSE extensions. As it was mentioned in the information about SSE in IA-32 architecture, this may shorten the time of computation up to four times.

Further reduction of execution time, results from unfolding the inner loop which is the most computationally intensive. The outer loop contains mainly instructions for reading samples x and writing output coefficients y.

Because the implementation for K=6 and 8 uses the same version (A) of the algorithm, execution time is almost identical in both cases. The same holds true for version B and K=10 and 12.

Regardless of the version of implementation and the value of K, the amount of time needed to compute DWT is proportional to the length of the input sequence (and number of iterations). It is a direct conclusion from the formula (2).

Eventually the version C, for even N, has been examined. The results are shown in Table 2.

Table 2. Results of measurement for pN = 100 000 000

Implementation	***N=64***	***N=256***	***N=1024***	***N=4096***
Assembler with SSE (version C)	465	1839	7389	29436

The execution times for version C are virtually identical to version A. As a matter of fact, it is an expected result as both implementations share the same code to compute output pairs of coefficients. Moreover, although the construction and analysis of version C is more complex than version A, the speed of version C remains the same. Therefore, it is sufficient to use version A for K=6 and 8, and version B for K=10 and 12 and exclude special implementations for N, that are even but not divisible by four.

6 Conclusions

The paper discusses a number of implementations of Discrete Wavelet Transform written as a formula (2). The experimental results show that manually optimized C++, with unfolded inner loop and inline assembly code with SSE extensions, is about 10 times more robust than reference program written in pure C++. What is more, the obtained speedup looks favorably, comparing to the results shown in [14] where the SSE enabled code was performed 6x faster than naïve, C++ implementation of the convolution algorithm.

Although it is possible to achieve even further speedup with the application of the thread level parallelism of contemporary multi-core processors, the necessary algorithms are considerably more complicated. Hence, the proposed solution that use only Data Level Parallelism with SSE extensions is an attractive alternative, available even on a simple one core processors.

Due to the lower complexity of versions A and B, they are recommended as effective templates for computation of DWT with application of SSE.

References

1. Zieliński T. P.,2009,*Cyfrowe przetwarzanie sygnałów. Od teorii do zastosowań*, WKŁ, Warszawa
2. Fleet P. J.: 2008, *Discrete wavelet transformations: An elementary Approach with applications*, John Wiley&Sons, New Jersey.
3. Strang G., Nguyen T., 1999,*Wavelets and filter banks*, Wellesley-Cambridge Press.
4. Lipiński P.,2011,*Watermarking software in practical applications*, Bulletin of Polish Academy of Sciences: Technical Sciences, Vol. 59, nr 1, pp. 21-25.

5. Cooklev T., 2006,*An efficient architecture for orthogonal wavelet transforms*, IEEE Signal processing letters, Vol. 13, nr 2, pp. 77-79.
6. Olkkonen J. T., Olkkonen H., 2007,*Discrete lattice wavelet transform*, IEEE Transactions on circuits and systems – II: Express briefs, Vol. 54, nr 1, pp. 71-75.
7. Daubechies I., Sweldens W., 1998,*Factoring Wavelet Transform into Lifting Steps*, The Journal of Fourier Analysis and Applications, Vol. 4, nr 3, pp. 245-267.
8. Denk T. C., Parhi K. K., 1997,*VLSI architectures for lattice structure based orthonormal discrete wavelet transforms*, IEEE Transactions on circuits and systems – II: Analog and digital signal processing, Vol. 44, nr 2, pp. 129-132.
9. Bernabe G., Garcia J. M., Gonzalez J., 2003,*Reducing 3D Wavelet Transform Execution Time through the Streaming SIMD Extensions*, IEEE Computer Society, Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 209-223
10. ShahbahramiA., Juurlink B., Vassiliadis S., 2008,*Implementing the 2-D Wavelet Transform on SIMD-Enhanced General-Purpose processors*, IEEE Transactions on multimedia, Vol. 10, nr 1, pp. 43-51.
11. Yatsymirskyy M., 2011,*Nowy model macierzowy dwukanałowego banku biortogonalnych filtrów*, Metody Informatyki Stosowanej, nr 1/2011 (26), Polska Akademia Nauk Oddział w Gdańsku, Komisja Informatyki, pp. 205-212.
12. Yatsymirskyy M.,Stokfiszewski K., 2012, Effectiveness of Lattice Factorization of Two-Channel Orthogonal Filter Banks, New Trends in Audio and Video/ Signal Processing Algorithms, Architectures, Arrangements and Applications, 27-29 September, Łódź, pp. 275-279.
13. *Intel® 64 and IA-32 Architectures. Software Developer's Manual, Volume 1: Basic Architecture*.
14. Gomersall H., 2012, *Speedy fast 1D convolution with SSE,* http://hgomersall.wordpress.com/2012/11/02/speedy-fast-1d-convolution-with-sse/

EQUIVALENCE IN JAVA AND CLOJURE, DESIGN AND IMPLEMENTATION CONSIDERATIONS

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Immutability and the functional programming style demand an extensible and generic approach in the domain of semantic and structural equivalence testing. The lack of a library or a framework offering such functionality for Clojure programming language led to some design and implementation efforts that this article undertakes to describe. Incidentally it tries to gather and present a collection of most severe mistakes that may be made by a programmer that attempts to test objects of various kinds for their equivalence, both in Clojure and the underlying Java run-time with it's standard library, showing simple yet usable ways to avoid them.

Key words: Equivalence testing, semantics, identity, functional programming, Clojure

1 Introduction

Growing multitasking programming needs and the popularity of functional programming style brought the notions of immutability and state to the forefront of elements a software engineer must think of when designing and implementing modern software systems. Immutable objects that are commonly associated with mathematical models of the real world make the structural equality a default choice, in the opposition to the explicitly expressed equality, based on an explicit identifier, physical memory location etc., that must be used under an assumption of the always present change. Additionally and in a resulting way, duck typing (see e. g. [1]) is a programming means of abstraction of a growing importance at least in some kinds of systems. This goes in an analogous ways in an opposition to the tag-based typing. Unfortunately, the state of the art in programming languages, even the most advanced ones is not an optimal one when talking about the objects' identity and structural equivalence. The paper gives an overview of these problems and tries to

present a generalized solution based on some solid abstractions. Then the important implementation details of an identity framework for Clojure is presented.

2 Reference Types Equivalence Problems

If we assume a concentration on the structural equivalence issues, then the lack of a generalized and extensible solution to the problem of both in Java and Clojure is apparent. These two languages are mentioned here for the following four reasons:

1. Java is a typical, strongly and statically typed programming language [2], very representative for a class of languages used widely nowadays and known as the object-oriented ones. The default identity is the memory location-based one.
2. There are multiple reasons why implementing a non-default identity criteria by overriding the ***java.lang.Object equals*** and ***hashCode*** methods is hard and error-prone [3, 4]. Taking a detailed look at these mechanisms and problems laying there is beyond the scope of this article, but will be presented elsewhere in the future works.
3. Clojure is a modern functional language [5, 6], supporting immutability and using ***Software Transactional Memory*** where the explicit state must be used to achieve a desired functionality. Clojure is strongly typed but in an opposition to Java it lacks static type-checking and uses duck-typing where possible.
4. The two languages both run on top of the JVM, Clojure shares Java libraries and is capable to run an arbitrary Java code, on the other hand embedding Clojure run-time in a Java application is an easy task. One can say these languages are related worlds despite the fundamental stylish and typing differences between them.

Clojure standard library as well as some run-time elements support structural equivalence with respect to collections, in particular. Sequences (vectors, lists), sets and associative collections (maps, records) all exhibit support for deep, structural comparison. Unfortunately, this support is not extensible. Yes, a presence of some interfaces suggests that the mechanisms are capable of being extended, but:

- There are some implementation details that effectively block extending the run-time abstractions with custom classes, written either in Clojure (records, types) or Java (classes). An example of this is introducing a new

composite numeric type, like a Complex number[1]. The new non-atomic numeric type does not fit into Clojure equivalence mechanisms for numbers and there is no way to solve this problem without making significant changes to the core of the language.
- The situation gets disclosed when trying to integrate an existing any computational library into the Clojure based system.
- Even if there were no barriers described above, using the default interface-based abstractions is impossible on already written types (Java classes in particular). Using AOP as described by Kiczales [7] is not an elegant nor easily accessible solution here.

All these problems are easily solvable with use of Clojure ***protocols*** [6], but currently there are no libraries of this kind. This is a very important premise that influenced creating a universal solution described in this article.

3 Equivalence of Numeric Values – Quirks and Corner Cases

Problems described in the previous section expand onto the primitive types, their values as well as their boxed counterparts. To focus our considerations, the general contract for equality and hashing must be provided to the reader. Java Language Specification [2] as well as some other resources [8] say that ***equals*** method implements an equivalence relation. It is:
- Reflexive: For any non-null reference value *x*, *x.equals(x)* must return *true*.
- Symmetric: For any non-null reference values *x* and *y*, *x.equals(y)* must return *true* if and only if *y.equals(x)* returns *true*.
- Transitive: For any non-null reference values *x*, *y*, *z*, if *x.equals(y)* returns *true* and *y.equals(z)* returns *true*, then *x.equals(z)* must return *true*.
- Consistent: For any non-null reference values *x* and *y*, multiple invocations of *x.equals(y)* consistently return *true* or consistently return *false*, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value *x*, *x.equals(null)* must return *false*.

For the ***hashCode*** method, the following set of constraints applies:
- Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

1 Complex numbers are not present in Clojure by default. ANSI Common Lisp ([9]) supports them.

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Neither the Java primitives[2] nor the derivatives of *java.lang.Number* possess the semantically correct implementations of equivalence mechanisms as a whole. Saying "semantically correct" we mean a correct behavior of the proper methods and operators with respect to *Liskov substitution principle* [10]. Moreover, using some values of these types lead to surprising results, especially the floating-point values representation in Java[3] causes real headaches when attempting to implement solid numeric codes.

The rest of this section is an attempt to present a catalog of semantically incorrect behaviors of numeric values. All examples are given in Clojure, and so we focus on boxed types rather than the primitive ones. We also use *clojure.core/=* and *clojure.core/hash-code* operators instead of calling *equals* and *hashCode* explicitly[4].

There are the most important examples of malfunctioning equivalence in Clojure and Java:

- Erroneous floats equivalence, both in primitive type values and in the boxed ones. Example:

```
> (= (float 1.234) 1.234)
false

> (hash 1.234)
-146307282

> (hash (float 1.234))
1067316150
```

This turns out to be eventually a conversion problem between floats and doubles, because when applying *clojure.core/=* operator the Clojure run-time

2 In the case of primitives we mean the == operator in Java, not the *equals/hashCode* complementary set of methods that apply only for reference types.

3 And all languages with standard IEEE 754 ([11]) floating-point representation.

4 The reader familiar with Clojure should be aware that these operators semantically wrap *equals* and *hashCode*.

All the following source code examples assume the following Clojure ***name-space*** context:

```
(ns kongra.behavior
  (:refer-clojure :exclude [rand])

  (:use     [kongra.core])
  (:require [clojure.set                 :as CSET]
            [clojure.math.combinatorics :as CMCOMB]

            [kongra.behavior            :as B]
            [kongra.identity            :as ID]
            [kongra.fressian            :as FRESS]))
```

The operator ***cat*** concatenates given arguments collections. It's internal workings are based on using the standard ***clojure.core/apply*** procedure:

```
(defn cat
  [& colls]
  (->> colls
       (apply concat)
       (with-correctness[1] (apply correctness[1] colls))))
```

In a Clojure REPL one could execute the following and observe the results[2] of using ***cat***:

```
> (cat [1 2 3 4] [[:a :b] [:c :d]])
(1 2 3 4 [:a :b] [:c :d])
```

When generating arguments by matching together single values from the passed sequences of values one can ***zip*** the sequences together:

```
(defn zip
  [& colls]
  (->> colls
       (apply map vector)
       (with-correctness (apply correctness colls))))
```

and the following occurs:

```
> (zip [1 2 3 4] [[:a :b] [:c :d]])
([1 [:a :b]]
 [2 [:c :d]])
```

1 For arguments' and arguments collections' correctness, please go to section 5 of this paper.
2 All procedures described in this section produce lazily evaluated results.

As you can see, the related (with respect to the same position) components of passed streams are combined to form new arguments and later placed in a resulting stream. The ***zip*** operator has it's variadic version called ***vzip***:

```
(defn vzip
  [& colls]
  (->> colls
       (apply map #(concat (butlast %&) (last %&)))
       (with-correctness (apply correctness colls))))
```

that produces a slightly different result when applied to the same set of data:

```
> (vzip [1 2 3 4] [[:a :b] [:c :d]])
((1 :a :b)
 (2 :c :d))
```

The ***vzip*** operator may be especially useful when creating streams of arguments to test procedures with variadic arities.

To combine every element of all arguments collections with one another one must use the Cartesian product ***prod***:

```
(defn prod
  [& colls]
  (->> colls
       (apply CMCOMB/cartesian-product)
       (with-correctness (apply correctness colls))))
```

or it's "variadic" counterpart – ***vprod***:

```
(defn vprod
  [& colls]
  (->> colls
       (apply B/prod)
       (map #(concat (butlast %) (last %)))
       (with-correctness (apply correctness colls))))
```

The two operators give results as follows:

```
> (prod [1 2 3 4] [[:a :b] [:c :d]])
((1 [:a :b])
 (1 [:c :d])
 (2 [:a :b])
 (2 [:c :d])
 (3 [:a :b])
 (3 [:c :d])
 (4 [:a :b])
 (4 [:c :d]))
```

```
> (vprod [1 2 3 4] [[:a :b] [:c :d]])
((1 :a :b)
 (1 :c :d)
 (2 :a :b)
 (2 :c :d)
 (3 :a :b)
 (3 :c :d)
 (4 :a :b)
 (4 :c :d))
```

These are the key arguments collections (streams) manipulating arguments. Among the arguments generators the most important ones are those which generate a stream of variable arity arguments sets:

```
(defn vargs
  [coll]
  (->> coll count inc range
       (map #(take % coll))
       (with-correctness (correctness coll))))

> (vargs [1 2 3 4])
(()
 (1)
 (1 2)
 (1 2 3)
 (1 2 3 4))
```

The ***vargs*** operator takes an example arguments vector and generates an arguments collection (stream, coll of arguments) with variable arguments vector size, as presented above. Similarly, ***vmaps***:

```
(defn vmaps
  [keyvals]
  (assert (even? (count keyvals)))
  (->> keyvals
       (partition 2) ;; all possible entries
       powerset      ;; all possible subsets
       (map #(apply hash-map (apply concat %)))
       (with-correctness (correctness keyvals))))
```

produces a stream of maps (associative collections) with all possible “arities” of map entries:

```
> (vmaps [:a 1 :b 2])
({}
 {:a 1}
 {:b 2}
 {:a 1, :b 2})
```

To produce a testing collection for the procedures with formal parameters of type 4 – the variable arity arglists with maps playing the role of keyword arguments carriage, a simple ***mapargs*** may be used:

```
(defn mapargs
  [m]
  (->> m
       (apply concat)
       (with-correctness (correctness m))))

> (mapargs {:a 1 :b 2})
(:a 1 :b 2)
```

together with a ***vmapargs*** operator:

```
(defn vmapargs
  [keyvals]
  (assert (even? (count keyvals)))
  (->> keyvals
       (partition 2) ;; all possible entries
       powerset      ;; all possible subsets
       (map #(apply concat %))
       (with-correctness (correctness keyvals))))

> (vmapargs [:a 1 :b 2])
(()
 (:a 1)
 (:b 2)
 (:a 1 :b 2))
```

that works almost like ***vmaps***, but converts any generated map into a flattened sequence of key-value pairs (map entries).

Finally the two following operators: ***powargs*** and ***permargs*** use power-sets and permutations to generate proper arguments collections:

```
(defn powargs
  [coll]
  (->> coll
       powerset
       (with-correctness (correctness coll))))

> (powargs [1 2 3])
(() (1) (2) (1 2) (3) (1 3) (2 3) (1 2 3))

(defn permargs
  [coll]
  (->> coll
```

```
       CMCOMB/permutations
       (with-correctness (correctness coll))))

> (permargs [1 2 3])
([1 2 3] [1 3 2] [2 1 3] [2 3 1] [3 1 2] [3 2 1])
```

5 The Correctness Abstraction

The correctness is an enumerated type with an integral ***code*** field:

```
(deftype ^:private Correctness
  [name code]

  java.lang.Object
  (toString [this] name))
```

Besides the correctness levels mentioned earlier there is also a ***CORRECTNESS-UNDEFINED.*** The enumeration values go as follows:

```
(def CORRECTNESS-UNDEFINED
     (Correctness. "CORRECTNESS-UNDEFINED" (byte 0)))
(def NON-BORDER
     (Correctness. "NON-BORDER"            (byte 1)))
(def BORDER
     (Correctness. "BORDER"                (byte 2)))
(def PARTIALLY-CORRECT
     (Correctness. "PARTIALLY-CORRECT"     (byte 3)))
(def INCORRECT
     (Correctness. "INCORRECT"             (byte 4)))
```

and the correctness of a collection of objects is the maximum correctness of the elements of the collection:

```
(defn- max-correctness
  ([c] c)
  ([c d]
     (if (> (.longValue ^Number (.code ^Correctness c))
            (.longValue ^Number (.code ^Correctness d)))
       c d))

  ([c d & more]
     (reduce max-correctness
             (max-correctness c d)
             more)))
```

Correctness of an object may be specified explicitly by setting a proper association in it's ***meta-data*** or implicitly, by using an indicator function implemented as a Clojure ***protocol*** method:

```
(defprotocol WithImplicitCorrectness
  (^:private implicit-correctness [this]))
(defn correctness
  ([obj]
     (or (::correctness (meta obj))
         (implicit-correctness obj)))

  ([obj & rest]
     (apply max-correctness
            (correctness obj)
            (map correctness rest))))
```

Finally the correctness may be applied to an object explicitly with:

```
(defn with-correctness
  [c obj]
  (vary-meta obj assoc ::correctness c))
```

The latter approach is used in all arguments manipulation routines.

6 Implicit Correctness for Some Known Types and Values

The framework described here introduces implicit correctness as a predefined set of procedures. In a conventional, imperative language with a static type system, like Ada or Java, achieving such functionality involves a significant change(s) in a standard library, as one needs to define a set of polymorphic[3] procedures dispatched on the types belonging to a standard library of the host language. Thankfully in Clojure we have protocols that are perfect means to implement the extension points for the desired functionality.

The implicit correctness of a sequential collection is the aggregate correctness of it's elements or a ***BORDER*** correctness if the collection is empty:

```
(defn- implicit-seq-correctness
  [coll]
  (if-let [s (seq coll)]
    (apply correctness s)
    ;; an empty sequence is intentionally qualified
    ;; as a BORDER one
    BORDER))
```

3 With an inclusive polymorphism as described by L. Cardelli [16]

For integrals we define 0, -1, 1, the maximum and minimum values as those having the ***BORDER*** correctness level and assign ***NON-BORDER*** to any others:

```
(defn- implicit-integral-correctness
  [^Number x ^Number min ^Number max]
  (let [x (.longValue x)]
    (if (or (= x (.longValue min))
            (= x (.longValue max))
            (= x  0)
            (= x  1)
            (= x -1))
      BORDER
      NON-BORDER)))
```

A similar approach applies to primitive floating-point values (***java.lang.Float*** and ***java.lang.Double*** both in Java and in Clojure). Additionally the ***infinite*** and ***NaN*** (***Not-a-Number***) values must be considered here.

```
(defn- implicit-double-correctness
  [^Double x]
  (let [d (.doubleValue x)]
    (if (or (Double/isNaN d)
            (Double/isInfinite d)
            (= d Double/MAX_VALUE)
            (= d Double/MIN_NORMAL)
            (= d Double/MIN_VALUE)
            (= d  0.0)
            (= d  1.0)
            (= d -1.0))
      BORDER
      NON-BORDER)))
```

And then there is the protocol named ***WithImplicitCorrectness***. Apart from the fact that it allows do implement all predefined out-of-the-box correctness values in the framework itself, it also gives the programmer a handle to define his own correctness assignments for types that will exist in the future:

```
(defprotocol WithImplicitCorrectness
  (implicit-correctness [this]))
```

The protocol when applied to collections uses the implicit-seq-correctness procedure, as defined earlier in this section. One exception is the pair (a type named ***kongra.core.Pair***), but it does not differ much in the semantics when compared to the mentioned implementation procedure:

```
(extend-protocol WithImplicitCorrectness
  ;; SEQUENTIAL COLLECTIONS
  clojure.lang.Sequential
  (implicit-correctness [this]
     (implicit-seq-correctness this))
  java.util.List
  (implicit-correctness [this]
    (implicit-seq-correctness this))
  kongra.core.Pair
  (implicit-correctness [this]
    (correctness (.first this) (.second this)))
  ;; SETS
  java.util.Set
  (implicit-correctness [this]
    (implicit-seq-correctness this))
```

Associative containers (maps) have their correctness defined as an aggregate correctness of all keys and values:

```
  ;; MAPS (INCLUDING RECORDS)
  java.util.Map
  (implicit-correctness [this]
    (if-let [entries (seq this)]
      (implicit-seq-correctness (apply concat entries))
      ;; an empty map has a BORDER correctness
      BORDER))
```

Strings have a ***BORDER*** correctness when they are blank (contain only white-space characters), and ***NON-BORDER*** otherwise:

```
  ;; STRING-LIKE
  java.lang.String
  (implicit-correctness [this]
    ;; a blank string is a BORDER one
    (if (blank? this) BORDER NON-BORDER))
```

Clojure ***symbols*** and ***keywords*** "adopt" a similar String-like rule – their names are checked for being blank:

```
  clojure.lang.Named ;; symbols, keywords
  (implicit-correctness [this]
    (if (blank? (.getName this)))
      BORDER
      NON-BORDER))
```

Here is how the implicit integral correctness is being defined in the protocol:

```
;; INTEGERS
java.lang.Byte
(implicit-correctness [this]
  (implicit-integral-correctness this
                                 Byte/MIN_VALUE
                                 Byte/MAX_VALUE))

java.lang.Short
(implicit-correctness [this]
  (implicit-integral-correctness this
                                 Short/MIN_VALUE
                                 Short/MAX_VALUE))

java.lang.Character
(implicit-correctness [this]
  (implicit-integral-correctness
    (int this)
    (int Character/MIN_VALUE)
    (int Character/MAX_VALUE)))

java.lang.Integer
(implicit-correctness [this]
  (implicit-integral-correctness
     this Integer/MIN_VALUE Integer/MAX_VALUE))

java.lang.Long
(implicit-correctness [this]
  (implicit-integral-correctness
     this Long/MIN_VALUE Long/MAX_VALUE))
```

The big-integer types in Java and in Clojure also "define" 0, -1 and 1 as their ***BORDER*** values. As they do not impose any limits on how the integral values are allowed to be (the memory and CPU time are the only constraints), there are no max- or min-values being taken into account:

```
;; BIG INTEGER, BIG INT
java.math.BigInteger
(implicit-correctness [this]
  (if (or (.equals this java.math.BigInteger/ZERO)
          (.equals this java.math.BigInteger/ONE)
          (.equals this BIG-INTEGER-MINUS-ONE))
    BORDER
    NON-BORDER))
clojure.lang.BigInt
(implicit-correctness [this]
  (if (or (.equals this  0N)
          (.equals this  1N)
```

```
            (.equals this -1N))
    BORDER NON-BORDER))
```

The same applies to the arbitrary precision floating-point type ***java.math.BigDecimal***:

```
;; BIG DECIMAL
java.math.BigDecimal
(implicit-correctness [this]
  (if (or (BD/= this  0M)
          (BD/= this  1M)
          (BD/= this -1M))
    BORDER
    NON-BORDER))
```

Due to their nature Clojure rational numbers represented by instances of ***clojure.lang.Ratio*** class [14], [15] are NON-BORDER values:

```
;; RATIO
clojure.lang.Ratio
(implicit-correctness [this] NON-BORDER)
```

Finally the protocol defines the correctness for floats:

```
;; FLOATS
java.lang.Float
(implicit-correctness [this]
  (implicit-double-correctness (ID/fldouble this)))

java.lang.Double
(implicit-correctness [this]
  (implicit-double-correctness this))
```

and any other types, including ***null*** values (***nil*** in Clojure) have their correctness undefined:

```
;; OTHERS
java.lang.Object
(implicit-correctness [this] CORRECTNESS-UNDEFINED)

nil
(implicit-correctness [_] CORRECTNESS-UNDEFINED))
```

7 Conclusions and Future Works

The paper presented only a fraction of the whole work needed to fully implement the initial idea. There are the following points that still wait for their detailed design and implementation:

1. Routines to explicitly specify values with various levels of correctness for types
2. The results model
3. Behaviors storage
4. Procedures evaluation with the automatically generated collections of arguments
5. Behaviors comparison

The main technical sections of the article concatenated on presenting the correctness-related mechanisms and the arguments manipulating operators. When talking about the latter, there is an urge to design and implement an embedded[4] DSL, a kind of a "regular expressions" language to make the usage of the arguments manipulation operators more effective in use than simply calling them explicitly. A sketch of an expression of this kind is like:

```
^:prod [x & ^:vmapargs {:y 1 :z 2}]
```

where the operators are used within the argist s-expression as a meta-data (defined with the Clojure keywords). Implementing this functionality is the first sub-task to be done during the future development activities on the framework presented here and it will be described in a future paper.

References

1. Koskela L., 2008, *Test Driven, Practical TDD and Acceptance TDD for Java Developers*, ISBN 1-932394-85-0, Manning Publications Co
2. E. W. Dijkstra, 1972, *The Humble Programmer*, ACM Turing Lecture
3. Thomas M., 2003, *The Modest Software Engineer*, Proc ISADS 2003, pp 169-174, IEEE Press
4. L. Williams, E. M. Maximilien, M. Vouk, 2003, *Test-Driven Development as a Defect-Reduction Practice*, ISSRE '03 Proceedings of the 14th International Symposium on Software Reliability Engineering, pp. 34
5. 2007, *Ada Reference Manual*, ISO/IEC 8652:2007(E) Ed. 3
6. Jones S. P., 2003, *Haskell 98 language and libraries: the Revised Report*, ISBN 0521826144, Cambridge University Press

4 in Clojure as the host language

7. 2008, *SPARK 95 - The SPADE Ada 95 Kernel*, Praxis High Integrity Systems Ltd
8. Hevery M., 2008, *Guide: Writing Testable Code*, http://misko.hevery.com/code-reviewers-guide/
9. Miller A., 2008, *Clojure and testing*, http://tech.puredanger.com/2013/08/31/clojure-and-testing/
10. Sierra S., 2014, *API for clojure.test*, http://richhickey.github.io/clojure/clojure.test-api.html
11. Martin M., 2014, *Speclj - A TDD/BDD framework for Clojure*, http://speclj.com/
12. 2014, *Midje Github Repository*, https://github.com/marick/Midje
13. 2014, *HUnit -- Haskell Unit Testing*, http://hunit.sourceforge.net/
14. Halloway S., 2009: *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf
15. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7
16. Cardelli L., Wegner P., 1985, *On Understanding Types, Data Abstraction and Polymorphism*, Computing Surveys, Vol. 17 n 4, pp. 471-522, 1994

Information for Authors

Authors are invited to submit original papers, within the scope of the JACSM, with the understanding that their contents are unpublished and are not being actively under consideration for publication elsewhere.

For any previously published and copyrighted material, a special permission from the copyright owner is required. This concerns, for instance, figures or tables for which copyright exists. In such a case, it is necessary to mention by the author(s), in the paper, that this material is reprinted with the permission.

Manuscripts, in English, with an abstract and the key words, are to be submitted to the Editorial Office in electronic version (both: source and pdf formats) via e-mail. A decision to accept/revise/reject the manuscript will be sent to the Author along with the recommendations made by at least two referees. Suitability for publications will be assessed on the basis of the relevance of the paper contents, its originality, technical quality, accuracy and language correctness.

Upon acceptance, Authors will transfer copyright of the paper to the publisher, i.e. the Academy of Management, in Lodz, Poland, by sending the paper to the JACSM Editorial Office.

The papers accepted for publication in the JACSM must be typeset by their Authors, according to the requirements concerning final version of the manuscripts. However, minor corrections may have been carried out by the publisher, if necessary.

The printing area is 122 mm × 193 mm. The text should be justified to occupy the full line width, so that the right margin is not ragged, with words hyphenated as appropriate. Please fill pages so that the length of the text is no less than 180 mm.

The first page of each paper should contain its title, the name of the author(s) with the affiliation(s) and e-mail address(es), and then the abstract and keywords, as show above. Capital letters must be applied in the title of a paper, and 14 pt. boldface font should be used, as in the example at the first page. Use: 11-point type for the name(s) of the author(s), 10-point type for the address(es) and the title of the abstract, 9-points type for the abstract and the key words.

For the main text, please use 11-point type and single-line spacing. We recommend using Times New Roman (TNR) fonts in its normal format. Italic type may be used to emphasize words in running text. Bold type and underlining should be avoided. With these sizes, the interline distance should be set so that some 43 lines occur on a full-text page.

The papers should show no printed page numbers; these are allocated by the Editorial Office.

Authors have the right to post their Academy of Management-copyrighted material from JACSM on their own servers without permission, provided that the server displays a prominent notice alerting readers to their obligations with respect to copyrighted material. Posted work has to be the final version as printed, include the copyright notice and all necessary citation details (vol, pp, no, …).

An example of an acceptable notice is:
“This material is presented to ensure timely dissemination of scholarly work, and its personal or educational use is permitted. Copyright and all rights therein are retained by the Copyright holder. Reprinting or re-using it in any form requires permission of the Copyright holder.

The submission format for the final version of the manuscript
and the MS Word template are available on our web site at:
http://acsm.spoleczna.pl

Journal of Applied Computer Science Methods

Contents