Computer Science • 17 (3) 2016 http://dx.doi.org/10.7494/csci.2016.17.3.297

Maciej Woźniak
Maciej Paszyński

APPLICATION OF PROJECTION-BASED INTERPOLATION ALGORITHM FOR NON-STATIONARY PROBLEM

Abstract *In this paper, we present a solver for non-stationary problems using L^2 projection and h-adaptations. The solver utilizes the Euler time integration scheme for time evolution mixed with projection-based interpolation techniques for solving the L^2 projection problem at every time step. The solver is tested on the model problem of a heat transfer in an L-shape domain. We show that our solver delivers linear computational cost at every time step.*

Keywords L-Shape, h-adaptivity, parallel, L2 projection, non-stationary

Citation Computer Science 17 (3) 2016: 297–319

1. Introduction

In this paper, we present how the projection-based interpolation algorithm (originally introduced by Leszek Demkowicz [2]) can be used for efficiently solving non-stationary problems. This is possible when the non-stationary problem is solved with the Euler scheme and reduced to a sequence of L^2 projections. Classical algorithms dealing with solving this problem (such as multi-frontal solvers [4, 5]) have $O(N^{1.5})$ computational cost for each time step, in the case of a regular grid. The possible parallel implementation can reduce the cost to $O(N)$ [1]. Additionally, for grids with point or edge singularities, it is possible to reduce the computational cost to $O(N)$ using a sequential multi-frontal solver [6, 9, 14]. The parallelization of these techniques for grids with point or edge singularities allow us to reduce the cost to $O(logN)$. In this paper, we present an alternative projection-based interpolation solver that reduces the computational cost to $O(N)$ in the case of sequential execution and even further to $O(\frac{N}{c})$ when we use c cores. The projection-based interpolation technique has been successfully used for generating continuous approximations of two- or three-dimensional bitmaps, with applications to material science [10, 11, 12, 15] or modeling of the human head [8]. In this paper, we focus on utilizing the Projection-Based Interpolation (PBI) algorithm for solving a sequence of projection problems arising from the Euler scheme used for time discretization of non-stationary problems.

2. Issues to be addressed in this work

In this chapter, we are going to indicate the algorithmic challenges as well as the functional and non-functional issues related to the problem of constructing an efficient sequential and parallel projection-based interpolation method solver.

2.1. Challenges

There are two key challenges in the algorithm. The main one is to enable explicit L^2 projections from the previous time step. The second challenge is to design a new error estimation and the element dividing rules.
Most implementations are parallelized using MPI or PVM. Both solutions are very effective, but they cannot run on GPU. In terms of rationalization, the major change would be the use of CUDA. This will bring the ability to run on massively parallel machines with shared memory as new generations of graphic cards. Last but not least, an important challenge is to achieve straightforward parallel implementation, based on the loop parallelization.

2.2. Architectural design principles

In this section, we compare the architectural design principles to those presented in state-of-the-art adaptive codes [10] and [7]. All changes in 2.1 imply the following architectural design principles. For efficient parallelization on a GPU architecture, we give up the refinement trees and utilize flat data structures more suitable for multi-core processing.

3. Functional requirements

1. Range of equations
   (a) Solves Partial Differential Equations in $2D$ with the Finite Element Analysis
      i. Supports h-adaptive refinements
      ii. Allows Dirichlet, Cauchy, and Neumann Boundary Conditions
      iii. Uses hierarchical basis functions
      iv. Uses Euler time discretization method
2. Provides visualization
   (a) Plots current solution in each time step
   (b) Plots current mesh in each time step
   (c) Prints to standard output current error in time step/adaptation step

4. Non-functional requirements

1. Ability to be understood by an average programmer
   (a) Easy algorithm
   (b) Simple data structures
2. Ability to perform on massive parallel machines
   (a) Easy scalability in shared memory model
   (b) Ability to run on GPU

5. Basis functions used for the solution of the L2 projection problem

The projection problem is solved over a finite element mesh. To the mesh that we used in our simulation is presented in Figure 5a, as well as the master element is presented in Figure 5a, while an exemplary element is presented in Figure 1. At the element vertices, edges, and interiors, we define the so-called basis functions used for the approximation of the projection problem solution.

5.1. 1D case

Over the $1D$ element, we define the following set of two nodal and one edge functions. See Figures 2a–2c.

$$\chi_1(\epsilon) = 1 - \epsilon \tag{1}$$

$$\chi_2(\epsilon) = \epsilon \tag{2}$$

$$\chi_3(\epsilon) = (1 - \epsilon) \tag{3}$$

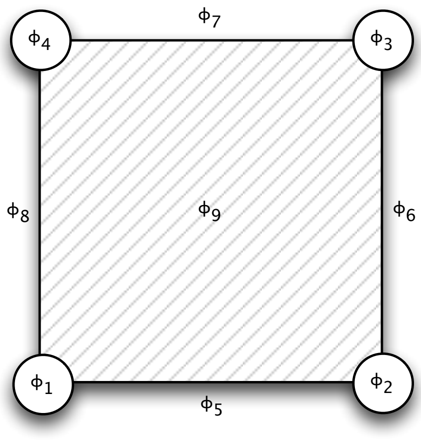


Figure 1. Element functions.

5.2. 2D case

Over the $2D$ element we define the following set of four vertex, four edge, and one internal bubble function.

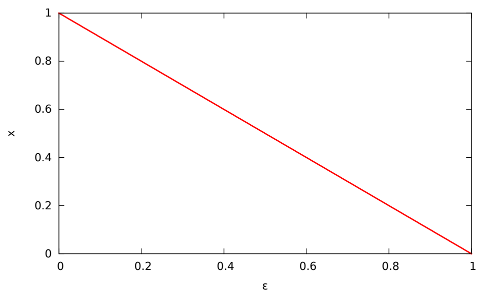
$$\hat{\phi}_1(x_1,x_2) = \chi_1(x_1)\chi_1(x_2) = (1-x_1)(1-x_2) \tag{4}$$
$$\hat{\phi}_2(x_1,x_2) = \chi_2(x_1)\chi_1(x_2) = x_1(1-x_2) \tag{5}$$
$$\hat{\phi}_3(x_1,x_2) = \chi_2(x_1)\chi_2(x_2) = x_1x_2 \tag{6}$$
$$\hat{\phi}_4(x_1,x_2) = \chi_1(x_1)\chi_2(x_2) = (1-x_1)x_2 \tag{7}$$
$$\hat{\phi}_5(x_1,x_2) = \chi_3(x_1)\chi_1(x_2) = (1-x_1)x_1(1-x_2) \tag{8}$$
$$\hat{\phi}_6(x_1,x_2) = \chi_2(x_1)\chi_3(x_2) = x_1(1-x_2)x_2 \tag{9}$$
$$\hat{\phi}_7(x_1,x_2) = \chi_3(x_1)\chi_2(x_2) = (1-x_1)x_1x_2 \tag{10}$$
$$\hat{\phi}_8(x_1,x_2) = \chi_1(x_1)\chi_3(x_2) = (1-x_1)(1-x_2)x_2 \tag{11}$$
$$\hat{\phi}_9(x_1,x_2) = \chi_3(x_1)\chi_3(x_2) = (1-x_1)x_1(1-x_2)x_2 \tag{12}$$

Element basis functions are presented in Figures 3a–3i.

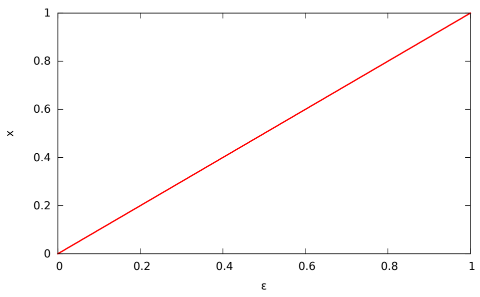
6. L2 projection

6.1. Definition

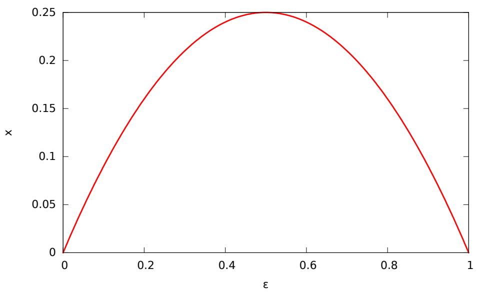
Here, we define the L^2 projection that will be used hereafter.



(a) Function χ_1



(b) Function χ_2



(c) Function χ_3

Figure 2. 1D basis functions.

The L^2 projection $u = P_h B \in V_h$ of a function of two variables $B \in L^2(\Omega)$ is defined by [13]:

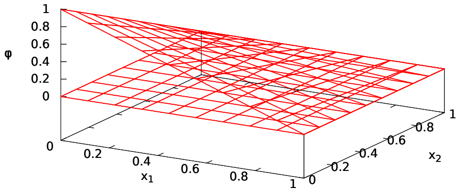
$$\forall v \in V_h : \int\limits_{\Omega} (B - u) v \, dx = 0 \tag{13}$$

We seek for approximation $u \approx \sum_i a_i e_i$ where e_i are basis functions over the computational mesh, such as the u approximates a given function B.

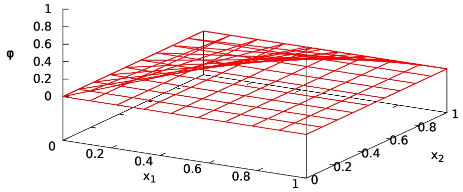
6.2. Solution of the L2 projection problem by projection-based interpolation

The projection problem can be solved by using the so-called projection-based interpolation technique originally introduced by Leszek Demkowicz in the context of error estimations for hp adaptive finite element method [3]. The method solves the projection problem locally over each element, starting from vertices:

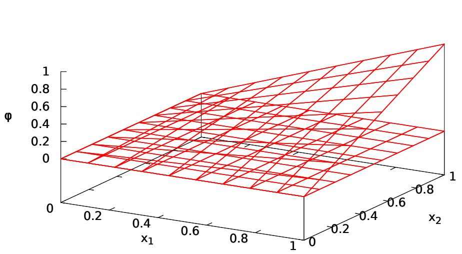
$$u_{vert} = \sum_{i=1}^{4} a_i \hat{\phi}_i \tag{14}$$



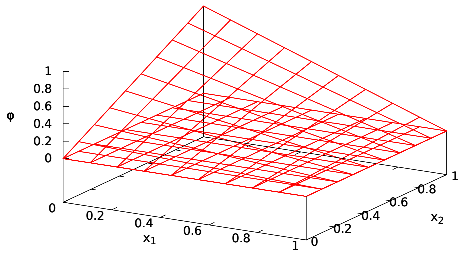
(a) Function ϕ_1



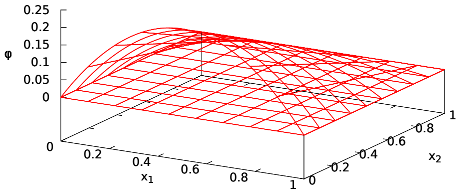
(b) Function ϕ_2



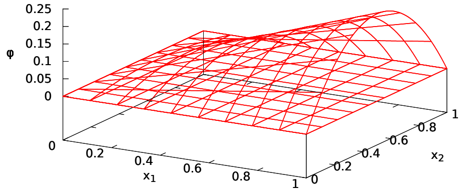
(c) Function ϕ_3



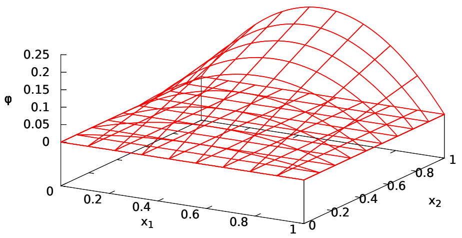
(d) Function ϕ_4



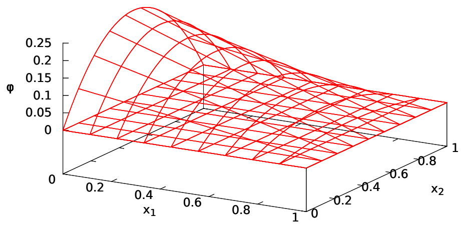
(e) Function ϕ_5



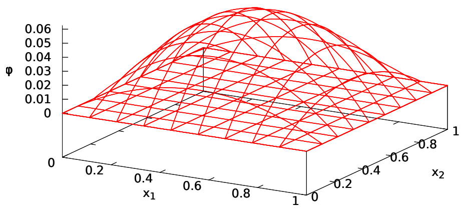
(f) Function ϕ_6



(g) Function ϕ_7



(h) Function ϕ_8



(i) Function ϕ_9

Figure 3. 2D basis functions.

through edges:

$$u_{edge} = \sum_{i=5}^{8} b_i \hat{\phi}_i \tag{15}$$

and finishing with interior:

$$u_{int} = c_9 \hat{\phi}_9 \tag{16}$$

finally:

$$u = u_{vert} + u_{edge} + u_{int} \tag{17}$$

Let us assume that $u_{t+1} = B(u_t)$, where u_t and u_{t+1} are solutions at time steps t and $t+1$. Let p be an arbitrary point in Ω, then $\forall p \in \Omega : u_{t+1}(p) = B(u_t(p))$.

6.3. Vertices

Let us start with a simple fact that, for approximation at a given point, we can simply use the value of the function at the point:

$$a_i = B(vertex_i) \tag{18}$$

6.4. Edges

We minimize the L^2 norm; namely, the difference between function B and it's approximation over edge:

$$\|(B - u_{vert}) - b_i \hat{\phi}_i\|_{L^2} \to 0 \tag{19}$$

By rewriting the norm, we obtain:

$$\forall v \in V_h \int\limits_{edge} [(B - u_{vert}) - b_i \hat{\phi}_i] v \, dx = 0 \tag{20}$$

in our case, $v = \hat{\phi}_i$ and E_i is edge:

$$\int\limits_{E_i} [(B - u_{vert}) - b_i \hat{\phi}_i] \hat{\phi}_i \, dx = 0 \tag{21}$$

$$\int\limits_{E_i} [(B - \sum_{j=1}^{4} a_j \hat{\phi}_j) - b_i \hat{\phi}_i] \hat{\phi}_i \, dx = 0 \tag{22}$$

$$b_i \int\limits_{E_i} \hat{\phi}_i^{\,2} \, dx = \int\limits_{E_i} \left(B - \sum_{j=1}^{4} a_j \hat{\phi}_j \right) \hat{\phi}_i \, dx \tag{23}$$

$$b_i = \frac{\int\limits_{E_i} \left(B - \sum\limits_{j=1}^{4} a_j \hat{\phi_j} \right) \hat{\phi_i} \, dx}{\int\limits_{E_i} \hat{\phi_i^2} \, dx} \tag{24}$$

All integrals can be simply calculated by Gaussian Quadratures. Similar considerations can be made for all of the remaining edges.

6.5. Internal bubble node

We minimize norm L^2; namely, the difference between function B and it's approximation over an interior:

$$\|(B - u_{vert} - u_{edges}) - c_9 \hat{\phi_9}\|_{L^2} \to 0 \tag{25}$$

In our case, $v = \hat{\phi_9}$ and F_i is element

$$\int\limits_{F_i} [(B - u_{vert} - u_{edges}) - c_9 \hat{\phi_9}] \hat{\phi_9} \, dx^2 = 0 \tag{26}$$

$$\int\limits_{F_i} [(B - \sum_{i=1}^{4} a_i \hat{\phi_i} - \sum_{j=5}^{8} b_j \hat{\phi_j}) - c_9 \hat{\phi_9}] \hat{\phi_9} \, dx^2 = 0 \tag{27}$$

$$c_9 \int\limits_{F_i} \hat{\phi_9}^2 \, dx^2 = \int\limits_{F_i} \left(B - \sum_{i=1}^{4} a_i \hat{\phi_i} - \sum_{j=5}^{8} b_j \hat{\phi_j} \right) \hat{\phi_9} \, dx^2 \tag{28}$$

$$c_9 = \frac{\int\limits_{F_i} \left(B - \sum\limits_{i=1}^{4} a_i \hat{\phi_i} - \sum\limits_{j=5}^{8} b_j \hat{\phi_j} \right) \hat{\phi_9} \, dx^2}{\int\limits_{F_i} \hat{\phi_9^2} \, dx^2} \tag{29}$$

All integrals can be simply calculated by Gaussian quadratures.

6.6. Error estimation

In the classical FEM, we estimate error by calculating the solutions for fine mesh and coarse mesh. In that case, error estimation is:

$$\text{error} = \|B(u_{fine}) - B(u_{coarse})\|_{L^2} \tag{30}$$

$$\text{error} = \int\limits_{F} [B(u_{fine}) - B(u_{coarse})] \, dx \tag{31}$$

In the projection method, we can estimate error without calculating fine mesh:

$$\text{error} = \|B(u_t) - u_{t+1}\|_{L^2} \tag{32}$$

$$\text{error} = \int_F [B(u_t) - u_{t+1}]\, dx \tag{33}$$

where $B(u_t)$ denotes applying B in each point and u_{t+1} is FE-interpolant with FEM. All integrals can be calculated by Gaussian quadratures, which simplifies calculating $B(u_t)$ to points of quadrature.

7. H-adaptations

There are different types of adaptations to mention; the most important ones being h and p adaptations. In the algorithm, we take a closer look only at h-adaptations. There are two basic rules for h-adaptations one has to follow [7.1–7.2].

7.1. Single irregularity

The element can be divided into smaller elements only if its neighbors are not greater in size than the element itself. This means that, if element has a bigger neighbor, we first have to divide the neighbor. For example, the upper neighbor has two bottom neighbors. This means that it is bigger and has to be divided first. See Figure 4.

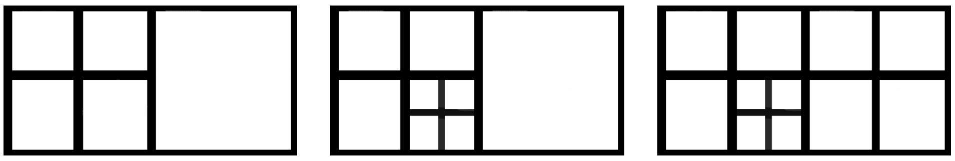


Figure 4. Left Panel: Initial elements. Middle panel: Attempt of breaking element. Right panel: Single irregularity rule applied.

7.2. Breaking of edges

In classical h-adaptation, the edge can be divided only if the neighbor sharing the edge is divided. As long as we use a different algorithm than the traditional one, we can change this rule. The edge can be divided even if its neighbor sharing the edge is not divided. Then, we have to enforce the values on the broken edge to make it act as if it is not divided until we divide the neighbor. The same rule applies to vertices on broken edges.

8. Data structures

In this section, we will show the data structures we used in the algorithm. See record definitions 1, 2, and 3. We will refer to them in Sections 9 and 10.

vertex	
integer	x
integer	y
float	a

Record definition 1
Vertex data structure

edge	
vertex	begin
vertex	end
float	b

Record definition 2
Edge data structure

element	
element	upperNeighbor[2]
element	bottomNeighbor[2]
element	leftNeighbor[2]
element	rightNeighbor[2]
integer	upperNeighborsCalculate
integer	bottomNeighborsCalculate
integer	leftNeighborsCalculate
integer	rightNeighborsCalculate
vertex	upperLefVertex
vertex	upperRightVertex
vertex	bottomLeftVertex
vertex	bottomRightVertex
edge	upperEdge
edge	bottomEdge
edge	leftEdge
edge	rightEdge
double	c
double	error

Record definition 3
Element data structure

8.1. Time step

Each time step is represented by a bitmap of floats. Each initial element has a resolution of several pixels to enable adaptation. Furthermore, after tests, it turned out that the resolution of an element below 10 pixels doesn't make sense for interpolation and error estimation.

9. Algorithm

In this section, we will describe in pseudo-code basis of algorithm. The serial implementation of the computations performed in a single time step can be expressed by the following pseudo-code, using the loop parallelization paradigm Listing 1.

Listing 1. Serial version.

```
BEGIN
$work = true$
while $work$
  for $i = 0, vertices\_number - 1:$
    $vertices[i].a = B(vertices[i])$
  end loop
  for $i = 0, edges\_number - 1:$
    $edges[i].b = calculate\_b(edges[i])$
  end loop
  for $i = 0, element\_number - 1:$
    $elements[i].c = calculate\_c(elements[i])$
  end loop
  $current\_max\_error = 0$(* for $i = 0, element\_number - 1:$
    $elements[i].error = compute\_error(elements[i])$
    if $elements[i].error > current\_max\_error$
      $current\_max\_error = elements[i].error$
    end if
  end loop
  if $current\_max\_error > max\_error$
    for $i = 0, element\_number - 1:$
      if $elements[i].error > 0.3 * current\_max\_error$
        break $elements[i]$
      end if
    end loop
  else
    $work = false$
  end if
end loop
for $i = 0, element\_number - 1:$
  $interpolate(elements[i])$
end loop
END
```

Algorithms for calculating coefficients a, b, and c can be easily obtained from 6.

10. Parallel algorithm

In this section, we will describe in pseudo-code basis of algorithm. The parallel implementation of the computations performed in a single time step can be expressed by the following pseudo-code, using the loop parallelization paradigm Listing 2.

Listing 2. Parallel version.

```
BEGIN
$work = true$
while $work$
  parallel for $i = 0, vertices\_number - 1:$
    $vertices[i].a = B(vertices[i])$
  end loop
  parallel for $i = 0, edges\_number - 1:$
    $edges[i].b = calculate\_b(edges[i])$
  end loop
  parallel for $i = 0, element\_number - 1:$
    $elements[i].c = calculate\_c(elements[i])$
  end loop
  $current\_max\_error = 0$
  for $i = 0, element\_number - 1:$
    $elements[i].error = calculate\_error(elements[i])$
    If $elements[i].error > current\_max\_error$
      $current\_max\_error = elements[i].error$
    end if
  end loop
  if $current\_max\_error > max\_error$
    for $i = 0, element\_number - 1:$
      if $elements[i].error > 0.3 * current\_max\_error$
        break $elements[i]$
      end if
    end loop
  else
    $work = false$
  end if
end loop
parallel for $i = 0, element\_number - 1:$
  $interpolate(elements[i])$
end loop
END
```

10.1. Breaking element

See Listing 3.

Listing 3. Element breaking rule.

```
BEGIN
enforce single irregularity rule
create new elements, vertices and edges in the center of old
    ↪ element
for each edge
  if edge was not broken
    break edge
    if edge is not edge of domain
      enforce values at new edges and vertices
    end if
  else
    get broken edges and vertices
  end if
end loop
set neighbors
END
```

11. Problem formulation

Let us take a closer look at the heat transfer in the L-shaped domain. The transfer can be represented with the following equation:

$$\frac{\partial u}{\partial t} - \Delta u = f \tag{34}$$

In our case, the upper left edge of the domain is heated to temperature 1. The bottom right edge is cooled to temperature -1. None of the other edges are cooled nor heated.

$$f = \begin{cases} 1 - u_t & \text{where } x = 0 \\ -1 - u_t & \text{where } y = 0 \\ 0 & \text{where } x \neq 0 \text{ and } y \neq 0 \end{cases} \tag{35}$$

We can transform the equation by using the Euler method with $\frac{\partial u}{\partial t} = \frac{u_{t+1} - u_t}{\delta}$, where $\delta \to 0$ in the following way:

$$\frac{u_{t+1} - u_t}{\delta} - \Delta u = f \tag{36}$$

$$u_{t+1} - u_t - \Delta u \delta = f\delta \tag{37}$$

$$u_{t+1} = u_t + \Delta u \delta + f\delta \tag{38}$$

Finally, we solve the above equation in the following time step as a sequence of projection problems.

12. Results

In this section, we will present the results of testing the algorithm on a benchmark heat transfer in the L-Shape domain. Both for meshes and heat distribution, we used an initial mesh of 1×1 element, resolution 640×640 per element, max error 0.001, and time step 0.5.

The resulting meshes are presented in Figures (5a–5f).

Heat distribution is presented in Figures (6a–6f).

12.1. Error decrease rate

We calculated error rates for meshes of 1 to 1024 elements without adaptation as well as with (see Fig. 7). The total bitmap size is 20480 pixels. In the plot, we treat adaptation levels as a constant mesh (the same size as the smallest elements). Relative error is estimated in the manner shown in Subsection 6.6.

12.2. Calculating time

We calculated error rates for meshes of 1 to 1024 elements without adaptation as well as with. The total bitmap size is 20480 pixels. In the plot, we use the same the initial mesh for both versions of algorithm (with and without adaptations). Since interpolation should take most of the calculating time, we tested the algorithm with different number of elements (each of size 10 pixels). For bitmap of size 10240×10240 and constant mesh total calculating time takes 68.49 seconds while interpolation 34.81 seconds. This means that interpolation can take 50% of the total calculating time: see Figures 8 and 9.

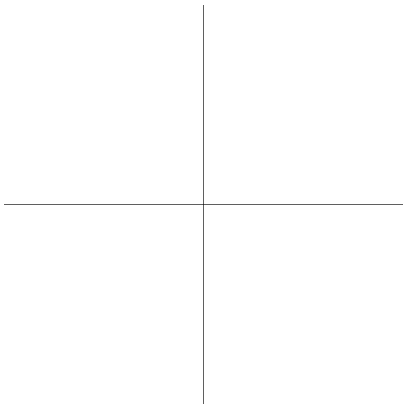
13. Computational cost estimates

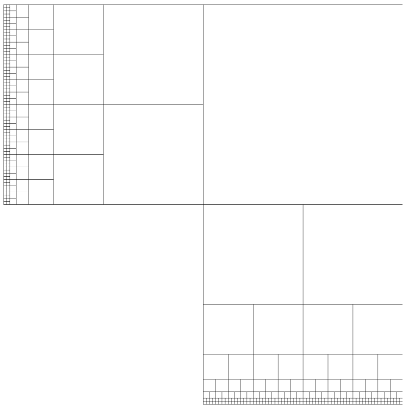
13.1. Solve

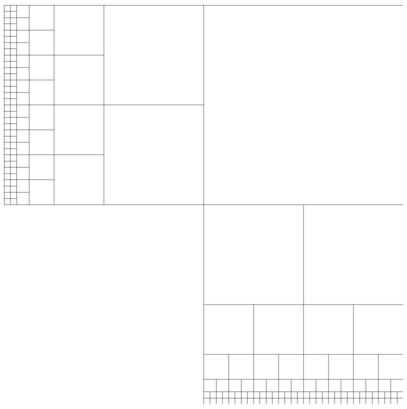
Each element is approximated by a set of polynomials: 4 polynomials order 1 on vertices, 4 polynomials order 2 on edges, and 1 polynomial order 2 in the center on element. Solving the element includes calculating the weights of polynomials over element. Calculating vertex weights is constant and takes Φ^{vertex}. Calculating weights of the edges requires the use of a Gauss quadrature with 2 points and takes $2\Phi^{edge}$, while calculating the weight of the center requires the use of a Gauss quadrature with 4 points and takes $4\Phi^{center}$. Take a closer look at Φ^{vertex}, Φ^{edge}, and Φ^{center}. We can see that these are constants. Calculating weights over element takes

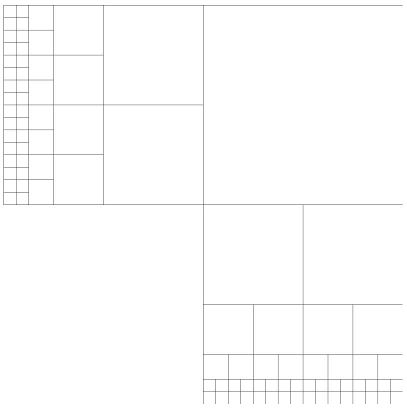
$$t_{solve} = \Phi^{vertex} + 2\Phi^{edge} + 4\Phi^{center} \tag{39}$$

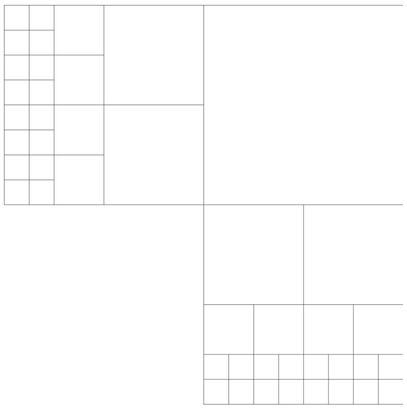
This means each element is being solved in constant time and complexity $O(1)$.


(a) Mesh in time step 0


(b) Mesh in time step 10


(c) Mesh in time step 40


(d) Mesh in time step 130


(e) Mesh in time step 400

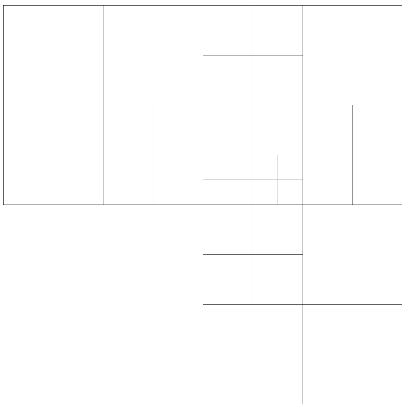
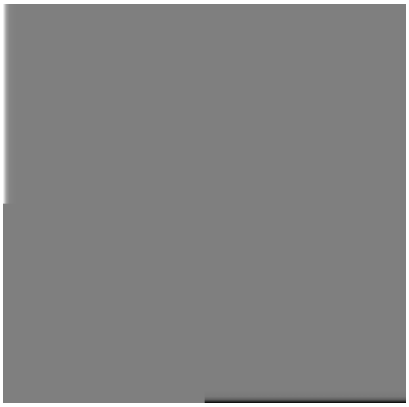
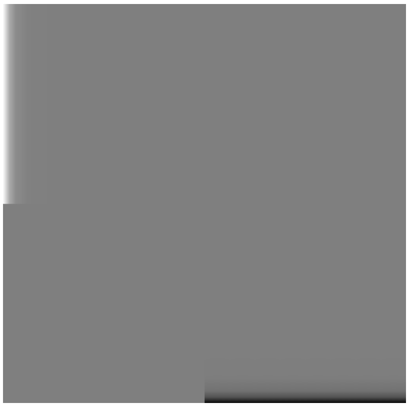
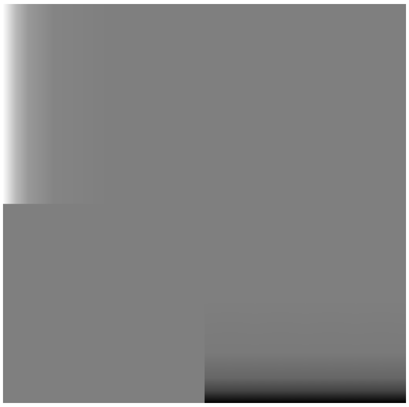

(f) Mesh in time step 5860

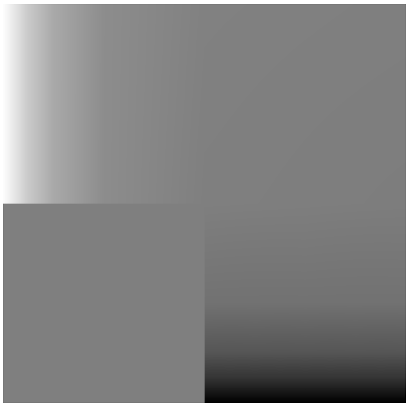
Figure 5. Mesh in different time steps.

(a) Heat distribution in time step 0


(b) Heat distribution in time step 10


(c) Heat distribution in time step 40


(d) Heat distribution in time step 130


(e) Heat distribution in time step 400

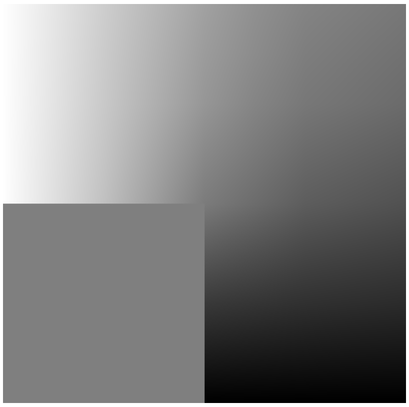

(f) Heat distribution in time step 5860

Figure 6. Heat distribution in different time steps.

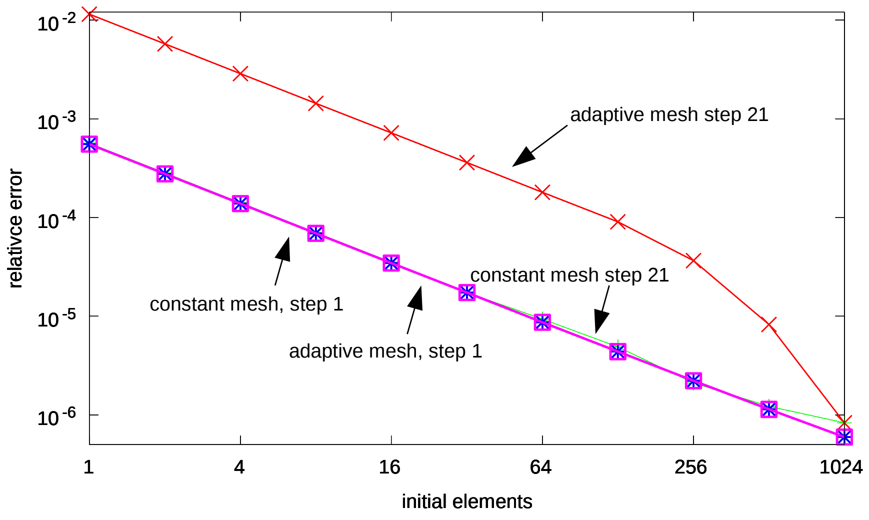


Figure 7. Error decrease rate.

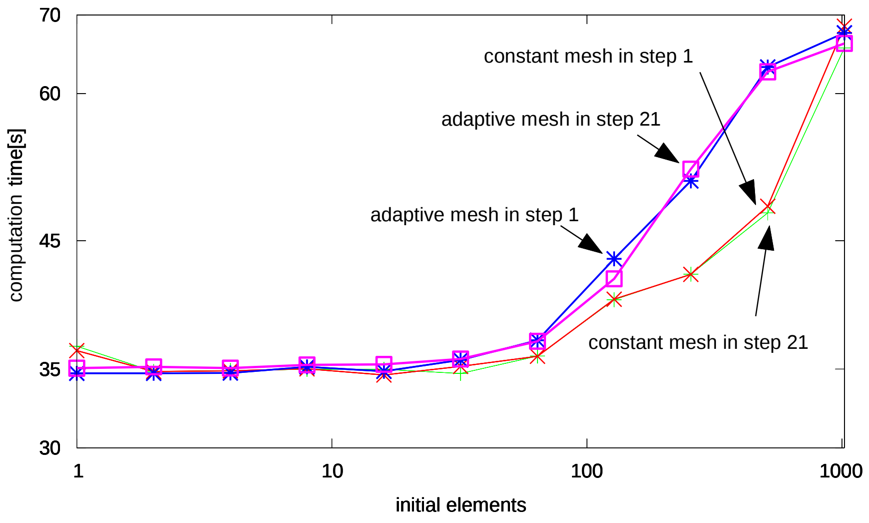


Figure 8. Calculating time.

13.2. Interpolation

Interpolating an output bitmap for each element includes calculating all 9 polynomials multiplied by weights. Let us assume that bitmap over element has resolution of $Res_w \times Res_h$, which gives $Res_w \times Res_h$ points. Calculating the value of each vertex polynomial in point costs ϕ^{vertex}. It is similar to other polynomials with costs of ϕ^{edge} and ϕ^{center}. This means that approximating the value of one point in an element costs $4\phi^{vertex} + 4\phi^{edge} + \phi^{center}$. Interpolating the whole element then costs:

$$t_{interpolate} = \left(4\phi^{vertex} + 4\phi^{edge} + \phi^{center}\right) Res_w Res_h \tag{40}$$

Complexity of interpolating element is $O(Res_w Res_h)$

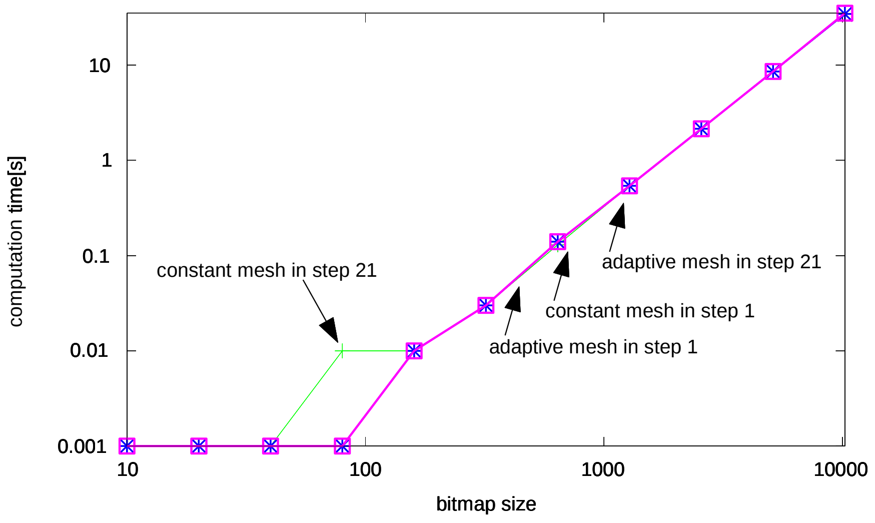


Figure 9. Interpolation time.

13.3. Error estimation

In this problem, we make use of several error estimators.

The first one requires the use of a Gauss quadrature with 4 points. Each point requires interpolating by element polynomials with previously calculated weights and calculate exact values for points as in. The cost of calculating polynomials is the same as proven in 13.2: $4\phi^{vertex} + 4\phi^{edge} + \phi^{center}$ in each point. Calculating exact values is the same as calculating weights for vertices (same operations): Φ^{vertex}. Furthermore, calculating values for a Gauss quadrature requires more operations with constant cost ϕ^{gauss}; therefore, the total cost of error estimation is $t_{error} = 4(4\phi^{vertex} + 4\phi^{edge} + \phi^{center} + \Phi^{vertex} + \phi^{gauss})$. The complexity of error estimation over element is $O(1)$. If $error_{elelemt}$ exceeds $error_{max}$, the element is broken.

A second estimator checks errors in Gaussian quadrature points. If the error exceeds $0.25 error_{max}$ at a given point, the element is broken.

The last estimator estimates errors in points of $0.1 size_{elelemt}$ and $0.9 size_{element}$. If the error exceeds $0.25 error_{max}$ at a given point, the element is broken.

13.4. H-adaptations

In each step of the adaptation, we divide 30% of elements with the highest error rate until reaching a given error level. In practice, it makes no sense to make more than five adaptations in one time step. Adaptation over one element takes θ, where θ is constant. Let us assume that we execute all of the five possible steps of adaptation, each time on 30% of elements. At the very beginning, we have $N_x \times N_y$ elements, which gives $0.3N_xN_y$ adaptations. Adapting one element gives 4 elements in the end. After the first step, we will have $N_xN_y - 0.3N_xN_y + 0.3 * 4N_xN_y = 1.9N_xN_y$ elements. Adaptations in first step will take $0.3N_xN_y\theta$. As one can see, the total cost of adaptations is

$$t_{adaptations} = 7.92033N_xN_y.\theta \tag{41}$$

This is the only part that is too hard, too expensive, or even impossible to run in parallel. That is why we will stay here with serial cost and complexity, which is $O(N_xN_y)$.

14. Serial implementation

14.1. Solving static mesh

Assume that we have a mesh of $N_x \times N_y$ elements of $Res_x \times Res_y$ resolution. The cost of serial solving, interpolation, and error estimation will be:

$$t_{serial} = N_xN_y(t_{solve} + t_{interpolate} + t_{error}) \tag{42}$$

and the complexity is

$$O\left(N_xN_yRes_wRes_h\right) \tag{43}$$

14.2. Solving adaptive mesh

Assume that we add h-adaptivity to step 14.1 just like in 13.4. Then we will have $264011N_xN_y$ elements to solve all together. Interpolation in each adaptation step will still strictly depend on the output bitmap resolution and will cost:

$$t_{allinterpolation} = N_xN_yRes_wRes_h(4\phi^{vertex} + 4\phi^{edge} + \phi^{center})$$

in each adaptation step.

All together, solving the adaptive mesh will cost:

$$t_{adaptationsolve} = 26.4011N_xN_y(t_{solve} + t_{error}) \tag{44}$$

with a total cost of:

$$\begin{aligned} t_{adaptation} = {} & N_xN_y(26.4011(t_{solve} + t_{error}) + \\ & + Res_wRes_h(4\phi^{vertex} + 4\phi^{edge} + \phi^{center})) \end{aligned} \tag{45}$$

and complexity $O\left(N_xN_yRes_wRes_h\right)$

15. Parallel implementation

15.1. Solving static mesh

Assume that we have a mesh of $N_x \times N_y$ elements of $Res_x \times Res_y$ resolution and U CPUs with zero time communication. Each element can be solved and interpolated independently. The cost of parallel solving, interpolation, and error estimation will be:

$$\frac{t_{parallel} = N_x N_y (t_{solve} + t_{interpolate} + t_{error})}{U} \tag{46}$$

while complexity is $O\left(\frac{N_x N_y Res_w Res_h}{U}\right)$.

If U is grater or equal to $N_x N_y$, then the mesh can be solved in

$$t_{solve} + t_{interpolate} + t_{error} = Res_x Res_y \left(4\phi^{vertex} + 4\phi^{edge} + \phi^{center}\right) \tag{47}$$

(we assume that error estimation and solve times are too minor to include them in the equation).

15.2. Solving adaptive mesh

Assume that we add h-adaptivity to step 15.1 just like in 13.4. Then, we will have $264011 N_x N_y$ elements to solve all together. Interpolation in each adaptation step will still strictly depend on output bitmap resolution and will cost:

$$t_{allinterpolation} = \frac{N_x N_y Res_w Res_h (4\phi^{vertex} + 4\phi^{edge} + \phi^{center})}{U} \tag{48}$$

in each adaptation step. All together, solving the adaptive mesh will cost:

$$t_{adaptationsolve} = \frac{26.4011 N_x N_y (t_{solve} + t_{error})}{U} \tag{49}$$

with total cost of:

$$\begin{aligned} t_{adaptation} = & \frac{N_x N_y (26.4011(t_{solve} + t_{error})}{U} + \\ & + \frac{Res_w Res_h (4\phi^{vertex} + 4\phi^{edge} + \phi^{center})}{U} \end{aligned} \tag{50}$$

and complexity

$$O\left(\frac{N_x N_y Res_w Res_h}{U}\right) \tag{51}$$

15.3. Communication

Since solving on a GPU requires sending and receiving data from a device and usually involves a bottleneck, we have to include this in the total cost and complexity. If the algorithm is optimized, it requires one send and receive of the output bitmap, one send of the input bitmap per time step, and one send and receive per adaptation. Let us assume that each time step consists of 5 adaptation steps (like in 13.4). Let α^{double} be the time of sending one double to or from the GPU. Then, sending the bitmap will take $t_{bitmap} = (N_x Res_w + 1)(N_y Res_h + 1)\alpha^{double}$. Let $\alpha^{element}$ be the time of sending one element to or from the GPU. Then, sending the set of $N_x N_y$ elements will take $t_{elements} = N_x N_y \alpha^{element}$. The total communication cost will be:

$$\begin{aligned} t_{communication} = {} & 2(N_x Res_w + 1)(N_y Res_h + 1)\alpha^{double} + \\ & + 52.8022 N_x N_y \alpha^{element} \end{aligned} \tag{52}$$

With complexity $O(N_x N_y Res_w Res_h)$.

15.4. Total parallel cost and complexity

The total cost will be the sum of communication, solving, and interpolation on the adaptive mesh. The total cost will be:

$$\begin{aligned} t_{total} = {} & \frac{N_x N_y (26.4011(t_{solve} + t_{error})}{U} + \\ & + \frac{Res_w Res_h (4\phi^{vertex} + 4\phi^{edge} + \phi^{center}))}{U} + \\ & + 2(N_x Res_w + 1)(N_y Res_h + 1)\alpha^{double} + 52.8022 N_x N_y \alpha^{element} \end{aligned} \tag{53}$$

and total complexity:

$$O\left(\frac{N_x N_y Res_w Res_h}{U} + N_x N_y Res_w Res_h\right) \tag{54}$$

As we were able to test the application only on a GPU with 16 cores (which is much smaller than a reasonable amount for testing speed), we will rely only on theoretical values.

16. Conclusions and future work

With respect to the results presented in section 12 and theoretical complexity in section 15, we tend to think that code and algorithm is not mature enough to be applicable in industrial solutions. It needs the development of efficient parallel element splitting. However, both parallel and serial implementations are extremely efficient and ready for $1D$, $2D$, and $3D$ application. As we were able to observe, interpolation of the bitmap can take half (about 50%) of the computation time. We can assume that the goal of this work was reached and the algorithm has logarithmic complexity according to the element number and ideal implementation. Since this work is only a Proof Of Concept, multiple ways of improvement and research exist for $2D$ problems. There are three basic categories of the possible future work. Since this is a new approach to h-adaptive FEM, different error estimation methods should be tested; for example, different norms as well as norms used to estimate basis function weights. Furthermore, classical error estimation for hp-adaptive FEM and other approaches should be tested. We didnt manage to define an efficient parallel method of splitting elements. This is part of the algorithm that should be further developed. To improve the algorithm and make it more efficient, p-adaptations should be tested. According to Prof. Demkowicz, adding p-adaptations to h-adaptations increases efficiency and improves estimation. It is not obvious how to add p-adaptations to this algorithm, since it is different from a classical FEM.

Acknowledgements

The research presented in this paper was supported by Polish National Science Center grant no. DEC-2012/07/B/ST6/01229

References

[1] Calo V.M., Collier N.O., Pardo D., Paszyński M.: Computational complexity and memory usage for multi-frontal direct solvers used in p finite element analysis. *Procedia Computer Science*, vol. 4, pp. 1854–186, 2011.

[2] Demkowicz L.: *Computing with hp Adaptive Finite Element Method Part I. One and Two Dimensional Problems.* CRC Press, Taylor & Francis, 2006.

[3] Demkowicz L.: Polynomial Exact Sequences and Projection-Based Interpolation with Application to Maxwell Equations. *Lecture Notes in Mathematics*, vol. 1939, pp. 101–158, 2008.

[4] Duff I.S., Reid J.K.: The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, vol. 9, pp. 302–325, 1983.

[5] Duff I.S., Reid J.K.: The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, vol. 5, pp. 633–641, 1984.

[6] Goik D., Jopek K., Paszyński M., Lenharth A., Nguyen D., Pingali K.: Graph Grammar based Multi-thread Multi-frontal Direct Solver with Galois Scheduler. *Procedia Computer Science*, vol. 29, pp. 960–969, 2014.

[7] Goik D., Sieniek M., Paszyński M., Madej L.: Employing an Adaptive Projection-based Interpolation to Prepare Discontinuous 3D Material Data for Finite Element Analysis. *Procedia Computer Science*, vol. 18, pp. 1535–1544, 2013.

[8] Goik D., Sieniek M., Woźniak M., Paszyńska A., Paszyński M.: Hypergraph Grammar based Adaptive Linear Computational Cost Projection Solvers for Two and Three Dimensional Modeling of Brain. *Procedia Computer Science*, *International Conference on Computational Science, Cairns, Australia, 10–12.06.2014*, vol. 29, pp. 1002–1013, Elsevier, 2014.

[9] Gurgul P.: A linear complexity direct solver for h-adaptive grids with point singularities. *Procedia Computer Science*, vol. 29, pp. 1090–1099, 2014.

[10] Gurgul P., Sieniek M., Magiera K., Skotniczny M.: Application of multi-agent paradigm to hp-adaptive projection-based interpolation operator. *Journal of Computational Science*, vol. 4(3), pp. 164–169, 2011.

[11] Gurgul P., Sieniek M., Paszyński M., Madej L.: Three-dimensional adaptive algorithm for continuous approximations of material data using space projection. *Computer Methods in Materials Science*, vol. 13(2), pp. 245–250, 2013.

[12] Gurgul P., Sieniek M., Paszyński M., Madej L., Collier N.: Two dimensional hp-adaptive algorithm for continuous approximations of material data using space projections. *Computer Science, AGH University of Science and Technology Press*, vol. 14(1), pp. 97–112, 2013.

[13] Larson M.G., Bengzon F.: *The Finite Element Method: Theory, Implementation, and Practice.* Springer, 2010.

[14] Paszyńska A., Paszyński M., Jopek K., Woźniak M., Goik D., Gurgul P., AbouEisha H., Moshkov M., Calo V.M., Lenerth A., Nguyen D., Pingali K.: Quasi-Optimal Elimination Trees for 2D Grids with singularities. *Scientiffic Programming*, vol. 2015(Article ID 303024), 2015.

[15] Sieniek M., Paszyński M., Madej L., Goik D.: Adaptive Projection-Based Interpolation as a pre-processing tool in the Finite Element workflow for elasticity simulations of the dual phase microstructures. *Steel Research International*, vol. 85, pp. 1109–1119, 2014.

Affiliations

Maciej Woźniak
AGH University of Science and Technology, Krakow, Poland, macwozni@agh.edu.pl

Maciej Paszyński
AGH University of Science and Technology, Krakow, Poland, paszynsk@agh.edu.pl

Received: 3.04.2015
Revised: 16.01.2016
Accepted: 18.01.2016