FACTA UNIVERSITATIS (NIŠ)
Ser. Math. Inform. 25 (2010), 35–45

WEIGHTED MOORE-PENROSE INVERSE: PHP vs. MATHEMATICA

Selver H. Pepić

Abstract. In this paper we compare two different implementations of the algorithm for computing the Weighted Moore-Penrose inverse for constant matrices, developed by Wang [6]. The general-purpose scripting language PHP and the symbolic computational package MATHEMATICA, web and desktop applications respectively, are used and both, the advantages and disadvantages are indicated. Several illustrative examples on a test and randomly generated matrices are presented. The numerical results and the CPU execution time are compared.

1. Introduction

For any matrix $A \in \mathbb{C}^{m\times n}$ ($\mathbb{C}$ be the set of complex numbers, $\mathbb{C}^{m\times n}$ be the set of $m \times n$ complex matrices of rank r: $\mathbb{C}_r^{m\times n}$=$\{X \in C^{m\times n} \mid \text{rank}(X) = r\}$) and positive definite Hermitian matrices M and N of the orders m and n respectively, consider the following equations in X, where $*$ denotes conjugate and transpose:

$$\begin{array}{llll} (1) & AXA = A & (2) & XAX = X \\ (3M) & (MAX)^* = MAX & (4N) & (NXA)^* = NXA. \end{array}$$

The matrix X satisfies equations (1), (2), (3M) and (4N) is called the weighted Moore-Penrose inverse of A [6], and it is denoted by $X = A^\dagger_{MN}$. In the partial case $M = I_m$ and $N = I_n$, the matrix $X = A^\dagger_{MN}$ becomes the Moore-Penrose inverse of A, and it is denoted by $X = A^\dagger$.

There are various methods for computing the Moore-Penrose inverse. The main methods are based on the Cayley-Hamilton theorem, the full-rank factorization and the singular value decomposition. The Greville's algorithm is used in various computations, where its dominance is verified over direct methods for the the pseudo-inverse computation [1]. There are a lot of extensions of the partitioning method. Wang in [6] generalized Greville's method to the weighted Moore-Penrose inverse. Effective partitioning method for computing the weighted MoorePenrose

Received November 30, 2010.
2010 *Mathematics Subject Classification.* Primary 68N15; Secondary 97N60, 65F50

inverse is considered in [2]. The Greville's recursive principle is generalized to various subsets of outer inverses and extended to the set of the one-variable rational and polynomial matrices in [4]. This paper is a continuation of paper [5].

The structure of the present paper is as follows. In the second section is introduced the implementation of the partitioning method for computing Weighted Moore Penrose Inverse from [6], using PHP and MATHEMATICA. In the third section are presented a few illustrative examples and the CPU computational times. Also, the advantages and disadvantages of the implementations are indicated.

2. The recursive algorithm for computing the Weighted Moore-Penrose Inverse $A^{\dagger}_{MN}$

Wang and Chen in [6] generalized Grevilles partitioning method. They obtained an algorithm for computing the weighted MoorePenrose inverse ($A^{\dagger}_{MN}$), and give a new technique for its proof. This method is also suitable for the weighted least-squares problem.

Theorem 2.1. (G.R. Wang, Y.L. Chen [6]) *Let $A \in \mathbb{C}^{m\times n}$ and let A_k be the submatrix of A consisting of its first k columns. For $k = 2, \ldots, n$, the matrix A_k is partitioned as $A_k = [A_k - 1|a]$, and the matrix $N_k \in \mathbb{C}^{k\times k}$ is the leading principal submatrix of N, and N_k is partitioned as*

$$N_k = \begin{bmatrix} N_{k-1} & l_k \\ l_k^* & n_{kk} \end{bmatrix}. \tag{2.1}$$

Let the matrices X_{k-1} and X_k be defined by

$$X_{k-1} = (A_{k-1})^{\dagger}_{M,N_{k-1}}, \quad X_k = (A_k)^{\dagger}_{M,N_k}, \tag{2.2}$$

the vectors d_k, c_k be defined by

$$d_k = X_{k-1}a_k \tag{2.3}$$

$$c_k = a_k - A_{k-1}d_k = (I - A_{k-1}X_{k-1})\,a_k. \tag{2.4}$$

Then

$$X_k = \begin{bmatrix} X_{k-1} - (d_k + (I - X_{k-1}A_{k-1})\,N_{k-1}^{-1}l_k)b_k^* \\ b_k^* \end{bmatrix}, \tag{2.5}$$

where

$$b_k^* = \begin{cases} \left(c_k^* M c_k\right)^{-1} c_k^* M, & c_k \neq 0 \\ \delta_k^{-1}\left(d_k^* N_{k-1} - l_k^*\right) X_{k-1}, & c_k = 0, \end{cases} \tag{2.6}$$

and $\delta_k = n_{kk} + d_k^* N_{k-1} d_k - \left(d_k^* l_k + l_k^* d_k(s)\right) - l_k^*\,(I - X_{k-1}A_{k-1})\,N_{k-1}^{-1} l_k$.

According to the above theorem we restate the Algorithm from [6].

Algorithm 2.1 (G.R. Wang, Y.L. Chen) Computing the weighted M-P inverse $A^{\dagger}_{M,N}$.

Require: Let $A \in \mathbb{C}^{m\times n}$, M and N be p.d. matrices of the order m and n respectively.

```
1: $A_1 = a_1$.
2: if $a_1 = 0$, then
3:     $X_1 = (a_1^* M a_1)^{-1} a_1^* M$;
4: else
5:     $X_1 = 0$.
6: end if
7: for $k = 2$ to $n$ do
8:     $d_k = X_{k-1} a_k$,
9:     $c_k = a_k - A_{k-1} d_k$,
10:    if $c_k \neq 0$, then
11:        $b_k^* = (c_k^* M c_k)^{-1} c_k^* M$, goto Step 16,
12:    else
13:        $\delta_k = n_{kk} + d_k^* N_{k-1} d_k - (d_k^* l_k + l_k^* d_k) - l_k^* (I - X_{k-1} A_{k-1}) N_{k-1}^{-1} l_k$,
14:        $b_k^* = \delta_k^{-1} (d_k^* N_{k-1} - l_k^*) X_{k-1}$,
15:    end if
16:    $X_k = \begin{bmatrix} X_{k-1} - (d_k + (I - X_{k-1}A_{k-1}) N_{k-1}^{-1} l_k) b_k^* \\ b_k^* \end{bmatrix}$.
17: end for
18: return $A^{\dagger}_{MN} = X_n$.
```

Also, the next auxiliary Algorithm 2.2, required in Algorithm 2.1 is stated in [6].

Algorithm 2.2 (G.R. Wang, Y.L. Chen.) Computing the inverse matrix N^{-1}.

Require: Let $N_k = \begin{bmatrix} N_{k-1} & l_k \\ l_k^* & n_{kk} \end{bmatrix} \in \mathbb{C}^{k\times k}$ be the leading principal submatrix of p.d. matrix N.

```
1: $N_1^{-1} = n_{11}^{-1}$.
2: for $k = 2$ to $n$ do
3:     $g_{kk} = (n_{kk} - l_k^* N_{k-1}^{-1} l_k)^{-1}$,
4:     $f_k = -g_{kk} N_{k-1}^{-1} l_k$,
5:     $E_{k-1} = N_{k-1}^{-1} + g_{kk}^{-1} f_k f_k^*$,
6:     $N_k^{-1} = \begin{bmatrix} E_{k-1} & f_k \\ f_k^* & g_{kk} \end{bmatrix}$.
7: end for
8: return $N^{-1} = N_n^{-1}$.
```

Algorithm 2.1, for computing the weighted Moore-Penrose inverse $A^{\dagger}_{MN}$ for constant matrices, is implemented in PHP and MATHEMATICA.

The package MATHEMATICA is used for symbolic computation mathematical expressions and enable simplifications of rational expressions. For the final processing of expressions is used a very powerful MATHEMATICA function *Simplify*. We note that for MATHEMATICA is typical high cost and robustness.

As opposed MATHEMATICA, PHP is a scripting language that has no ready-made functions for mathematical calculation or the final simplification of rational expressions. PHP application is characterized by the following properties, which are not in valid in MATHEMATICA:

- possibility to distribute to multiple computers simultaneously; independence of the platform; it is an open source; it is not robust and does not take the memory in the computer; since it is installed on the server; it does not require an expensive installation on the computer, but only internet browsers and internet connections.

2.1. PHP and MATHEMATICA implementation details for constant matrices

Now we will compare the equivalent procedures written in PHP and MATHEMATICA package for the computation of Algorithm 2.1. Some procedures written in PHP are contained in the computational package MATHEMATICA. In the beginning are described and compared several auxiliary procedures.

P1 - Generate the jth column a_j of A:

A. PHP implementation case

```
function IthCol($col,$ArrayDataMatrix){
   $m=count($ArrayDataMatrix); $n=count($ArrayDataMatrix[0]);
    for($i=0;$i<$m;$i++){
        for($j=0;$j<$n;$j++){
            if($j+1==$col)
               $ArrayCol[$i][0]= $ArrayDataMatrix[$i][$j];}}
    return $ArrayCol;}
```

B. MATHEMATICA implementation case

```
TakeCol[A_, k_] := Transpose[Take[Transpose[A], {k}]];
```

P2 - The submatrix A_j=[$a_{1,...,}a_j$] of the matrix $A = A_n$:

A. PHP implementation case

```
function FrstICol($columns,$ArrayDataMatrix){
   $m=count($ArrayDataMatrix); $n=count($ArrayDataMatrix[0]);
    for($i=0;$i<$m;$i++){
        for($j=0;$j<$n;$j++){
            if($j+1<=$columns)
               $ArrayCol[$i][$j]= $ArrayDataMatrix[$i][$j];}}
    return $ArrayCol;}
```

B. MATHEMATICA implementation case

```
TakeCols[A_, k_] := Transpose[Take[Transpose[A], k]];
```

P3 - Separate the scalar n_{kk} from N_k:

A. PHP implementation case

```
function MSskalar($ArrayDataMatriz,$k){
  $skalar=$ArrayDataMatriz[$k-1][$k-1];
  return $skalar;}
```

B. MATHEMATICA implementation case

```
TakeElement[A_, i_, j_] := TakeCol[Take[A, {i}], j];
```

P4 - Separate the positive definite $(n-1)$ by $(n-1)$ matrix N_{k-1} from N_k:

A. PHP implementation case

```
function MSskalar($ArrayDataMatriz,$k){
  $skalar=$ArrayDataMatriz[$k-1][$k-1];
  return $skalar;}
```

B. MATHEMATICA implementation case

```
TakeElements[A_, i_, j_] := TakeCols[Take[A, i], j];
```

Other procedures, for computing weighted Moore-Penrose inverse are contained in MATHEMATICA, but in PHP they should be implemented (for example *AppRow()*, *SubsMatrices()*, *Transpose()*, *ScalarMatrix()*, *MatricesMultiplication()*, *Inverse()*, etc.).

P5 - Procedure *MatNum()* returns from matrix $A_{1\times1}$ value of its single element.

```
function MatNum($ArrayDataMatrix){
   $m=count($ArrayDataMatrix); $m=count($ArrayDataMatrix[0]);
   if ($m==1&$m==1)
   return $ArrayDataMatrix[0][0];}
```

P6 - Function *IsZero()* determines whether the matrix zero matrix or not.

```
function IsZero($ArrayDataMatrix){
   $isZero=0; $m=count($ArrayDataMatrix); $n=count($ArrayDataMatrix[0]);
   for($i=0;$i<$m;$i++) {
      for($j=0;$j<$n;$j++){
         if(round($ArrayDataMatrix[$i][$j],3)!=0)
            $isZero++;}}
   return $isZero;}
```

P7 - Function *AppRow()* adds matrix Y to a matrix X as $m+1$ column. Matrix Y is a vector column.

```
function AppRow($ArrayDataMatrix1,$ArrayDataMatrix2){
   $isZero=0; $m1=count($ArrayDataMatrix1);
   $n2=count($ArrayDataMatrix2[0]);
   for($i=0; $i<$n2; $i++){
      $ArrayDataMatrix1[$m1][$i]=$ArrayDataMatrix2[0][$i];   }
   return $ArrayDataMatrix1;}
```

P8 - The matrices substitution is given by the following procedure (matrices have equal dimensions).

```
function  SubsMatrices($ArrayDataMatrix1 ,$ArrayDataMatrix2){
   $m1=count($ArrayDataMatrix1);  $n1=count($ArrayDataMatrix1[0]);
   for($i=0;$i<$m1;$i++)  {
      for($j=0;$j<$n1;$j++){
         $ArraySubs[$i][$j]=$ArrayDataMatrix1[$i][$j]
                -$ArrayDataMatrix2[$i][$j];}}
   return  $ArraySubs;}
```

P9 - Transpose of the matrix is given by:

```
function  Transpose($ArrayDataMatrix)  {
   $m=count($ArrayDataMatrix);  $n=count($ArrayDataMatrix[0]);
   $ArrayTranspose=array();
   for($i  =  0;$i<$filas;$i++){
      for($j=0;$j<$columnas;$j++){
         $ArrayTranspose[$j][$i]=$ArrayDataMatrix[$i][$j];}}
   return  $ArrayTranspose;}
```

P10 - The multiplication matrix with scalar is given by following

```
function  ScalarMatriz($ArrayDataMatrix ,$scalaar)  {
   $m=count($ArrayDataMatrix);  $n=count($ArrayDataMatrix[0]);
   $matrix=array();
   for($i=0;$i<$m;$i++){
      for($j=0;$j<$n;$j++)  {
         $matrix[$i][$j]=$ArrayDataMatrix[$i][$j]*$scalar;}}
   return  $matrix;}
```

P11 - The matrix multiplication (number of columns of the first matrix is equal to the number of the second rows matrix) is implemented by the following procedure.

```
function  MultiplicationMatrices($ArrayDataMatrix1 ,$ArrayDataMatrix2)  {
   $m1=count($ArrayDataMatrix1);  $n1=count($ArrayDataMatrix1[0]);
   $n2=count($ArrayDataMatrix2[0]);  $m2=count($ArrayDataMatrix2);
   for($i=0;$i<$m1;$i++){
      for($j=0;$j<$n2;$j++){
         $ArrayMultipli[$i][$j]=0;$sum=0;
            for($M=0;$M<$n1;$M++){
               $ArrayMultipli[$i][$j]=$ArrayMultipli[$i][$j]+
               $ArrayDataMatrix1[$i][$M]*$ArrayDataMatrix2[$M][$j];}}}
    return  $ArrayMultipli;}
```

Finally, the implementation of the Algorithm 2.1 is given by the following functions in PHP and MATHEMATICA.

A. PHP implementation case

```
function  WeightedInverse($ArrayDataMatrixM,
                          $ArrayDataMatrixA ,$ArrayDataMatrixN)
   {$m=count($ArrayDataMatrixA);  $n=count($ArrayDataMatrixA[0]);
   $aa=IthCol(1,$ArrayDataMatrizA);
   if(IsZero($aa)==0)  $ar=Transpuesta($aa);
   else{
      $ta=Transpuesta($aa);
      $alb=MultiplicationMatrices($ta ,$ArrayDataMatrixM);
      $ali=MultiplicationMatrices($alb ,$aa);
      $inv=Inverse($ali);$ar=MultiplicacionMatrices($inv ,$alb);}
   for($i=2;$i<=$n;$i++){
    $ii=IthCol($i,$ArrayDataMatrixA);$di=MultiplicationMatrices($ar,$ii);
    $fc=FrstiCol($i-1,$ArrayDataMatrixA);
    $mm=MultiplicationMatrices($fc,$di);
    $ci=SubsMatrices($ii,$mm); $nim1=MMinus($ArrayDataMatrixN,$i);
    $nim1g=Inverse($nim1); $li=MColumn($ArrayDataMatrixN,$i);
    if(isZero($ci)==0){
      $nii=MSkalar($ArrayDataMatrixN,$i);
      $dit=Transpose($di); $dtn=MultiplicationMatrices($dit,$nim1);
      $dtnd=MultiplicationMatrices($dtn,$di);
      $dtnd=MatNum($dtnd); $ditl=MultiplicationMatrices($dit,$li);
      $lit=Transpose($li); $litdi=MultiplicationMatrices($lit,$di);
      $litdi=ScalarMatrix($litdi,-1); $ditdi=SubsMatrices($ditl,$litdi);
      $ditdi=MatNum($ditdi); $nim1li=MultiplicationMatrices($nim1g,$li);
      $litnli=MultiplicationMatrices($lit,$nim1li);
      $litnli=MatNum($litnli); $lktar=MultiplicationMatrices($lit,$ar);
      $arnklk=MultiplicationMatrices($fc,$nim1li);
      $novo=MultiplicationMatrices($lktar,$arnklk);
      $novo=MatNum($novo); $si=$nii+$dtnd-$ditdi-$litnli+$novo;
      $dtnar=MultiplicationMatrices($dtn,$ar);
      $dilit=SubsMatrices($dtnar,$lktar);
      $bit=ScalarMatrix($dilit,1/$si);}
   else{ $cit=Transpose($ci);
      $citm=MultiplicationMatrices($cit,$ArrayDataMatrixM);
      $pom=MultiplicationMatrices($citm,$ci); $pom=MatNum($pom);
      $bit=ScalarMatrix($citm,1/$pom); }
      $nim1li=MultiplicationMatrices($nim1g,$li);
      $arai=MultiplicationMatrices($ar,$fc);
      $arnim1=MultiplicationMatrices($arai,$nim1li);
      $pi=SubsMatrices($nim1li,$arnim1);
      $k1=MultiplicationMatrices($di,$bit);
      $ark1=SubsMatrices($ar,$k1);$pib=MultiplicationMatrices($pi,$bit);
      $ar=SubsMatrices($ark1,$pib);  $ar=AppRow($ar,$bit);}
   return $ar;}
```

Implementation of the Algorithm 2.2 is given by the following function.

```
function Inverse($ArrayDataMatrix){
   $m=count($ArrayDataMatrix); $n=count($ArrayDataMatrix[0]);
   $nii=MScalar($ArrayDataMatrix,1);
   $N=1/$nii;$N=array(array($N));
   for($i=2;$i<=$n;$i++){ $s=MScalar($ArrayDataMatrix,$i);
    $nii=array(array($s)); $li=MColumn($ArrayDataMatrix,$i);
    $lit=Transpose($li); $giip=MultiplicationMatrices($lit,$N);
    $gii1=MultiplicationMatrices($giip,$li);
    $gii=SubsMatrices($nii,$gii1);  $gii2=MatNum($gii);
    $gii=1/$gii2;  $fip=MultiplicationMatrices($N,$li);
    $fi1=ScalarMatrix($fip, $gii); $fi=ScalarMatrix($fi1,-1);
    $fit=Transpose($fi); $E1=MultiplicationMatrices($fi,$fit);
    $E2=ScalarMatrix($E1,1/$gii); $E2=ScalarMatrix($E2,-1);
    $E=SubsMatrices($N,$E2); $ArrayDataMatrix1=AppRow($E,$fit);
    $giin=array(array($gii));
    $ArrayDataMatrix2=AppRow($fi,$giin);
    $N=Finaly($ArrayDataMatrix1,$ArrayDataMatrix2);}
   return $N;}
```

B. MATHEMATICA implementation case

Details concerning the implementation of the partitioning algorithm corresponding to the weighted Moore-Penrose inverse can be found in [3]. For implementation of Algorithm 2.2 in MATHEMATICA is incorporated procedure *Inverse*.

```
AWang[A_,M_,N_]:=Module[{I,m,n,d,c,a,A1,l,del,X,tmp,tmp1,N1,n11},
   a = TakeCol[A, 1]; {m, n} = Dimensions[A];
   If[Together[a]===0*a,X=T[a],X=Inverse[T[a].M.a].T[a].M ];A1={};
   Do[I=IdentityMatrix[i-1]; a=TakeCol[A,i];A1=TakeCols[A,i-1];
     d=X.a; c=Together[a-A1.d]; n11= TakeElement[N,i,i];
     N1=TakeElements[N,i-1,i-1];l=T[TakeCols[Take[N,{i}],i-1]];
     tmp=T[d].N1-T[l]; tmp1=(I-X.A1).Inverse[N1].l//Together;
      If[Together[c]=!=0*c, b=Inverse[T[c].M.c].T[c].M;
        b=T[b]// Together,
        del=n11+T[d].N1.d-(T[d].l+T[l].d)-T[l].tmp1//Together;
        b=T[Inverse[del].tmp.X]//Together];
     X = Together[Join[X-(d+tmp1).T[b], T[b]]];
    ,{i, 2, n}]; Return[X]];
```

3. Examples and CPU execution time

In order to achieve the main idea, the auxiliary examples and the CPU times for computation of the weighted Moore-Penrose inverse, are consider.

Example 3.1. Consider the test matrix $A_{11\times10}$ from [7], in the case $a = 1$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 1 & 1 & 3 & 4 & 6 & 2 \\ 1 & 3 & 4 & 6 & 2 & 2 & 3 & 4 & 5 & 3 \\ 2 & 3 & 4 & 5 & 3 & 3 & 4 & 5 & 6 & 4 \\ 3 & 4 & 5 & 6 & 4 & 4 & 5 & 6 & 7 & 6 \\ 4 & 5 & 6 & 7 & 6 & 6 & 6 & 7 & 7 & 8 \\ 6 & 6 & 7 & 7 & 8 & 1 & 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 1 & 3 & 4 & 6 & 2 & 1 & 2 \\ 4 & 6 & 2 & 2 & 3 & 4 & 5 & 3 & 1 & 3 \\ 4 & 5 & 3 & 3 & 4 & 5 & 6 & 4 & 2 & 3 \\ 5 & 6 & 4 & 4 & 5 & 6 & 7 & 6 & 3 & 4 \\ 6 & 7 & 6 & 6 & 6 & 7 & 7 & 8 & 4 & 5 \end{bmatrix}_{11\times10}$$

and randomly generated symmetric positive definite matrices $M_{10\times10}$ and $N_{11\times11}$:

$$M = \begin{bmatrix} 499 & -151 & 53 & 129 & -69 & -85 & -131 & 7 & 109 & 502 \\ -151 & 453 & 78 & -176 & 96 & 58 & 65 & -114 & -264 & 22 \\ 53 & 78 & 474 & 43 & -15 & -150 & -99 & 20 & -32 & 28 \\ 129 & -176 & 43 & 407 & -176 & -131 & 36 & 16 & 20 & -93 \\ -69 & 96 & -15 & -176 & 522 & 121 & 103 & -73 & -25 & 172 \\ -85 & 58 & -150 & -131 & 121 & 541 & 45 & 397 & -256 & 3 \\ -131 & 65 & -99 & 36 & 103 & 45 & 147 & -88 & -9 & -3 \\ 7 & -114 & 20 & 16 & -73 & 397 & -88 & 573 & -281 & -96 \\ 109 & -264 & -32 & 20 & -25 & -256 & -9 & -281 & 538 & 43 \\ 502 & 22 & 28 & -93 & 172 & 3 & -3 & -96 & 43 & 271 \end{bmatrix}_{10\times10},$$

$$N=\begin{bmatrix} 633 & -80 & 129 & 77 & -66 & 109 & 142 & -95 & 9 & -133 & -16 \\ 91 & 413 & -74 & -119 & -317 & -41 & -12 & 67 & 180 & -53 & 54 \\ 129 & 15 & 370 & 243 & 109 & -87 & -179 & -54 & -195 & 145 & 16 \\ 77 & 76 & 243 & 653 & -18 & -98 & -163 & 133 & -112 & 274 & -11 \\ -66 & -38 & 109 & -18 & 351 & -179 & 9 & -39 & -1 & -66 & 22 \\ 109 & 38 & -87 & -98 & -179 & 468 & -38 & -131 & -3 & -71 & -38 \\ 142 & -261 & -179 & -163 & 9 & -38 & 467 & -68 & 230 & -141 & -288 \\ -95 & 135 & -54 & 133 & -39 & -131 & -68 & 404 & 10 & 5 & 37 \\ 9 & -171 & -195 & -112 & -1 & -3 & 230 & 10 & 387 & -315 & -127 \\ -133 & 200 & 145 & 274 & -66 & -71 & -141 & 5 & -315 & 484 & 21 \\ -16 & 219 & 16 & -11 & 22 & -38 & -288 & 37 & -127 & 21 & 352 \end{bmatrix}_{11\times11}.$$

The weighted Moore-Penrose inverse $A^{\dagger}_{MN}$ is equal to:

$$A^{\dagger}_{MN}=\begin{bmatrix} -0.284 & -0.214 & -0.409 & 1.373 & -0.728 & 0.043 & -0.314 & -0.066 & 0.716 & -0.206 & -0.011 \\ 0.587 & -0.058 & 0.397 & -1.508 & 0.802 & 0 & 0.175 & 0.584 & -0.466 & -0.485 & 0.254 \\ -0.286 & -0.122 & -0.839 & 2 & -1.031 & -0.01 & -0.458 & -0.482 & 1.734 & -0.843 & 0.204 \\ -0.377 & 0.507 & 0.017 & 0.043 & -0.116 & 0.016 & -0.012 & -0.025 & -0.275 & 0.533 & -0.268 \\ -0.108 & -0.002 & 0.528 & -1.256 & 0.716 & 0.108 & 0.407 & 0.006 & -1.233 & 0.81 & -0.202 \\ 1.732 & -0.496 & 0.179 & -3.362 & 2.211 & -0.102 & 0.488 & 0.319 & 0.908 & -2.865 & 1.346 \\ -1.014 & 0.386 & -0.472 & 2.394 & -1.498 & 0.006 & -0.178 & -0.508 & 0.166 & 1.397 & -0.771 \\ -0.457 & 0.165 & 0.32 & 0.334 & -0.45 & -0.021 & -0.082 & -0.051 & -1.328 & 1.652 & -0.313 \\ 1.206 & -0.498 & 0.341 & -1.984 & 1.293 & 0.005 & 0.36 & 0.325 & 0.211 & -1.52 & 0.609 \\ -0.98 & 0.165 & -0.157 & 1.857 & -0.973 & -0.021 & -0.381 & -0.051 & -0.029 & 0.951 & -0.611 \end{bmatrix}.$$

The Moore-Penrose inverse can be generated in the case $M = \alpha I$, $N = \beta I$, and it is equal to

$$A^{\dagger}=\begin{bmatrix} 0.294 & -0.169 & -1.511 & 1.415 & -0.391 & 0.041 & -0.04 & -0.26 & 0.454 & -0.264 & 0.23 \\ -0.067 & -0.074 & 1.227 & -1.064 & 0.23 & 0.001 & -0.192 & 0.649 & -0.103 & -0.191 & -0.102 \\ 0.227 & -0.179 & -0.692 & 0.705 & -0.214 & -0.009 & -0.254 & -0.238 & 1.514 & -1.319 & 0.449 \\ -0.165 & 0.547 & -0.657 & 0.376 & -0.116 & 0.015 & 0.179 & -0.196 & -0.456 & 0.531 & -0.104 \\ -0.297 & -0.01 & 1.035 & -0.972 & 0.359 & 0.108 & 0.238 & 0.044 & -1.065 & 0.938 & -0.366 \\ -0.008 & -0.474 & 1.663 & -1.319 & 0.352 & -0.103 & -0.428 & 0.224 & 1.83 & -1.844 & 0.414 \\ 0.062 & 0.368 & -1.349 & 1.081 & -0.33 & 0.006 & 0.426 & -0.434 & -0.442 & 0.712 & -0.156 \\ -0.081 & 0.158 & 0.029 & -0.144 & -0.034 & -0.021 & 0.087 & -0.019 & -1.499 & 1.448 & -0.138 \\ 0.195 & -0.484 & 1.198 & -0.791 & 0.211 & 0.005 & -0.19 & 0.268 & 0.765 & -0.906 & 0.049 \\ -0.081 & 0.158 & -0.971 & 0.856 & -0.034 & -0.021 & 0.087 & -0.019 & -0.499 & 0.448 & -0.138 \end{bmatrix}.$$

Example 3.2. The following example demonstrates the power of the symbolic computation, and especially the final process (simplifications of rational expressions) in the computational package MATHEMATICA. Consider the one variable test matrix

$$A=\begin{bmatrix} 10+x & 9+x & 8+x & 7+x & 6+x & 5+x & 4+x & 3+x & 2+x & 1+x \\ 9+x & 9+x & 8+x & 7+x & 6+x & 5+x & 4+x & 3+x & 2+x & 1+x \\ 8+x & 8+x & 8+x & 7+x & 6+x & 5+x & 4+x & 3+x & 2+x & 1+x \\ 7+x & 7+x & 7+x & 7+x & 6+x & 5+x & 4+x & 3+x & 2+x & 1+x \\ 6+x & 6+x & 6+x & 6+x & 6+x & 5+x & 4+x & 3+x & 2+x & 1+x \\ 5+x & 5+x & 5+x & 5+x & 5+x & 5+x & 4+x & 3+x & 2+x & 1+x \\ 4+x & 4+x & 4+x & 4+x & 4+x & 4+x & 4+x & 3+x & 2+x & 1+x \\ 3+x & 3+x & 3+x & 3+x & 3+x & 3+x & 3+x & 3+x & 2+x & 1+x \\ 2+x & 2+x & 2+x & 2+x & 2+x & 2+x & 2+x & 2+x & 1+x & x \\ 1+x & 1+x & 1+x & 1+x & 1+x & 1+x & 1+x & 1+x & x & -1+x \end{bmatrix}$$

proposed in [7] and M and N as identity matrices of the appropriate dimensions. The following result is generated applying the function *WPartit* from [3]:
`WPartit[A,IdentityMatrix[10],IdentityMatrix[10]]]`:

$$A^{\dagger}_{MN}(M=I_m,N=I_n)=A^{\dagger}=\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -\frac{5}{6} & -\frac{1}{3} & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{5}{6} & \frac{7}{9}-\frac{x}{4} & \frac{4}{9} & \frac{1}{9}+\frac{x}{4} \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & \frac{4}{9} & \frac{1}{9} & -\frac{2}{9} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} & \frac{1}{9}+\frac{x}{4} & -\frac{2}{9} & -\frac{5}{9}-\frac{x}{4} \end{bmatrix}.$$

Example 3.3. The CPU time needed for the computation of the weighted Moore-Penrose inverse (according to the Algorithm 2.1 and Algorithm 2.2) implemented in PHP and MATHEMATICA is compared in the next table. Testing was done on the local machine and from the client in a wireless network. We have an access to the web server using the infrastructure mode wireless networking with the access point. Local machine has the following performance:

- Windows edition: *Windows Vista*$^{(TM)}$ *Ultimate*;

- Processor: *Intel*(*R*) *Pentium*(*R*) *Dual CPU T*3200 @ 2.00*GHz*; Memory (*RAM*) : 2940*MB*;

- Software: MATHEMATICA 6, Free Software: *WAMP* 5 1.7.4: PHP 5.2.5, *Apache* 2.2.6 *Server*, *MySQL* 5.0.45 − *community* − *nt* and *phpMyAdmin* 2.11.2.1

Also, in the table are arranged results obtained on the set of randomly generated test and matrices $A_{m\times n}$ from [6], and randomly generated symmetric positive definite matrices $M_{m\times m}$ and $N_{n\times n}$.

$m\times n$	**PHP**	**MATHEMATICA**
15×15	0.166 sec.	0.234 sec.
20×20	0.277 sec.	0.889 sec.
30×30	0.880 sec.	5.444 sec.
45×45	4.665 sec.	35.771 sec.
45×70	19.920 sec.	439.127 sec.
50×50	6.203 sec.	62.822 sec.
60×60	12.627 sec.	179.058 sec.
80×80	40.750 sec.	621.477 sec.

Table 1. PHP and MATHEMATICA execution time.

According to results from Table 1, PHP implementation gives the better results compared to MATHEMATICA implementation. This is especially evident for large matrices. The speed performance of the matrix operations such as: addition, subtraction, transpose, multiplication, multiplication with scalar, generating *i*-th matrix column and the sub-matrix is negligible when we calculate the weighted MP inverse .

$m\times n$	expression	PHP	MATHEMATICA
60×60	A^{-1}	0.416 sec.	0.515 sec.
70×70	A^{-1}	0.690 sec.	0.969 sec.
80×80	A^{-1}	1.081 sec.	1.825 sec.
80×80	$X_1 = (a_1^* M a_1)^{-1} a_1^* M$	0.007 sec.	0.234 sec.
80×80	$X_1 = (A^* M A)^{-1} A^* M$	2.285 sec.	25.709 sec.

Table 2. Comparison of the execution times for the calculation of the expressions.

For example, penultimate expression in the table, which configures in Algorithm 2.1, shows that PHP time execution gives about 33 times better results than MATHEMATICA. For large matrix, computation of weighted MP inverse, demands carrying out large number of iterations, and there are great differences in time calculation. The purpose of these mentioned times is to demonstrate that in certain circumstances you can always dramatically improve on the CPU time required for lengthy computations by using PHP code instead of advanced *QPEs* (*QuantitativeProgrammingEnvironments*) such as MATHEMATICA.

4. Conclusions

In this paper are presented results, which indicate the usefulness of both, PHP and MATHEMATICA implementations of the algorithms for computations the weighted Moore-Penrose inverse, from [6]. In the instant case, for the computing of the weighted Moore-Penrose inverse for constant matrices, we can use PHP because it gives better results. In this way, the user bypasses the use of the expensive and robust application. Contrary, for symbolic calculation of the rational and polynomial matrices should be used MATHEMATICA. We have not yet created in PHP environment a better function then MATHEMATICA function $Simplify[\ ]$. The future research will be focused on the PHP implementation of the algorithms used for the symbolic computation of rational and polynomial matrices, as well as the simplification of the rational expressions. Also, future work will be including database storage system interconnected with MySQL.

REFERENCES

1. A. Ben-Israel and T. N. E. Greville: *Generalized inverses: theory and applications*. Second Ed., Springer, 2003.
2. M. D. Petković, P. S. Stanimirović and M. B. Tasić: *Effective partitioning method for computing weighted MoorePenrose inverse*. Comput. Math. Appl. **55** (2008), 1720-1734.
3. M. B. Tasić, P. S. Stanimirović and M. D. Petković: *Symbolic computation of weighted Moore-Penrose inverse using partitioning method*, Appl. Math. Comput. **189** (2007), 615–640.
4. M. B. Tasić and P. S. Stanimirović: *Symbolic and recursive computation of different types of generalized inverses*, Appl. Math. Comput. **199** (2008), 349–367.
5. M. B. Tasić, P. S. Stanimirović and S. H. Pepić: *About the generalized LM inverses and the Weighted Moore Penrose inverse*, Appl. Math. Comp. **216** (2010), 114–124.
6. G. R. Wang and Y. L.Chen: *A recursive algorithm for computing the weighted Moore-Penrose inverse $A^{\dagger}_{MN}$*, J. Comput. Math. **4** (1986), 74–85.
7. G. Zielke, *Report on test matrices for generalized inverses*, Computing **36** (1986), 105–162.

Selver H. Pepić

University of Niš, Faculty of Sciences and Mathematics,

P. O. Box 224, Višegradska 33, 18000 Niš, Serbia

p_selver@yahoo.com