Maciej Rafał BURAK
West Pomeranian University of Technology, Applied Informatics

**COMPUTING DISCRETE POISSON PROBABILITIES FOR UNIFORMIZATION ALGORITHM**

**Summary.** This paper deals with the computation of discrete Poisson probabilities and their precise tail bounds for the use in solving of transient Markov models with the uniformization with steady-state detection algorithm. The algorithm calculates the weights corresponding to the probabilities, maintaining the beneficial properties for the use in uniformization algorithm of the popular Fox-Glynn method, while improving the estimation of truncation points with significant effect on the overall performance.

**Keywords:** computing discrete distributions, Poisson, uniformization, numerical methods, algorithms

**OBLICZANIE PRAWDOPODOBIEŃSTW DYSKRETNEGO ROZKŁADU POISSONA DLA ALGOTRYTMU UNIFORMIZACJI**

**Streszczenie.** W artykule przedstawiany jest algorytm wyliczający wartości prawdopodobieństw dyskretnego rozkładu Poissona dla algorytmu uniformizacji, wykorzystywanego dla modelowania stanów nieustalonych łańcuchów Markowa z czasem ciągłym. Algorytm jest rozwinięciem popularnej metody Foxa-Glynną i oblicza wagi odpowiadające ww. rozkładowi w sposób umożliwiający dokładniejsze ustalenie wartości granicznych dla zadanego błędu obliczeń, co przekłada się bezpośrednio na efektywność obliczeniową.

**Słowa kluczowe:** rozkład dyskretny Poissona, uniformizacja, metody numeryczne
1. Introduction

The paper deals with the computation of discrete Poisson distribution values for the use in the uniformization with steady-state detection algorithm and is a corrected version of an unpublished paper available at authors researchgate personal profile.

The algorithm computes, for a given error bound $\varepsilon$ representing the remaining value of the cdf Poisson distribution, the temporary truncation points $L_t$ and $R_t$ such that both, the probabilities $Q_t(L_t)$ that at least $\lfloor L_t \rfloor$ and $T_t(R_t)$ that at most $\lfloor R_t \rfloor$ events happen for given $\lambda$ as defined in [3], are always smaller than $\varepsilon / 2 \times 10^{-3}$, using simple approximation related to the central limit theorem. Afterwards, starting from the mode $m = \lfloor \lambda \rfloor$ we recursively compute the weights and, then, their sum – the total weight (used for scaling the weights in order to calculate the discrete probabilities). Further, the precise truncation points $L$ and $R$, such that both $Q_t(L)$ and $T_t(R)$ are smaller than $\varepsilon / 2$, are obtained by recursive summation of the weights starting from $L_t$ and $R_t$.

The paper is structured as follows. The next section reviews the uniformization with steady-state detection algorithm with emphasis on the role of the truncation bounds. In section 3 the proposed method of calculating the discrete Poisson probabilities is introduced, followed by the implementation details and computational examples. The paper ends with a summary of the numerical experiments results and conclusions.

2. Uniformization Method of Solving CTMC's

A Continuous Time Markov Chain (CTMC) is described by infinitesimal generator matrix $Q : n+1 \times n+1, Q = (q_{i,j})$ and the initial state probability vector $p(0)$, where the value $q_{i,j} (i \neq j)$ is the rate at which the state $i$ changes to state $j$ and $q_{i,i} = - \sum_{j \neq i} q_{i,j}$ represents the rate for the event of staying in the same state. The transient distribution at time $t$ $p(t)$ can be calculated using Kolmogorov’s forward equations:

$$p'(t) = p(t)Q$$

where the vector $p(t) = [p_0(t), \ldots, p_n(t)]$ gives probabilities of the system being in any of the states at time $t$.

Uniformization or Randomization, known since the publication of Jensen in 1953 and, therefore, referenced to as the Jensen method, is the method of choice for computing transient
behavior of CTMCs. Many authors concluded that it is computationally superior to
differential equation solvers e.g. [4],[8] or [1]. To use uniformization we first define the
matrix
\[ P = I + \frac{Q}{\alpha} \]  \hspace{1cm} (2)
which for the uniformization rate \( \alpha \geq \max(|q_{ij}|) \) is a stochastic matrix.

In order to preserve the sparsity of the matrix \( P \) the usual method of implementing
uniformization is to compute the transient distribution \( p(t) \) using:
\[ p(t) = \sum_{i=0}^{\infty} \Pi(i) e^{-\alpha t} \frac{(\alpha t)^i}{i!} \]  \hspace{1cm} (3)
where \( \alpha \) is uniformization rate, as described in (2), and \( \Pi(i) \) is the state probability vector of
the underlying DTMC after each step \( i \) computed iteratively by:
\[ \Pi(0) = p(0), \Pi(i) = \Pi(i-1)P \]  \hspace{1cm} (4)
as proposed e.g. in [8].

To compute \( p(t) \), within prespecified error tolerance, the computation can stop when the
remaining value of cdf of Poisson distribution is less than the error bound \( \varepsilon \) :
\[ 1 - \sum_{i=0}^{k} e^{-\alpha t} \frac{(\alpha t)^i}{i!} \leq \varepsilon \]  \hspace{1cm} (5)
with \( k \) being the right truncation point.

As \( \alpha t \) increases, the corresponding probabilities of small number of \( i \) Poisson events
occurring become less significant. This allows us to start the summation from the \( l \)th
iteration called left truncation point with the equation 3 reduced to:
\[ p(t) = \sum_{i=l}^{k} \Pi(i) e^{-\alpha t} \frac{(\alpha t)^i}{i!} \]  \hspace{1cm} (6)
with the values of \( l \) and \( k \) equal to:
\[ \sum_{i=0}^{l-1} e^{-\alpha t} \frac{(\alpha t)^i}{i!} \leq \frac{\varepsilon}{2}, 1 - \sum_{i=0}^{k} e^{-\alpha t} \frac{(\alpha t)^i}{i!} \leq \frac{\varepsilon}{2} \]  \hspace{1cm} (7)

The main computational effort of the algorithm lies in consecutive \( k \) matrix vector
multiplications (MVM) in (4). It can be significantly reduced in case of a converging \( \Pi(i) \),
with a limiting distribution \( \Pi(\infty) \), when achieving of the steady-state \( \Pi(S) \) can be predicted,
as proposed in [2], at \( i \) (where \( i < S < l \)) within some predefined error (steady-state detection
threshold):
\[ \frac{\| \Pi(S) - \Pi(\infty) \|_{\infty}}{\| \Pi(\infty) \|_{\infty}} < \delta \]  \hspace{1cm} (8)
3. Computation of Discrete Poisson Probabilities

The computation of the discrete Poisson distribution is crucial for the correct and efficient implementation of the uniformization algorithm. The discrete probabilities:

$$\beta(\alpha t, k) = e^{-\alpha t} \frac{(\alpha t)^k}{k!} \quad (9)$$

used for calculation of the state probability vector $p(t)$ in [8] have to be calculated precisely, in order to produce its correct final value. Moreover, the precise estimation of the truncation points influences directly the performance of the computation either because of a reduced number of MVMs due to the tight right truncation $k$ or as a result of the detection of steady-state occurring before the left truncation point $l$.

The common method of calculating the discrete Poisson density function in the uniformization algorithm is the method proposed in [3] which approximates both truncation points $l$ and $k$ in advance. It calculates further the resulting number of weights $w[i], l < i < k$ starting from a mode $m = \left\lfloor \alpha t \right\rfloor$ recursively and the corresponding total weight $W$ such that $\beta(\alpha t, i) = w[i]/W$ and $\sum_i w[i] = 1$.

This approach has two very beneficial properties when used for the uniformization algorithm. Firstly the influence of rounding errors accumulated while calculating the weights is scaled by their value, i.e. the weights for the essential probability mass of the Poisson distribution have the smallest rounding error. Secondly multiplication of the often very small probabilities of the consecutive stochastic vectors $\Pi$ by relatively big values of weights prevents underflow to subnormal values or even to zero. This could otherwise occur when multiplying them by the actual discrete probabilities of the Poisson distribution, causing in effect a further reduction of the accuracy.

Another factor that could contribute to numerical inaccuracy of the Fox-Glynn algorithm is the error of calculation of the total weight, induced by adding the values of weights, which are of very different magnitudes. This, however, is practically eliminated by estimating the number of weights in advance and adding their values in increasing order (for much more detailed analysis of the error of sum of exponentially distributed numbers in increased order also [9]).

Unfortunately, implementing the original algorithm as described in [3] can be quite challenging e.g. as explained in [5]. It is mainly because of separate handling of many different cases corresponding to the different $\lambda$ values and the corresponding estimated bounds, dependent on chosen truncation limits for the distribution tails $\varepsilon$ (e.g. $0 < \lambda < 25$,
\( R < 400 \) etc.) due to the different approximations for the truncation points used for different \( \lambda \) values.

Further, the different rigorous tests for possible underflow were, at the time the article was written, justified by available implementations of single precision arithmetic, where a single underflow could stop entire application and there was no guarantee for standardized handling of special cases (e.g. authors analyze both - rounding and chopping of least significant bits when discussing the round-off errors).

Another possible improvement could be, as already suggested by Fox and Glynn in concluding remarks of the original paper, to refine the accuracy of the estimation of the truncation limits.

This is, as already mentioned, particularly significant for the computational efficiency of the uniformization with steady-state detection, and was the main motivation of rewriting of the algorithm for the modern standardized programming language (C++) assuming IEEE 754 conformity of the floating point formats and computation, while preserving its advantageous performance, stability and accuracy properties.

Another alternative algorithm, proposed in [7] and using similar method for calculating the weights values, enables exact bounding as well, as it calculates the remaining tail probability mass parallel to the calculation of the weights. Unfortunately, it does not allow to estimate the number of weights (that have to be stored in order to reuse them for the uniformization algorithm) in advance and introduces bigger rounding error due to a different calculation method for the normalization of the weights.

### 3.1. Implementation details

The algorithm presented in Appendix is a corrected version of the algorithm firstly presented in the unpublished authors paper, available at arxiv [2]. It assumes, that some basic floating point arithmetic is implemented conforming to IEEE 754-2008 standard (e.g. GCC Default IEEE 754 compliance on Intel CPUs). In particular, the numerical accuracy and stability analysis (rounding, underflow and overflow) assumes floating point formats implemented as binary32, binary64 or binary128 (radix 2), correct rounding to the nearest even for basic arithmetic operations and correct conversion from integers.

The algorithm computes for given values \( \lambda \) (at in the uniformization algorithm), \( \varepsilon \) and \( \varepsilon_{ssd} \): the limits \( l_{ssd}, l, k \), the normalization value \( W \) and the weights \( w(j), j \in \{l_{ssd}, k\} \), such that the individual Poisson probabilities: \( p(i) = w(i) / W, i \in \{l..k\} \) and \( w(S) / W = \varepsilon_S, S \in \{l_{ssd}, l-1\} \), where \( \varepsilon_S \) is equal to the cdf of the discrete Poisson distribution function, as in [2].
If $\varepsilon_{\text{rel}} = \varepsilon$, only the discrete Poisson probabilities are calculated and stored (e.g. in cases of a non converging system), like for the traditional uniformization algorithm.

The algorithm computes the weights recursively starting from mode $m = \lfloor \lambda \rfloor$, with $w(m)$ being the initial weight, similarly to [3]. It uses the (double precision) IEEE 754-2008 binary64 format with $p=53$ bit precision and $w=11$ bit exponent. The initial value used for $w(m)$ is a round binary number (exponent equal to 0x1.0), $2^{124}$ times smaller than the biggest representable binary64 number – similar to the heuristic proposal in [3] where the proposed divisor was equal to the number of weights multiplied by $10^3$ in order to prevent overflow while calculating the total weight.

The calculation of every weight, both left and right to $m$, requires one multiplication and one division. As we assume basic IEEE 754-2008 conformity, the relative rounding error for every next $n$'th calculated weight will be upper bounded by $(1 + 2^{-53})^{2^n-2} - 1$.

The rounding error of $W$ is bounded by $(\text{start} - \text{end})\text{eps}$, where $\text{eps} = 2^{-53} \approx 1.11\times10^{-16}$ is the machine epsilon for binary64, but is in reality much smaller due to the summation performed in increasing order and the already mentioned specific distribution of the values of the weights.

To bound the number of calculated weights the algorithm uses the property of the Poisson distribution having truncation points being asymptotically of $O(\sqrt{\lambda})$ for constant remaining cdf(tail) value and approximates the required number of temporary weights using simple function of $\sqrt{\lambda}$. The parameters of the approximation are chosen such, that for the preset minimal truncation threshold the value of corresponding remaining probability mass in the tail will be calculated with precision better than $1\times10^{-3}$. The temporary weights are used to calculate the total weight normalization value, which is then used to estimate precise truncation point for the given value $\varepsilon$ bigger than the minimal truncation threshold for the algorithm. This is done by subsequent addition in increasing order. The number of operations of addition grows with the truncation threshold but as it involves repeated operation on consecutive memory locations its cost is relatively small on modern CPU architectures (i.e. 1 CPU cycle per addition due to pipelining).

The number of temporary weights for the calculation of the normalization value $W = \sum_{\text{start}}^{\text{end}} w()$ is chosen so that the remaining (weighted) cdf is always smaller than the numerical accuracy of $W$: $\Pr(X_\text{., start}) + \Pr(X_\text{., end}) < \text{eps}$,
this results in elimination of the normalization error of the weights, i.e. the calculated Poisson values are not larger than the actual ones as in the original algorithm. Another alternative modification that avoids the normalization error is discussed in [10].

Consequently, \( \sum_{i} p(i) = 1 - \varepsilon \) and is not always equal to one, as in [3].

The values of the weights and the total weight \( W \) are, due to the choice of \( w[m] \) and the bounding by the approximation, always between under- and overflow limits of binary64 format. The implemented algorithm was tested for \( \lambda \) values up to \( 4 \times 10^{12} \) for the tail truncation thresholds \( \varepsilon \) down to \( 1 \times 10^{-50} \).

The temporary weights are stored dynamically (on the stack) in order to avoid unnecessary memory allocation by the OS functions calls and to speed-up the operations on weights. Only the weights between the calculated truncation points are finally stored permanently into RAM. The computational complexity is due to the bounding method of \( O(\sqrt{\lambda}) \).

### 3.2. Computational Examples

The proposed algorithm was compared with the implementation of the original Fox-Glynn algorithm from the Markov Reward Model Checker v1.5 package maintained under GNU General Public License v2 by the Software Modeling and Verification group at the RWTH Aachen (Germany) led by Prof. dr. ir. Joost-Pieter Katoen [6].

The improvement in the estimation of the truncation points for some chosen \( \lambda \) values is presented in the Table 1. The overall performance gain for the uniformization algorithm, due to the tighter bond estimation, will be about 10% in transient cases for system sizes analyzed in [2]. For bigger \( \lambda \) values the results of both algorithms are very similar (less than 1% difference for \( \lambda \geq 1 \times 10^6 \). The gain due to the steady-state detection before the left truncation point will also increase, although it depends, then, on the modeled system and is, therefore, more difficult to quantify.

The performance of both algorithms has been evaluated by experiments of calculating 4444 random uniformly distributed \( \lambda \) values each, for one of three \( \varepsilon \) values of \( 1 \times 10^{-13}, 1 \times 10^{-7}, 1 \times 10^{-5} \). The results for different ranges of randomly generated \( \lambda \) are in the Table 2.

The experiments were performed on a 1.7GHz PC under 64bit Linux OS with a processor supporting SSE 128bit vector operations (an Intel i5-3317U with CPU throttling disabled via kernel scaling governor), compiled with GNU GCC compiler.
Table 1

Comparison of estimated truncation points

<table>
<thead>
<tr>
<th>$\varepsilon = 1 \times 10^{-13}$</th>
<th>Fox-Glynn</th>
<th>Burak 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>L bound</td>
<td>R bound</td>
</tr>
<tr>
<td>0.3</td>
<td>0</td>
<td>228</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>231</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>258</td>
</tr>
<tr>
<td>100</td>
<td>18</td>
<td>328</td>
</tr>
<tr>
<td>300</td>
<td>159</td>
<td>528</td>
</tr>
<tr>
<td>1000</td>
<td>745</td>
<td>1360</td>
</tr>
<tr>
<td>3000</td>
<td>2560</td>
<td>3622</td>
</tr>
<tr>
<td>10000</td>
<td>9198</td>
<td>11133</td>
</tr>
<tr>
<td>$1 \times 10^6$</td>
<td>991998</td>
<td>1011316</td>
</tr>
</tbody>
</table>

Table 2

Comparison of performance

<table>
<thead>
<tr>
<th>$\varepsilon \in {1 \times 10^{-13}, 1 \times 10^{-7}, 1 \times 10^{-5}}$</th>
<th>Fox-Glynn</th>
<th>Burak 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$ range</td>
<td>ms</td>
<td>ms</td>
</tr>
<tr>
<td>33.0 - 65.0</td>
<td>70.38</td>
<td>20.11</td>
</tr>
<tr>
<td>65.0 - 129.0</td>
<td>72.50</td>
<td>23.53</td>
</tr>
<tr>
<td>129.0 - 257.0</td>
<td>72.30</td>
<td>31.79</td>
</tr>
<tr>
<td>1025.0 - 2049.0</td>
<td>111.49</td>
<td>87.99</td>
</tr>
<tr>
<td>16387.0 - 32774.0</td>
<td>429.69</td>
<td>323.45</td>
</tr>
<tr>
<td>524289.0 - 1048578.0</td>
<td>2465.46</td>
<td>1886.98</td>
</tr>
<tr>
<td>2097153.0 - 4194305.0</td>
<td>5006.76</td>
<td>3818.70</td>
</tr>
<tr>
<td>16777777.0 - 335555553.0</td>
<td>14181.37</td>
<td>10821.12</td>
</tr>
</tbody>
</table>

All measurements use standard Unix `time.h/clock()` function - returning CPU time. All times are in milliseconds.

The performance gain has been achieved mainly due to the modification of calculation of the total weight in order to enable consecutive memory access similar to the refinement of the truncation bounds, as it allowed to use vectorized processing of the CPU (calculating 2 double operations simultaneously). The gain would be even greater if the tests were compiled and run using avx instruction set (with 4 or 8 parallel double operations dependent on CPU type). This modification could be also easily applied to the original Fox-Glynn algorithm. Without the use of vector processing, both algorithms perform very similarly.
4. Conclusion

In this paper we were able to improve the estimation of truncation bounds for the uniformization algorithm with corresponding significant computational savings compared to the popular Fox-Glynn algorithm, while simultaneously preserving its advantageous properties. The resulting algorithm profits from the functionality offered by modern CPUs. It uses in particular the implementation of the floating point operations according to the IEEE 754 standard and use of (double precision) binary64 floating point format instead of the single precision format used by the original algorithm, which enables the algorithm to be much simpler and, therefore, easier to implement, without compromising its precision and functionality.

APPENDIX: Class B14 - Computing Poisson Probabilities

```cpp
#ifndef Poisson_Burak2014
#define Poisson_Burak2014

#include<cmath>
#include<cfloat>
#include<cstdlib>

class B14 {
private:
    long wsize,start;
    long left_l,right_k;
    double *weights;
    double total_weight;
public:
    //constructor - creates weights
    B14(const double Lambda, double Epsilon, double Epsilon_ssd);
    long LS(){return start;}   //left truncation point for ssd epsilons
    long L() {return left_l;}  //left truncation point
    long R() {return right_k;} //right truncation point
    //weights value by the n-th poisson probability
    double p(const double value, const long n)
    {if(value > total_weight)
    return((n>=left_l)&&(n<(wsize+start)) ?
            (value/total_weight)*weights[n-start] : 0.0);
    return((n>=left_l)&&(n<(wsize+start)) ?
            (value*weights[n-start])/total_weight : 0.0);};
    //returns S-th cdf
    double essd(const long S)
    {return ((S>=start) && (S<left_l) ?
            weights[S-start]/total_weight : 0.0);};
    //returns n-th weight, use together with W()
    //for weighted sums of small values e.g. probabilities:
    //sum_i( weight(i) * p_i ) / W()
    double weight(const long n)
    {return ((n>=start) && (n<wsdirme+start) ?
        weights[n-start] : 0.0);}

    #endif
}```
double W(){return total_weight;};

~B14(){free(weights);}; //end of B14 class

B14::B14(const double Lambda, double Epsilon, double Epsilon_ssd)
    if(!(Lambda > 0.0)||!(Epsilon<1.0d))
        (left_l=right_k=0;weights=&total_weight;
            total_weight=nan(""));return;
    if(!(Epsilon>1e-13))Epsilon=1e-13;
    if(!(Epsilon_ssd<1e-16))Epsilon_ssd=1e-16;
    if(!(Epsilon_ssd<Epsilon))Epsilon_ssd=Epsilon;  //LS=L essd=0.0
    long m=(long)floor(Lambda);
    //number of temporary weights stored
    //long mw = 31;long ma = 80;long ms = 40; //for Epsilon > 1e-50
    //long mw = 25;long ma = 47;long ms = 23; //for Epsilon > 1e-30
    //long mw = 20;long ma = 34;long ms = 17; //for Epsilon > 1e-20
    //long mw = 19;long ma = 26;long ms = 13; //for Epsilon > 1e-16
    long mw = 17;long ma = 24;long ms = 11; //for Epsilon > 1e-13
    long tsize=(long)(sqrt(Lambda) * mw) + ma;
    long tstart=m+ms-tsize/2;if(tstart<0)tstart=0;
    //if short of stack and big Lambda: alloca -> malloc+free
    double *tweights=(double*)alloca(tsize*sizeof(double));
    long j=m-tstart;
    tweights[j]=ldexp(0x1.0p0,DBL_MAX_EXP - 124); //weight[m]
    for(j=m-tstart;j>0;j--)
        tweights[j]=(tweights[j]*(j+tstart))/Lambda;
    for(j=m-tstart+1;j<tsize;j++)
        tweights[j]=(Lambda*tweights[j-1])/(j+tstart);
    //compute total_weight
    total_weight=0.0;
    for(j=0;j<(m-tstart);j++)total_weight+=tweights[j];
    double suml=0.0;
    for(j=(tsize-1);j>=(m-tstart);j--)
        suml+=tweights[j];
    total_weight+=suml;
    //calculate truncation points
    double ogon=(Epsilon_ssd*total_weight)/2.0;
    long i=0;
    double cdf=tweights[i];
    while(cdf<ogon) cdf+=tweights[++i];
    start=i+tstart=(i>0);
    double cdf_start=cdf;
    ogon=(Epsilon*total_weight)/2.0;
    while(cdf<ogon) cdf+=tweights[++i];
    left_l=i+tstart=(i>0);
    //if(i==0)ogon*=2; //for historical compatibility
    i=tsize-1;
    cdf=tweights[i];
    while(cdf<ogon) cdf+=tweights[--i];
    right_k=i+tstart+1;
    //only weights/cdfs between LS and R are stored
    wsize=right_k-start+1;
    weights = (double *)malloc(wsize * sizeof(double));
    weights[0]=cdf_start;
for(j=start+1;j<left_l;j++)
weights[j-start]=weights[j-start-1]+tweights[j-tstart];
for(j=left_l;j<=right_k;j++)
weights[j-start]=tweights[j-tstart];
}//end of B14 constructor

BIBLIOGRAPHY

Omówienie

W artykule przedstawiono modyfikację popularnego algorytmu Foxa-Glynna poprawiającą dokładność obliczenia wartości granicznych dyskretnego rozkładu Poisson dla zadanego błędu (ogona dystrybuanty rozkładu), co umożliwia znaczące zmniejszenie ilości operacji mnożenia matryca-wektor determinujących złożoność obliczeniową algorytmu uniformizacji wykorzystywanego do modelowania łańcuchów Markowa z czasem ciągłym, bez pogarszania jego innych korzystnych właściwości. Dodatkowo, zmieniony algorytm ustala wartości prawdopodobieństw ww. rozkładu bez błędu normalizacji (podobnie jak w rozwiązaniu zaproponowanym w [10]) oraz oferuje znacząco prostszą implementację, dzięki wykorzystaniu współczesnych standardowych operacji zmiennoprzecinkowych, zgodnie ze standardem IEEE 754.

Algorytm dzięki odpowiednemu doborowi parametrów wyjściowych nie dopuszcza do wyliczenia wartości niebędących normalnymi liczbami zmiennoprzecinkowymi (e.g. underflow, gradual underflow, overflow).

Addresses

Maciej Rafał BURAK: West Pomeranian University of Technology, Applied Informatics, ul. Sikorskiego 37, 90-313 Szczecin, Poland, Maciej.Burak@zut.edu.pl.