paszynsk@agh.edu.pl

MATLAB implementation of the adaptive algorithm for the bitmap projection problem

In this module, we present the MATLAB code that computes the L2 bitmap projection using the adaptive finite element method.
In particular, this code has an implementation of the h-adaptation algorithm in MATLAB.
Code execution is also possible in the free Octave environment. The code is presented below

```
% This is a implementation of h-adaptive bitmap projection.
%
% How to use
%
% bitmap(filename as a string, number of elements along x axis, , number of elements along y axis, maximu
%
% Examples
%
% bitmap("mp.JPG", 10,10,0.1,3,false)
% bitmap("basket.JPG", 20,20,0.5,1,true)
function bitmap_h(filename,elementsx,elementsy,maxerror,max_refinement_level,color_edges_black)

% read image from file
```

```
XX = imread(filename);

% exctract red, green and blue components
RR = XX(:,:,1);
GG = XX(:,:,2);
BB = XX(:,:,3);

% read size of image
ix = size(XX,1);
iy = size(XX,2);

% global count of vertexes
total_vertexes = 0;
% global count of elements
total_elements = 0;

% element structure contains:
% * vertices - organized as followed:
%
% ul - ur
% |    |
% dl - dr
%
% ul - up-left
% ur - up-right
% dl - down-left
% dr - down-right
%
```

```
% * neighbours (elements) ogranized as followed
% there can be up to two neighbours on each edge
% default neighours (if there is one neighbour) are:
% eul, elu, eru, edl
%
%      eul eur
%      ___ ___
% elu |       | eru
%     |       |
% eld |___ ___| erd
%      edl edr
%
% eul - element-up-left
% eur - element-up-rgiht
% elu - element-left-up
% eld - element-left-down
% eru - element-right-up
% erd - element-right-down
% edl - element-down-left
% edr - element-down-right
%
% * active - we don't delete inactive elements, rather tag them as inactive
% * index - index of element in global elements table
%
elements = struct('dl',{},'ul',{},'dr',{},'ur',{},'active',{},'elu',{},'eld',{},'edl',{},'edr',{},'eul',{
%
% vertex data structure contains
% * x and y coordinates
```

```
% * r,g,b - red, green and blue components
% * index - index of vertex in global vertexes table
% * real - false if vertes is hanging node and has interpolated r,g,b components
%
vertexes = struct('x',{},'y',{},'r',{},'g',{},'b',{},'index',{},'real',{});

% initialize unbroken mesh
init_mesh();

redo_error_test = true;
refinemenet_level = 0;

% repeat until we match maximum local estimation error or maximum refinemenet level
while (redo_error_test && (refinemenet_level < max_refinement_level))
  redo_error_test = false;
% loop through elements
  for i=1:total_elements
% check only active elements
    if (elements(i).active)
% estimate realtive interpolation error in red, green and blue components
      [rr,gg,bb] = estimate_error(i);
% if eny of the errors is higher than our maximum -> break element and repeat entire loop
      if ((rr >= maxerror) || (gg >= maxerror) || (bb >= maxerror))
        redo_error_test = true;
        break_element(i);
      end
    end
  end
```

```
  refinemenet_level = refinemenet_level + 1;
end

% interpolate all active elements - recreate bitmap red green and blue compoments
for i=1:total_elements
  if (elements(i).active)
    interpolate_elem(i,color_edges_black);
  end
end

% recreate bitmap from red, green and blue compoments
RGB=XX;
RGB(:,:,1) = RR;
RGB(:,:,2) = GG;
RGB(:,:,3) = BB;

% display image
imshow(RGB);


% create vertex (non hanging node)
function index=create_vertex(x,y)
  vert.x = x;
  vert.y = y;
  vert.r = RR(x,y);
  vert.g = GG(x,y);
  vert.b = BB(x,y);
  total_vertexes = total_vertexes + 1;
```

```
  vert.index = total_vertexes;
  vert.real = true;
  vertexes(total_vertexes) = vert;
  index = total_vertexes;
end


% create vertex (hanging node)
function index=create_vertex_rgb(x,y,r,g,b)
  vert.x = x;
  vert.y = y;
  vert.r = r;
  vert.g = g;
  vert.b = b;
  total_vertexes = total_vertexes + 1;
  vert.index = total_vertexes;
  vert.real = false;
  vertexes(total_vertexes) = vert;
  index = total_vertexes;
end


% update vertex - when hanging node becomes non-hanging node
function vert_update(index)
  vert = vertexes(index);
  vert.r = RR(vert.x,vert.y);
  vert.g = GG(vert.x,vert.y);
  vert.b = BB(vert.x,vert.y);
```

```
  vert.real = true;
  vertexes(index) = vert;
end


% create initial element
function element=create_element(v1,v2,v3,v4)
  element.dl = v1;
  element.ul = v2;
  element.dr = v3;
  element.ur = v4;

  element.active = true;

% set all neighbours to null
  element.elu = 0;
  element.eld = 0;
  element.edl = 0;
  element.edr = 0;
  element.eul = 0;
  element.eur = 0;
  element.eul = 0;
  element.eru = 0;
  element.erd = 0;

  total_elements = total_elements + 1;
  element.index = total_elements;
end
```

```
% initialize mesh
function init_mesh()
%
% vertexes mapping
%
% v2 -> ul
% v4 -> ur
% v1 -> dl
% v3 -> dr
%
  elem_width = floor(ix / elementsx);
  elem_hight = floor(iy / elementsy);

  x = 0;
  y = 0;

% create all vertexes
  for i=0:elementsy-1
    for j=0:elementsx-1
      vertex = create_vertex(x+j*elem_width+1,y+1);
    end
    vertex = create_vertex(ix,y+1);
    y = y + elem_hight;
  end
  for j=0:elementsx-1
    vertex = create_vertex(x+j*elem_width+1,iy);
```

```
    end
    vertex = create_vertex(ix,iy);

  % crete all elements
    for i=1:elementsy
      for j=1:elementsx
        v1 = (i-1)*elementsy+j + i-1;
        v3 = v1+1;
        v2 = i*elementsy+j+1 + i-1;
        v4 = v2+1;
        element = create_element(v1,v2,v3,v4);
        index = element.index;
  % set neighbours for each element
        if(j~=1)
          element.elu = index-1;
        end
        if(j~=elementsx)
          element.eru = index+1;
        end
        if(i~=1)
          element.edl = index-elementsx;
        end
        if(i~=elementsy)
          element.eul = index+elementsx;
        end
        elements(index) = element;
      end
    end
```

```
end


% interpolate r,g,b components for hanging node
% v1 and v2 are vertexes of given element on edges of broken edge
% v3 is interpolated vertex between v1 and v2
function v3=interpolate_rgb(v1,v2,element)
  elem = elements(element);
  width = vertexes(elem.dr).x - vertexes(elem.dl).x;
  hight = vertexes(elem.ul).y - vertexes(elem.dl).y;
  vert1 = vertexes(v1);
  vert2 = vertexes(v2);

  vert3.x = (vert1.x + vert2.x) /2;
  vert3.y = (vert1.y + vert2.y) /2;
  vert3.x = floor(vert3.x);
  vert3.y = floor(vert3.y);

  xx = vert3.x - vertexes(elem.dl).x;
  yy = vert3.y - vertexes(elem.dl).y;
  [r,g,b] = inpoint(xx,yy,width,hight,elem);

  vert3.r = r;
  vert3.g = g;
  vert3.b = b;
  vert3.real = false;
```

```
    total_vertexes = total_vertexes + 1;
    vert3.index = total_vertexes;
    v3 = total_vertexes;
    vertexes(v3) = vert3;
  end


  % interpolate r,g,b components of a element
  function interpolate_elem(element,color_edges_black)
    elem = elements(element);
    width = vertexes(elem.dr).x - vertexes(elem.dl).x;
    hight = vertexes(elem.ul).y - vertexes(elem.dl).y;
    width = abs(width);
    hight = abs(hight);
    dlx = vertexes(elem.dl).x;
    dly = vertexes(elem.dl).y;

    for xx=0:width
      for yy=0:hight
        [r,g,b] = inpoint(xx,yy,width,hight,elem);
        RR(dlx+xx,dly+yy) = r;
        GG(dlx+xx,dly+yy) = g;
        BB(dlx+xx,dly+yy) = b;
      end
    end

  % create black edges on element if requested
    if (color_edges_black)
```

```
      for xx=0:width
        RR(dlx+xx,dly) = 0;
        GG(dlx+xx,dly) = 0;
        BB(dlx+xx,dly) = 0;

        RR(dlx+xx,dly+hight) = 0;
        GG(dlx+xx,dly+hight) = 0;
        BB(dlx+xx,dly+hight) = 0;
      end

      for yy=0:hight
        RR(dlx,dly+yy) = 0;
        GG(dlx,dly+yy) = 0;
        BB(dlx,dly+yy) = 0;

        RR(dlx+width,dly+yy) = 0;
        GG(dlx+width,dly+yy) = 0;
        BB(dlx+width,dly+yy) = 0;
      end
    end
  end


  % computes r,g,b components of element in given point
  function [r,g,b]=inpoint(xx,yy,width,hight,elem)
    f1 = fi1(xx,yy);
    f2 = fi2(xx,yy);
    f3 = fi3(xx,yy);
```

```
    f4 = fi4(xx,yy);

    r = vertexes(elem.dl).r * f1;
    r = r + vertexes(elem.ul).r * f2;
    r = r + vertexes(elem.dr).r * f3;
    r = r + vertexes(elem.ur).r * f4;
    r = floor(r);

    g = vertexes(elem.dl).g * f1;
    g = g + vertexes(elem.ul).g * f2;
    g = g + vertexes(elem.dr).g * f3;
    g = g + vertexes(elem.ur).g * f4;
    g = floor(g);

    b = vertexes(elem.dl).b * f1;
    b = b + vertexes(elem.ul).b * f2;
    b = b + vertexes(elem.dr).b * f3;
    b = b + vertexes(elem.ur).b * f4;
    b = floor(b);

  % basis functions defined over element
    function ret=fi1(xx,yy)
      x = xx/width;
      y = yy/hight;
      ret = (1-x) * (1-y);
    end

    function ret=fi2(xx,yy)
```

```
    x = xx/width;
    y = yy/hight;
    ret = (1-x) * y;
  end

  function ret=fi3(xx,yy)
    x = xx/width;
    y = yy/hight;
    ret = x * (1-y);
  end

  function ret=fi4(xx,yy)
    x = xx/width;
    y = yy/hight;
    ret = x * y;
  end
end


% if neighbour is already bigger than element that we try to break - we should break it as well
function break_neighbours(index)
  element = elements(index);

  check_left();
  check_right();
  check_up();
  check_down();
```

```
  function check_left()
% no neighbours on the left
    if (element.elu == 0)
      return;
    end
% two neighbours on the left
    if(element.eld ~= 0)
      return;
    end
% only one neighbour on the left
    left = elements(element.elu);
    if (left.erd ~= 0)
% neighbour on the left has two neighbours on the right
      break_element(element.elu);
    end
  end

  function check_right()
% no neighbours on the right
    if (element.eru == 0)
      return;
    end
% two neighbours on the right
    if (element.erd ~= 0)
      return;
    end
% only one neighbour on the right
```

```
      right = elements(element.eru);
      if (right.eld ~= 0)
    % neighbour on the right has two neighbours on the left
        break_element(element.eru);
      end
    end

    function check_up()
    % no neighbours on the top
      if (element.eul == 0)
        return;
      end
    % two neighbours on the top
      if (element.eur ~= 0)
        return;
      end
    % only one neighbour on the top
      up = elements(element.eul);
      if (up.edr ~= 0)
    % neighbour on the top has two neighbours on the bottom
        break_element(element.eul);
      end
    end

    function check_down()
    % no neighbours on the bottom
      if (element.edl == 0)
        return;
```

```
    end
% two neighbours on the bottom
    if (element.edr ~= 0)
      return;
    end
% only one neighbour on the bottom
    down = elements(element.edl);
    if (down.eur ~= 0)
% neighbour on the bottom has two neighbours on the top
      break_element(element.edl);
    end
  end
end


% breaking element
function break_element(index)

  element = elements(index);
  if (~element.active)
    disp('error!!!');
  end
  break_neighbours(index);
  element = elements(index);
% vertexes of element are organized as followed
%
% ul - ur
% |    |
```

```
    % dl - dr
    %
    % they are mapped to local vertices
    %
    % v2 - v4
    % |  e  |
    % v1 - v3
    %
    %
    % after breaking element vertices and new elements are organized as followed
    %
    % v2 - v9 - v4
    %  | e2 | e4 |
    % v6 - v7 - v8
    %  | e1 | e3 |
    % v1 - v5 - v3
    %
    %
    % e  -> e2 e4
    %       e1 e3
    %

      v1 = element.dl;
      v2 = element.ul;
      v3 = element.dr;
      v4 = element.ur;

      v5=0;
```

```
    v6=0;
    v7=0;
    v8=0;
    v9=0;

  % if we have two neighbours left
    if (element.eld ~= 0)
      eld = elements(element.eld);
      v6 = eld.ur;
      vert_update(v6);
  % if we have unbroken neighbour left
    else
      v6 = interpolate_rgb(v1,v2,index);
    end
    if (element.elu == 0)
      vert_update(v6);
    end

  % if we have two neighbours right
    if (element.erd ~= 0)
      erd = elements(element.erd);
      v8 = erd.ul;
      vert_update(v8);
  % if we have unbroken neighbour right
    else
      v8 = interpolate_rgb(v3,v4,index);
    end
    if (element.eru == 0)
```

```
      vert_update(v8);
    end

  % if we have two neighbours up
    if (element.eur ~= 0)
      eur = elements(element.eur);
      v9 = eur.dl;
      vert_update(v9)
  % if we have unbroken neighbour up
    else
      v9 = interpolate_rgb(v2,v4,index);
    end
    if (element.eul == 0)
      vert_update(v9);
    end

  % if we have two neighbours down
    if (element.edr ~= 0)
      edr = elements(element.edr);
      v5 = edr.ul;
      vert_update(v5);
  % if we have unbroken neighbour down
    else
      v5 = interpolate_rgb(v1,v3,index);
    end
    if (element.edl == 0)
      vert_update(v5);
    end
```

```
    x = vertexes(v5).x;
    y = vertexes(v6).y;

    v7 = create_vertex(x,y);

    element.active = false;
    elements(element.index) = element;

    e1 = create_element(v1,v6,v5,v7);
    e2 = create_element(v6,v2,v7,v9);
    e3 = create_element(v5,v7,v3,v8);
    e4 = create_element(v7,v9,v8,v4);

  % set neighbours between new elements
    e1.eru = e3.index;
    e1.eul = e2.index;
    e2.edl = e1.index;
    e2.eru = e4.index;
    e3.elu = e1.index;
    e3.eul = e4.index;
    e4.elu = e2.index;
    e4.edl = e3.index;

  % set neighbours between new and old elements
    e1.edl = element.edl;
    if (element.edl ~= 0)
      edl = elements(element.edl);
```

```
    edl.eul = e1.index;
    elements(edl.index) = edl;
  end
  if (element.edr ~= 0)
    e3.edl = element.edr;
    edr = elements(element.edr);
    edr.eul = e3.index;
    elements(edr.index) = edr;
  else
    e3.edl = element.edl;
    if (element.edl ~= 0)
      edl = elements(element.edl);
      edl.eur = e3.index;
      elements(edl.index) = edl;
    end
  end

  e2.elu = element.elu;
  if (element.elu ~= 0)
    elu = elements(element.elu);
    elu.eru = e2.index;
    elements(elu.index) = elu;
  end
  if (element.eld ~= 0)
    e1.elu = element.eld;
    eld = elements(element.eld);
    eld.eru = e1.index;
    elements(eld.index) = eld;
```

```
    else
      e1.elu = element.elu;
      if (element.elu ~= 0)
        elu = elements(element.elu);
        elu.erd = e1.index;
        elements(elu.index) = elu;
      end
    end

    e2.eul = element.eul;
    if (element.eul ~= 0)
      eul = elements(element.eul);
      eul.edl = e2.index;
      elements(eul.index) = eul;
    end
    if (element.eur ~= 0)
      e4.eul = element.eur;
      eur = elements(element.eur);
      eur.edl = e4.index;
      elements(eur.index) = eur;
    else
      e4.eul = element.eul;
      if (element.eul ~= 0)
        eul = elements(element.eul);
        eul.edr = e4.index;
        elements(eul.index) = eul;
      end
    end
```

```
    e4.eru = element.eru;
    if (element.eru ~= 0)
      eru = elements(element.eru);
      eru.elu = e4.index;
      elements(eru.index) = eru;
    end
    if (element.erd ~= 0)
      e3.eru = element.erd;
      erd = elements(element.erd);
      erd.elu = e3.index;
      elements(erd.index) = erd;
    else
      e3.eru = element.eru;
      if (element.eru ~= 0)
        eru = elements(element.eru);
        eru.eld = e3.index;
        elements(eru.index) = eru;
      end
    end

    elements(e4.index) = e4;
    elements(e3.index) = e3;
    elements(e2.index) = e2;
    elements(e1.index) = e1;
  end
```

```
% estimate relative error of interpolation over given element
function [error_r,error_g,error_b]=estimate_error(index)
  element = elements(index);
  dl = element.dl;
  ul = element.ul;
  dr = element.dr;
  ur = element.ur;

  xl = vertexes(dl).x;
  yd = vertexes(dl).y;
  xr = vertexes(ur).x;
  yu = vertexes(ur).y;

  elementWidth = xr - xl;
  elementHeigth = yu - yd;

% interpolate using L2 norm and Gaussian quadrature rule
  x1 = elementWidth/2.0 - elementWidth / (sqrt(3.0) * 2.0);
  x2 = elementWidth/2.0 + elementWidth / (sqrt(3.0) * 2.0);
  y1 = elementHeigth/2.0 - elementHeigth / (sqrt(3.0) * 2.0);
  y2 = elementHeigth/2.0 + elementHeigth / (sqrt(3.0) * 2.0);

  x1 = floor(x1);
  x2 = floor(x2);
  y1 = floor(y1);
  y2 = floor(y2);

  [r1,g1,b1]=inpoint(x1,y1,elementWidth,elementHeigth,element);
```

```
[r2,g2,b2]=inpoint(x1,y2,elementWidth,elementHeigth,element);
[r3,g3,b3]=inpoint(x2,y1,elementWidth,elementHeigth,element);
[r4,g4,b4]=inpoint(x2,y2,elementWidth,elementHeigth,element);

r1 = r1 - RR(x1+xl,y1+yd);
g1 = g1 - GG(x1+xl,y1+yd);
b1 = b1 - BB(x1+xl,y1+yd);

r2 = r2 - RR(x1+xl,y2+yd);
g2 = g2 - GG(x1+xl,y2+yd);
b2 = b2 - BB(x1+xl,y2+yd);

r3 = r3 - RR(x2+xl,y1+yd);
g3 = g3 - GG(x2+xl,y1+yd);
b3 = b3 - BB(x2+xl,y1+yd);

r4 = r4 - RR(x2+xl,y2+yd);
g4 = g4 - GG(x2+xl,y2+yd);
b4 = b4 - BB(x2+xl,y2+yd);

error_r = r1*r1 + r2*r2 + r3*r3 + r4*r4;
error_g = g1*g1 + g2*g2 + g3*g3 + g4*g4;
error_b = b1*b1 + b2*b2 + b3*b3 + b4*b4;

error_r = double(error_r);
error_g = double(error_g);
error_b = double(error_b);
```

```
    error_r = sqrt(error_r) * 100.0 / (255.0 * 2.0);
    error_g = sqrt(error_g) * 100.0 / (255.0 * 2.0);
    error_b = sqrt(error_b) * 100.0 / (255.0 * 2.0);
  end



  end
```

Listing 1 (Pobierz): Adaptacyjna projekcja bitmapy

In order to run the codes, we save them in the Octave working directory.
We set the variables with the path to the input file in tif format
$filename =' C:\Users\Maciej\Dropbox\bitmapa.tif'$
then we give the number of mesh elements in the x and y directions, and the degrees of B-spline in these directions
$elementsx = 4, elementsy = 4$ are the size of the initial grid.
We enter the maximum approximation error
$maxerror = 0.5,$
the maximum number of adaptation iterations $maxlevel = 4$
and indicate whether the mesh edges are to be drawn
$edges = 1.$
Then we start the first procedure
$bitmap\_h(filename, elementsx, elementsy, maxerror, maxlevel, edges).$
The code after executing the adaptation sequence draws the bitmap in the open window.

Utworzona przez admin. Ostatnia aktualizacja: Środa 28 z Październik, 2020 08:50:00 UTC przez paszynsk@agh.edu.pl. Autor: Maciej Paszynski

STATUS: W opracowaniu Zgłoś do recenzji Edytuj

Jak to działa?
O e-podręcznikach AGH
Regulamin
Polityka prywatności
Licencja CC BY-SA

Partnerzy
Kontakt
Prześlij opinię
About

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie
Centrum e-Learningu

Wersja mobilna