SCI INSTITUTE
TECHNICAL REPORT

Minimum and Maximum Patch Size Clustering on a Single Refinement Level

Oren E. Livne

UUSCI-2006-002

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

January 23, 2006

Abstract:

In Report 1 we described the Berger-Rigoustos (BR) algorithm for clustering points. The algorithm needs to be modified for our CSAFE needs. In particular, we have to add controls over the minimum and maximum patch size. In this report we introduce a modified BR algorithm, whose implementation is based on a more systematic philosophy in terms of *actions* taken to dissect boxes. We show how to add to the list of actions tests on the minimum and maximum box size, and study the effect of these parameters on the algorithms efficiency.

THE UNIVERSITY OF UTAH

Minimum and Maximum Patch Size Clustering on a Single Refinement Level

Oren E. Livne *

January 23, 2006

Abstract

In Report 1 we described the Berger-Rigoustos (BR) algorithm for clustering points. The algorithm needs to be modified for our CSAFE needs. In particular, we have to add controls over the minimum and maximum patch size. In this report we introduce a modified BR algorithm, whose implementation is based on a more systematic philosophy in terms of *actions* taken to dissect boxes. We show how to add to the list of actions tests on the minimum and maximum box size, and study the effect of these parameters on the algorithm's efficiency.

Key words. Clustering, dissection actions, patches.

1 Introduction

Our ultimate goal is to develop an algorithm to generate rectilinear patches from a given set of flagged cells ("cells that need further refinement") in the ICE code. A natural candidate is the Berger-Rigoustos (BR) algorithm, that was discussed and implemented in its classical form in Report 1. However, the goals we described in Report 1 were not attained – on purpose, though: we wanted to first address part of the goals, and add the rest in a modular way in subsequent versions of the algorithm.

*SCI Institute, 50 South Central Campus Dr., Room 3490, University of Utah, Salt Lake City, UT 84112.Phone: +1-801-581-4772. Fax: +1-801-585-6513. Email address: livne@sci.utah.edu

To achieve this modularity, we reformulate the BR algorithm as a "smart bisection" algorithm that recursively strives to dissect any "bad patch". For every such bad patch, we perform a series of *actions*, trying to find a good cutting plane (again we stick in this report to 2D experiments, where this is a 2D line). More actions can now be added to accommodate more control parameters. In accordance with our goals, we added two more parameters: the minimum permitted side length of a box, and the maximum allowed box volume.

In §2 we review our goals, describe the new algorithm, and include a MATLAB code of its main parts. In §3, we study the algorithm's efficiency (how much non-flagged cells are covered by the final set of boxes) versus the each of the parameters. We also run our standard test cases to check the algorithm's efficiency versus the BR algorithm of Report 1.

In sum, we achieved four of the five objectives of the algorithm. The fifth objective (minimum flag distance from box boundary) is expected to be achieved at the next step of development, by a pre-processing step of *dilating* the area of the flagged cells. This objective is left last, because it has implications on *multi-* level refinement, whereas the rest can be discussed in the context of a single refinement level.

2 The New Algorithm

We recall that for a general set of flagged cells, we focus on obtaining a set of covering non-overlapping boxes. Our ideal objectives for the boxes are as follows.

1. *Objective 1 - Maximum efficiency:* the ratio of the number of flagged cells to the total box area, should be as close to 1 as possible. We would like to minimize the wasted "blank space" by the rectangles: the total work and storage of the patches in the actual solver is proportional to the total box area.

2. *Objective 2 - Minimum box size:* the smallest box should not be less than (say) 4 cells in every direction. Otherwise, there would be a large overhead that would not justify the use of such boxes.

3. *Objective 3 - Maximum box volume:* patches that are 32^3 or $64 \times 64 \times 8$ are equivalent in terms of memory and cost, but we would not want very

large box volumes in light of a worse load balancing between processors, in a parallel processing framework.

4. *Objective 4 - Minimum flag distance from box boundary:* the boxes have to be re-generated every (several) time-steps, and the location of the flags move (e.., a shock wave front). We want to keep the same rectangles for as long as possible, and so would not want flagged cells right near the box edges, as they might move out in the next time-step.

5. *Objective 5 - Minimum box mutual-boundary area:* to minimize processor communication, we would like the boxes to have as low mutual edges as possible. One way to indirectly achieve this is by trying to construct more "cubic" boxes than "thin" ones, thereby reducing the edge area of each box (independently of the other boxes' edge area, though).

In Report 1 we treated Objectives 1 and 5. In this report we revise the algorithm to include Objectives 2 and 3. Objective 4 will be discussed in a future report.

2.1 The Philosophy of Actions

The core of the BR algorithm [BR91],[JBW94] is a recursive procedure that takes any box that is not yet acceptable, and dissects it into two smaller boxes (or leaves it intact, if nothing can be done). In [BR91], a box is processed only if its efficiency is below the required threshold. Each such "bad box" is passed a series of tests: we try to look for *holes* (a line without any flagged cells); if a hole is found, we choose it and proceed to the next box. Otherwise, we proceed to look for inflection points, and so on. This form of the algorithm requires the code to have several recursive "if-statements" that become very complicated with the addition of minimum and maximum box size controls.

Alternatively, we propose to slightly revise the algorithm's flow: we loop over boxes until we process all of them; for each box under consideration, we *loop* over a series of *actions*. Each action is a piece of code that tries outputs the cut plane direction and location, or indicates that a cut was not found. The loop is performed until a cut is found. Although this seems almost identical to the description of the previous paragraph, we replace the recursive "ifs" by a single "switch" statement, making it much easier to add

new actions, and to distinguish between actions. A general pseudo-code of this algorithm follows.

```
box_index = 1;
while (box_index <= num_boxes)
  Look at box number k, compute signatures.
  cut_found = 0;
  for action = 1:num_actions
    switch(action)
      case 1:
      if (eff >= threshold)
         if (box_volume > max_volume)
           dissect_because_big_box;
         end if
         if (not cut_found)
           break_the_for_loop;
         end if
      end
      case 2: look_for_holes;
      case 3: look_for_inflection_points;
      case 4:
         if (efficiency <= 0.5)
           if (not rectangle_too_small) dissect_rectangle;
           end if
         else
           if (box_volume > max_volume)
             dissect_because_big_box;
           end if
         end if
      end switch
      if (cut_found)
        break_the_for_loop;
      end if
   end for
   if (cut_found)
     replace box k by its two halves, numbered k and k+1;
   else
     k = k+1;
```

```
   end
end while
```

Note that we have to check whether the box is too big in two places in the code. We check whether the rectangle is too small when trying to dissect it, but also when we look for holes (action 2) and inflection points (action 3), and when we actually construct the two halves (k and $k+1$).

2.2 Treating Small Boxes

We start by defining the input parameter s as the minimum allowed side length of a box (in any dimension - x or y). Any box with side length $s-1$ or less is not allowed to be part of our box covering.

The parameter s affects the following parts of the algorithm (in the sequel, the current box-under-consideration is $[x_1, y_1] \times [x_2, y_2]$:

- Holes: when we look for hole locations, we allow only holes in the range $[x_1 + s, x_2 - s]$ in the x-dimension, and in $[y_1 + s, y_2 - s]$ in the y-dimension. The actual cut is performed between the location x^* and $x^* + 1$ (similarly in y), and it is easy to see that by limiting ourselves to these ranges, we allow the smaller "half" of the box to have a side length s or larger.

- Inflection points: again, we look for the sharpest edge *within* the ranges $[x_1 + s, x_2 - s]$ and $[y_1 + s, y_2 - s]$, for the x- and y- inflection searches, respectively.

- Dissection: if a box admits no holes or inflection points, we check its efficiency. If it is less than 50%, the standard algorithm dissects the box along the longest dimension. Instead, we first check whether the longest dimension is at least $2s$; if yes, we dissect; if no, this box cannot be dissected. Note that only boxes with minimum side length between s and $2s$ fall into this category, thus in practice we might end up with some boxes that are larger than the minimum side length (that is perfectly acceptable!).

- Tight bounding boxes: when we construct the two halves (k and $k+1$) after deciding to dissect the "old-k" rectangle, we originally reduced the two halves to the smallest ("tight") bounding box around the flagged

cells in their respective areas. Instead, we now use a more sophisticated method of finding the smallest allowed bounding box: we first find the tight bounding box around the flagged cells (say, in the left half of the old box). For every dimension, if it is smaller than the minimum box size in this dimension, we *increase* it back to the minimum size, and then *shift* it so that it is still a subset of the original box (see Fig. 1).

2.3 Treating Large Boxes

The control parameter for large boxes is a maximum permitted volume parameter, v. Large boxes are easier to handle, because we should take care of them only if we decide *not* to dissect the rectangle. This happens at two instances in the code:

- If the rectangle already has a high enough efficiency, we still check whether it is too large, and dissect if it is.
- If a rectangle is not dissected by holes or inflection points or the default bisection for efficiency less than .5, we check if it is too large. If it is, we dissect it.

Because only efficient boxes are checked for being large, the dissection of a box whose volume exceeds v is done to maximize the efficiency of its halves, rather to just bisect it. We search the longest dimension only. For simplicity, assume this is the x dimension. Then, we choose the cut in $[x_1 + s, x_2 - s]$ (to conform to the minimum box size criterion), that minimizes the ratio of the largest to smallest efficiency of the two halves. Although their efficiency might be lower than the original box, it should not be too low, because the original box is efficient.

2.4 Full MATLAB Code

The input for the algorithm is a list of gridpoints flagged as needing refinement. In the description below, we use the `flag` input array as a 2D binary image. In addition, the structure `opts` contains the various control parameters. If `opts` is omitted, we use some default values.

The output is a list of boxes, that is, an $K \times 4$ array, where K is the number of boxes, and each box is designated by (x_1, y_1, x_2, y_2): its lower-left

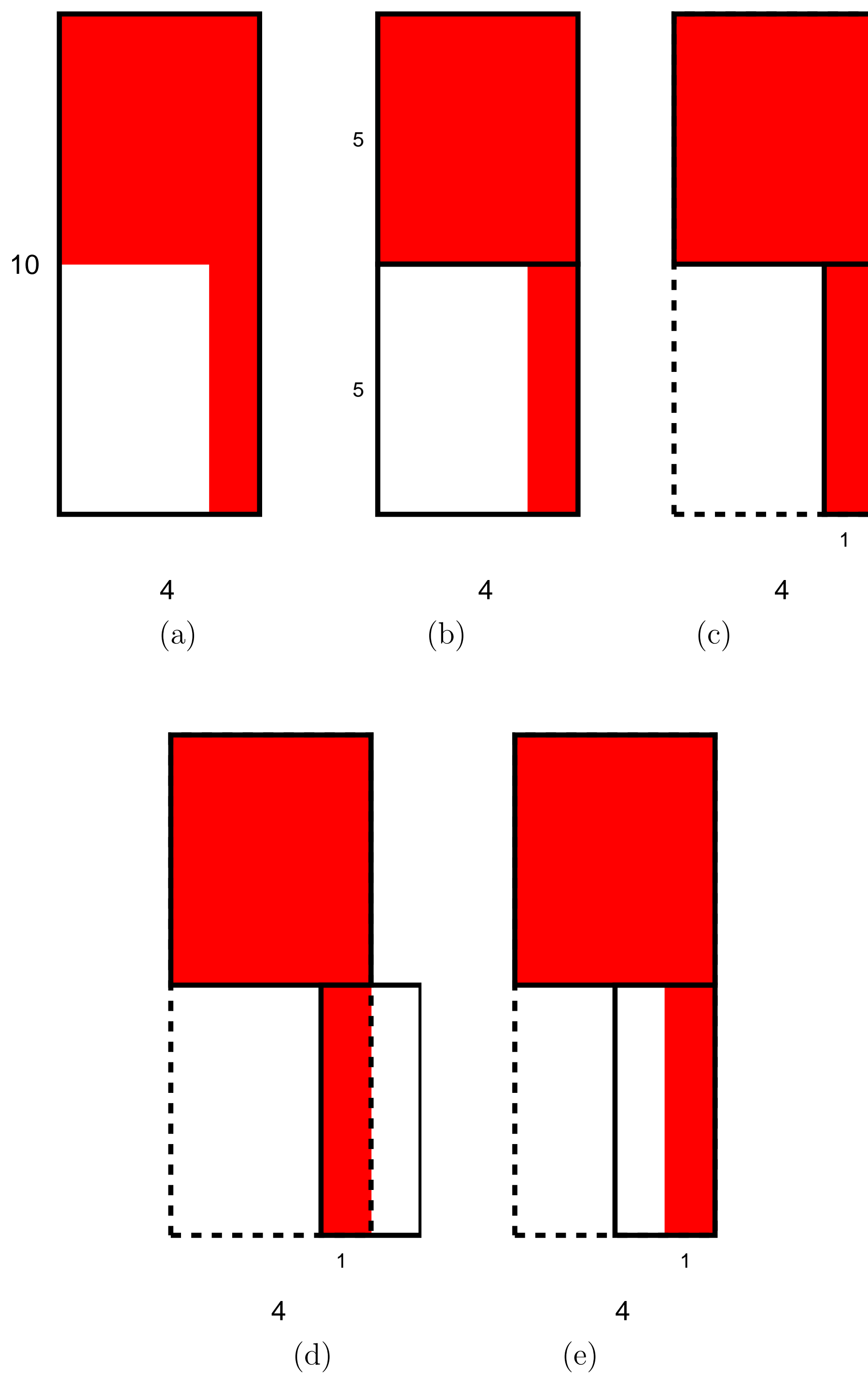


Figure 1: An example of generating two smaller boxes from a big one. (a) The red area denotes flagged cells. They are surrounded by the box 4×10. (b) For instance, suppose we decide to bisect in the y-dimension. (c) Tight bounding boxes around the red areas. (d) If the minimum allowed side length is 2, we have to extend the lower box. (e) If the extended box steps outside the old box, we shift it back inside.

corner (x_1, y_1), and upper-right corner (x_2, y_2). This output array is denoted by `rect`.

```
function rect = cluster_br_3(points,opts);
if (nargin < 2)
end
opts.efficiency = 0.8;                                      % Lowest box efficiency allowed
opts.low_eff    = 0.5;                                      % Efficiency threshold for boxes that don't have holes/inflections
opts.min_side   = 4; %4;                                    % Minimum box side length allowed (in all directions)
opts.max_volume = 100;                                      % Maximum box volume allowed
opts.pause      = 0;

fprintf('Parameters:\n');                                   % Print parameters
fprintf('Efficiency threshold: %.1f%%\n',100*opts.efficiency);  % Efficiency threshold
fprintf('Min. box side length: %d\n',opts.min_side);        % Min. box size
fprintf('Max. box volume     : %d\n',opts.max_volume);      % Max. box volume

%%%%%%%%%% Initialize and find the first box
dim         = length(size(points));                         % Dimension of the problem
[i,j]       = find(points);                                 % Index arrays [i,j] of flagged cells
rect        = [min(i) min(j) max(i) max(j)];                % Bounding box for flagged cells
num_actions = 4;

%%%%%%%%%% Main algorithm: loop over boxes, process them, and possiblly add more boxes
k       = 1;                                                % Index of box to be processed
while (k <= size(rect,1))                                   % Do until all boxes have been processed
    r               = rect(k,:);                            % Box coordinates
    s               = points(r(1):r(3),r(2):r(4));          % Flag data of this box
    sz              = box_size(r);                          % Vector containing the size of the box: [size_x,size_y]
    efficiency      = length(find(s))/prod(sz);             % Percentage of flagged cells in s
    [a,sorted_dims] = sort(-sz);                            % Sort box sizes in descending orders
    sig             = compute_signatures(s);                % Compute signatures
    fprintf('Considering box #%3d at coordinates [%3d,%3d,%3d,%3d]   size = %d x %d,  vol = %d, efficiency = %f\n',k,r,sz+1,box_volume(r),efficiency);

    %%%%% Plot-outs: plot the points and the current boxes. The considered box is in red.
    if (opts.pause)
        figure(1);
        clf;
        plot_points(points);
        hold on;
        plot_boxes(rect);
        offset = 0.2;
        h = rectangle('Position',[rect(k,1:2)-offset,[rect(k,3:4)-rect(k,1:2)]+2*offset]);
        set(h,'EdgeColor','Red');
        set(h,'LineWidth',2);
        pause
    end

    %%%%% Loop over actions to find a cut
    cut.found   = 0;                                        % Start: we don't know where to dissect the box
    for action = 1:num_actions,                             % Try different actions to find a dissection plane ("cut")
        switch(action)                                      % Each action is attached to a certain piece of code below
        case 1
            fprintf('Action 1: check efficiency; if efficienct but box too big, dissect\n');
            if (efficiency >= opts.efficiency)              % box efficient, but check if it's too big
                fprintf('Box has the required efficiency\n');
                if (box_volume(r) > opts.max_volume)        % Box too big
                    cut.dim = sorted_dims(1);               % Longest dimension
                    cut     = dissect_big_box(points,r,...
                        sig,cut.dim,opts.min_side);         % Dissect it
                end
                if (~cut.found)
                    break;
                end
            end
        case 2
            fprintf('Action 2: looking for holes\n');
            cut = find_hole(r,sig,sorted_dims,opts.min_side);   % Look for a hole
        case 3
            fprintf('Action 3: looking for inflection points\n');
            cut = find_inflection(r,sig,opts.min_side);     % Look for an inflection point
        case 4                                              % No holes or inflection points; base box acceptance on its efficiency; bisect if not eff
            fprintf('Action 4: no holes or inflections, dissect if not efficienct and not too small; otherwise, check if too big\n');
            cut.dim             = sorted_dims(1);           % Longest dimension
            if (efficiency <= opts.low_eff)                 % If box not efficient (efficiency <= 50% - the diagonal black case included, = 50%)
                if (sz(cut.dim) >= 2*opts.min_side)         % Bissect only if the halves are still larger than the minimum side permitted
```

```
                    cut.found       = 1;                        % We will cut, this time - bisect
                    cut.place       = floor(...
                        sz(cut.dim)/2)+r(cut.dim)-1;            % The middle absolute coordinate of this dimension
                    fprintf('Bisecting box because efficiency = %f < %f and side size = %d > min_side\n',efficiency,opts.low_eff,sz(cut.dim));
                end
            else                                                % Efficiency > 50%
                if (box_volume(r) > opts.max_volume)            % Box too large
                    cut = dissect_big_box(points,r,sig,cut.dim,opts.min_side);
                end
            end
        otherwise
            fprintf('Unknown action! exiting\n');
            rect = [];
            return;
        end
        if (cut.found)
            break;
        end
    end

    %%%%% Make the cut: replace the current box by its two "halves"
    if (cut.found)
        %%%%% Plot-outs and printouts: plot the cutting plane in green
        if (opts.pause)
            fprintf('This box is dissected at cut.dim = %d, cut.place = %3d\n',cut.dim,cut.place);
            if (cut.dim == 1)                                   % This code is specific for 2D, need to generalize to dim-D later
                h = line([cut.place cut.place]+0.5,[r(2)-0.2,r(4)+0.2]);
            else
                h = line([r(1)-0.2,r(3)+0.2],[cut.place cut.place]+0.5);
            end
            set(h,'LineWidth',3);
            set(h,'Color','Green');
        end

        rn          = dissect_box(points,r,cut,opts.min_side);
        rect        = [rect; rn{1}; rn{2}];                     % Add the two halves to the list
        rect(k,:)   = [];                                       % Delete box k from the list, so now k points to the "next box" to be considered
        if (opts.pause)
            pause
        end
    else
        fprintf('This box is accepted\n');
        k = k+1;                                                % Couldn't find a cut accept this box and consider the next box on the list
    end
    fprintf('\n');
end

final_stats(points,rect);                                       % Final printouts (overall statistics) and plot-outs


%---------------------------------------------------------------------------------------------------------
function sig = compute_signatures(s)
% Compute signatures of a rectilinear patch s

dim         = length(size(s));                                  % Dimension of the problem
sig         = cell(dim,1);                                      % Signature in all dimensions
for d = 1:dim                                                   % Loop over dimensions
    sig{d} = s;                                                 % Start from the box
    for j = 1:dim                                               % For all dimensions...
        if (j ~= d)                                             % Loop over all dimensions except d
            sig{d} = sum(sig{d},j);                             % Sum along the j's dimension
        end
    end
end

%---------------------------------------------------------------------------------------------------------
function cut = find_hole(r,sig,sorted_dims,min_side)
% Look for a hole, given the signature arrays sig and the minimum side length min_side. Sorted_dims
% specifies the order by which we loop over the dimensions when we look for holes

cut.found   = 0;                                                % Default: no cut found
for d = sorted_dims                                             % Loop over dimensions in decreasing box size
    len     = length(sig{d});                                   % Length of box in this dimension
    hole    = find(sig{d} == 0);                                % Look for holes in direction d
    hole    = intersect(hole,[min_side:len-min_side]);          % Do not allow holes that are too close to the boundaries, because we need to keep a mini
    if (~isempty(hole))                                         % If found hole...
        center      = len/2+0.5;                                % Center coordinate of this direction (origin at 1)
```

```
        distance    = abs(hole+0.5-center);                      % Distance of holes from the center
        best_hole   = hole(find(distance == min(distance)));     % Find the holes closest to the center
        cut.found   = 1;                                         % Found a cut, flag up
        cut.place   = best_hole(1)+r(d)-1;                       % Choose one of them and convert back into absolute coordinates
        cut.dim     = d;                                         % Dimension along which we cut
        fprintf('Found hole\n');                                 % Printout
        break;                                                   % We terminate the loop over dimensions when we find a legal hole
    end
end

%-----------------------------------------------------------------------------------------------------------
function cut = find_inflection(r,sig,min_side)
% Look for an inflection point (a zero-crossing in the second derivative of the signature), given the signature arrays sig
% and the minimum side length min_side.

cut.found   = 0;                                                 % Default: no cut found
sz          = box_size(r);                                       % Rectangle side lengths
dim         = size(sig,1);                                       % Dimension of the problem
delta       = cell(dim,1);                                       % Second-derivative-of-signature in all dimensions
best_place  = -ones(dim,1);                                      % Absolute coordinate of best place to cut
value       = -ones(dim,1);                                      % Sharpness value of edge; -1 = dummy (no allowed edge found)
for d = 1:dim                                                    % Loop over dimensions
    delta{d} = diff(diff(sig{d}));                               % Discrete second-derivative
    schange_abs = abs(diff(delta{d}));                           % Gradient absolute value
    schange = zeros(size(schange_abs));                          % Array of flags of sign changes in delta
    for i = 1:length(delta{d})-1                                 % Loop over the delta array in this direction
        schange(i) = sign(delta{d}(i)*delta{d}(i+1));            % sign change from i to i+1: 1, none; 0, one of them is zero (so sign change); -1, sign c
    end
    len         = length(sig{d});                                % Length of box in this dimension
    zero_cross  = find(schange <= 0);                            % Indices i for which delta changes sign (i -> i+1)
    zero_cross  = intersect(zero_cross,...
        [min_side:len-min_side]-1);                              % Do not allow zero crossing that are too close to the boundaries, because we need to kee
    if (~isempty(zero_cross))                                    % If there exist zero crossing...
        edge            = schange_abs(zero_cross);               % Save only the relevant indices (zero_cross) in schange_abs
        max_cross_value = max(edge);                             % Find the sharpest edge value
        max_cross       = zero_cross(...
            find(edge == max_cross_value)) + 1;                  % Find where are the sharpest edge locations; +1 because delta is defined on [2..len-1] a
        center          = len/2+0.5;                             % Center coordinate of this direction (origin at 1)
        distance = abs(max_cross+0.5-center);                    % Distance of zero-crossings for the center
        best_cross = max_cross(...
            find(distance == min(distance)));                    % Find those closest to the center
        best_place(d) = best_cross(1) + r(d)-1;                  % Covert back to absolute coordinates and save in best_place
        value(d) = max_cross_value;                              % Save edge value for comparison between dimensions
    end
end
zero_cross_dims = find(value >= 0);                              % All dims for which there exists a zero crossing
if (~isempty(zero_cross_dims))                                   % If there exists a zero crossing
    %              best_place
    %              value
    max_cross_dims  = find(value == max(value));                 % Find sharpest edge(s)
    best_dims       =  max_cross_dims(find(...
        sz(max_cross_dims) == max(sz(max_cross_dims))));         % Find the sharpest edge(s) in the longest direction(s)
    cut.found       = 1;                                         % Found a cut, flag up
    cut.dim         = best_dims(1);                              % Dimension along which we cut
    cut.place       = best_place(cut.dim);                       % Coordinate of cut along that dimension
    fprintf('Inflection point found\n');                         % Printout
end

%-----------------------------------------------------------------------------------------------------------
function sz = box_size(r)
% Side length of a box in all dimensions

dim         = length(r)/2;                                       % Dimension of the problem
sz          = r(dim+1:2*dim) - r(1:dim) + 1;                     % r = [x1_start,...,xd_start,x1_end,...xd_end]; +1 because if xi_start=xi_end, size=1

%-----------------------------------------------------------------------------------------------------------
function vol = box_volume(r)
% Volume of a box

sz          = box_size(r);                                       % Rectangle side lengths
vol         = prod(sz);                                          % Total volume

%-----------------------------------------------------------------------------------------------------------
function cut = dissect_big_box(points,r,sig,d,min_side)
% When box is larger than the maximum volume, break it down. r = box coordinates, sig = signature arrays, min_side = minimum
% side length permitted, cut.dim = dimension along which we try to cut (usually the longest)
```

```
fprintf('Box too big\n');
cut.found   = 0;                                            % Default: no cut found
dim         = size(sig,1);                                  % Dimension of the problem
len         = length(sig{d});                               % Length of this side
eff         = zeros(2,len-1);                               % Efficiency of the two halves of the prospective cut coordinate
rn          = cell(2,1);                                    % The two halves
rn{1}       = r;                                            % Create the "left half"
rn{2}       = r;                                            % Create the "right half"
for i = 1:len-1                                             % This could be done more efficiently in a sliding window manner
    rn{1}(d+dim) = r(d)-1+i;                                % The "left half": change the end index in the d-direction to the cut index
    rn{2}(d)     = r(d)-1+i+1;                              % The "right half": change the beginning index in the d-direction to the cut index
    for h = 1:2
        s = points(rn{h}(1):rn{h}(3),rn{h}(2):rn{h}(4));    % Flag data of this box; specific for 2D
        eff(h,i) = length(find(s))/(rn{h}(d+dim)-rn{h}(d)+1);
    end
end
ratio       = max(eff,[],1)./min(eff,[],1);
ratio_inner = ratio(min_side:len-min_side);
best        = find(ratio_inner == min(ratio_inner))+min_side-1;
center      = len/2+0.5;                                    % Center coordinate of this direction (origin at 1)
distance    = abs(best+0.5-center);                         % Distance of zero-crossings for the center
best_center = best(find(distance == min(distance)));        % Find those closest to the center
cut.found   = 1;                                            % Found a cut, flag up
cut.place   = best_center(1)+r(d)-1;
cut.dim     = d;

%-----------------------------------------------------------------------------------------------------------
function rn = dissect_box(points,r,cut,min_side)
% Given a rectangle r and cut information (cut), return rn = the coordinates of the two halves. We try to
% fit the tightest bounding box, up to the restriction of the minimum box side length (min_side). points is
% the array of the flagged points (boolean image)

dim         = length(r)/2;                                  % Dimension of the problem
rn          = cell(2,1);                                    % The two halves
rn{1}       = r;                                            % Create the "left half"
rn{2}       = r;                                            % Create the "right half"
rn{1}(cut.dim+dim) = cut.place;                             % The "left half": change the end index in the d-direction to the cut index
rn{2}(cut.dim)     = cut.place+1;                           % The "right half": change the beginning index in the d-direction to the cut index
s           = points(rn{1}(1):rn{1}(3),rn{1}(2):rn{1}(4));  % Flag data of this box
[i,j]       = find(s);                                      % 2D indices of the points in this box
rn{1}         = [rn{1}(1:2) rn{1}(1:2)]-1 + ...
    [min(i) min(j) max(i) max(j)];                          % Tight bounding box around the points

s           = points(rn{2}(1):rn{2}(3),rn{2}(2):rn{2}(4));  % Flag data of this box
[i,j]       = find(s);                                      % 2D indices of this box
rn{2}         = [rn{2}(1:2) rn{2}(1:2)]-1 + ...
    [min(i) min(j) max(i) max(j)];                          % Tight bounding box around the points
fprintf('New boxes before extending:\n');
fprintf('Left  half coordinates [%3d,%3d,%3d,%3d]  size = %d x %d\n',rn{1},rn{1}(3)-rn{1}(1)+1,rn{1}(4)-rn{1}(2)+1);
fprintf('Right half coordinates [%3d,%3d,%3d,%3d]  size = %d x %d\n',rn{2},rn{2}(3)-rn{2}(1)+1,rn{2}(4)-rn{2}(2)+1);

%%%%% Correct bounding boxes to at least the minimal required side length
for h = 1:2
    for d = 1:dim
        lc          = d;
        rc          = lc+dim;
        slack       = min_side - (rn{h}(rc)-rn{h}(lc)+1);       % If we're too thin, slack is positive
        if (slack > 0)
            ext_left  = floor(slack/2)                      % How much to extend tight bounding box on the left
            ext_right = slack - ext_left;                   % How much to extend tight bounding box on the right
            rn{h}(lc) = rn{h}(lc) - ext_left;               % Left extend
            rn{h}(rc) = rn{h}(rc) + ext_right;              % Right extend
            if (rn{h}(lc) < r(lc))
                rn{h}(lc:rc) = rn{h}(lc:rc) + (r(lc) - rn{h}(lc));
            end
            if (rn{h}(rc) > r(rc))
                rn{h}(lc:rc) = rn{h}(lc:rc) + (r(rc) - rn{h}(rc));
            end
        end
    end
end
fprintf('New boxes after tightening bounding boxes:\n');
fprintf('Left  half coordinates [%3d,%3d,%3d,%3d]  size = %d x %d\n',rn{1},rn{1}(3)-rn{1}(1)+1,rn{1}(4)-rn{1}(2)+1);
fprintf('Right half coordinates [%3d,%3d,%3d,%3d]  size = %d x %d\n',rn{2},rn{2}(3)-rn{2}(1)+1,rn{2}(4)-rn{2}(2)+1);

%-----------------------------------------------------------------------------------------------------------
function final_stats(points,rect)
```

```
% Final printouts (overall statistics) and plot-outs
x               = (rect(:,3)-rect(:,1)+1);
y               = (rect(:,4)-rect(:,2)+1);
rect_area       = x.*y;
total_rect_area = sum(rect_area);
flagged_area    = length(find(points));
fprintf('Total Number of flagged pts : %d\n',flagged_area);
fprintf('Total Area in boxes         : %d\n',total_rect_area);
fprintf('\n');
fprintf('flagged points/tot.box.vol. : %.1f%%\n',100*flagged_area/total_rect_area);
fprintf('Number of boxes             : %d\n',size(rect,1));
fprintf('Minimum box edge            : %d\n',min(min(rect(:,3)-rect(:,1)+1,rect(:,4)-rect(:,2)+1)));
fprintf('Maximum box volume          : %d\n',max(rect_area));
fprintf('Average box volume          : %f\n',mean(rect_area));
fprintf('Average box side ratio      : %.1f%%\n',100*mean(min(x,y)./max(x,y)));

figure(1);
clf;
plot_points(points);
print -depsc cells.eps

figure(2);
clf;
plot_points(points);
hold on;
plot_boxes(rect);
print -depsc cover.eps

%-----------------------------------------------------------------------------------------------------------
function plot_points(points)
% Plot the points in the current figure
[i,j]   = find(points);
h       = plot(i,j,'b.');
set(h,'MarkerSize',10);
axis([min(i)-1 max(i)+1 min(j)-1 max(j)+1]);
axis off;

%-----------------------------------------------------------------------------------------------------------
function plot_boxes(rect)
% Plot the current box covering in the current figure
offset = 0.2;
for i = 1:size(rect,1)
    rectangle('Position',[rect(i,1:2)-offset,[rect(i,3:4)-rect(i,1:2)]+2*offset]);
end
```

3 Numerical Experiments

Each test case is a set of flagged cells in 2D. For each case, we plot the original points and the resulting covering. Tables include summarizing run statistics (see Report 1). In all cases, we used a minimum efficiency threshold of .8.

3.1 Generic Test Cases

We first tested the new algorithm (with the restricting parameters $s = 4$, $v = 100$; note that in the test cases that follow, v is a significant portion of the total area of the flagged points) on the test cases (G1–G4) described in [BR91], and compared the results with the old algorithm. The latter can be viewed as a special case of the former, without restrictions ($s = 0, v = \infty$). The results are summarized in the following tables and figures.

Table 1: Generic Test Cases: Old Algorithm, $s = 0, v = \infty$

Case	Flagged Cells	Tot.Box Volume	Effi- -ciency	# Boxes	Min. Edge	Max. Volume	Avg.Box Volume	Avg.Box Edge.Rat.
G1	614	668	.919	11	2	128	60.7	.336
G2	623	683	.912	13	2	288	52.5	.564
G4	245	281	.872	11	1	110	25.5	.761
G5	233	264	.883	10	1	104	63.8	.638

Table 2: Generic Test Cases: New Algorithm, $s = 4, v = 100$

Case	Flagged Cells	Tot.Box Volume	Effi- -ciency	# Boxes	Min. Edge	Max. Volume	Avg.Box Volume	Avg.Box Edge.Rat.
G1	614	664	.925	15	4	96	44.3	.601
G2	623	688	.906	16	4	80	43.0	.711
G4	245	298	.822	6	5	88	49.7	.710
G5	233	292	.798	6	4	72	48.7	.698

3.2 Hard Test Cases

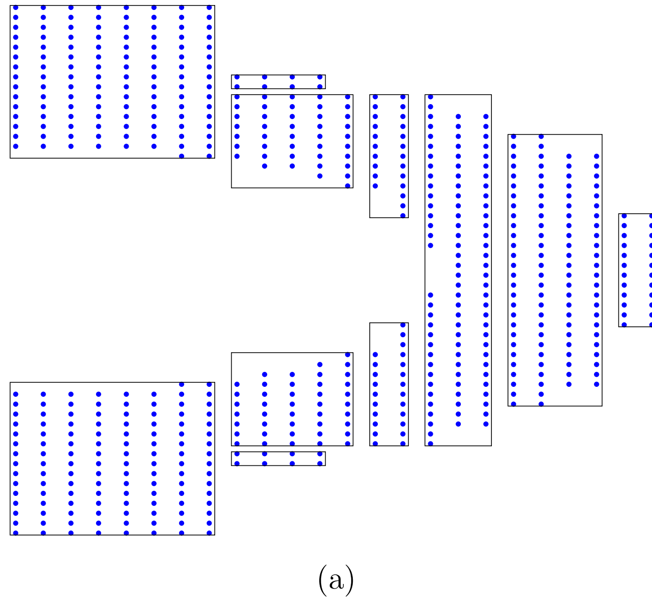
We compare the new and old algorithms the hard test cases from Report 1, §4.2. Because the total number of flagged cells for these test cases is ≈ 100, we use $s = 4$ and $v = 50$ as our control parameters for the new algorithm.

Table 3: Hard Test Cases: Old Algorithm, $s = 0, v = \infty$

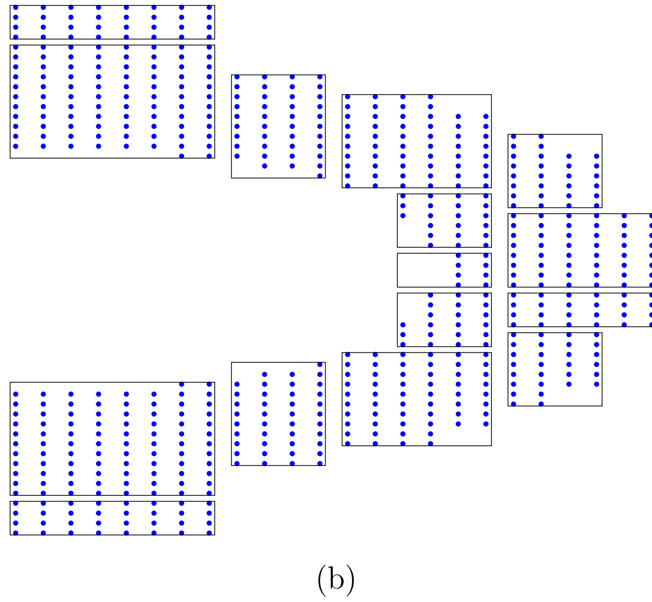
Case	Flagged Cells	Tot.Box Volume	Effi- -ciency	# Boxes	Min. Edge	Max. Volume	Avg.Box Volume	Avg.Box Edge.Rat.
H1	82	84	.976	3	1	42	28	.571
H2	108	108	1.00	2	6	54	54	.667
H3	61	81	.753	41	9	9	2.0	1.00
H4	66	81	.815	30	1	9	2.7	.869

Table 4: Hard Test Cases: New Algorithm, $s = 4, v = 50$

Case	Flagged Cells	Tot.Box Volume	Effi- -ciency	# Boxes	Min. Edge	Max. Volume	Avg.Box Volume	Avg.Box Edge.Rat.
H1	82	105	.781	3	4	42	35	.714
H2	108	108	1.00	4	4	30	27	.750
H3	61	121	.504	36	5	36	30.3	.917
H4	66	171	.386	8	4	25	21.4	.908

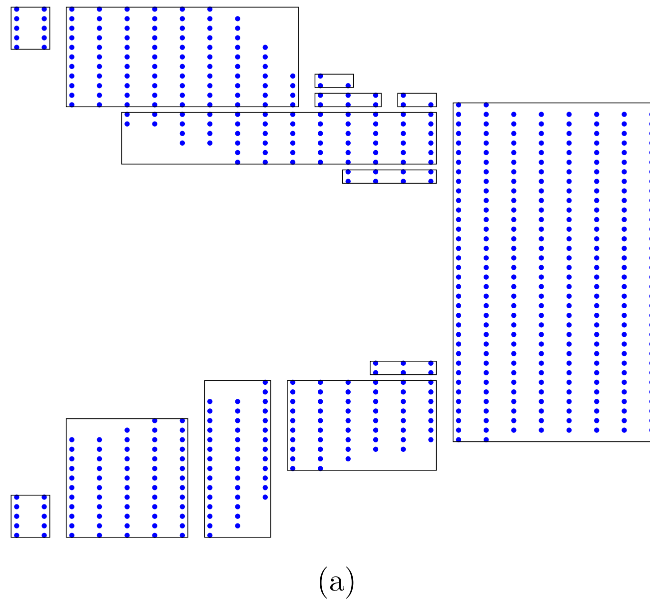


(a)

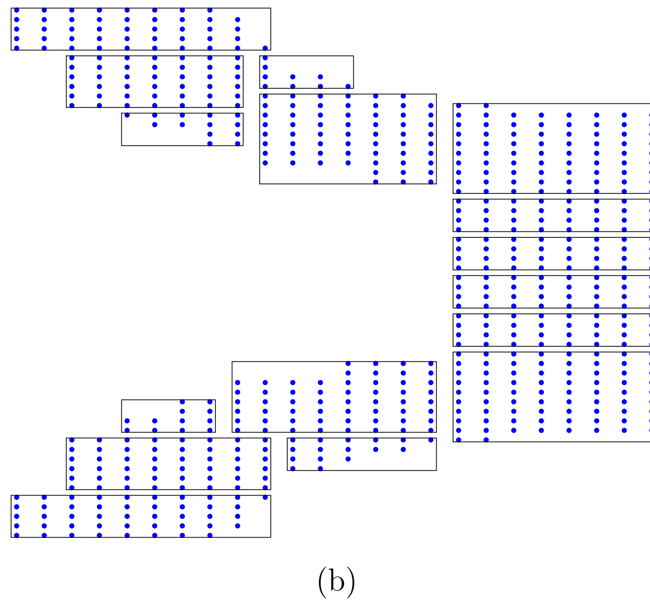


(b)

Figure 2: Test Case G1 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

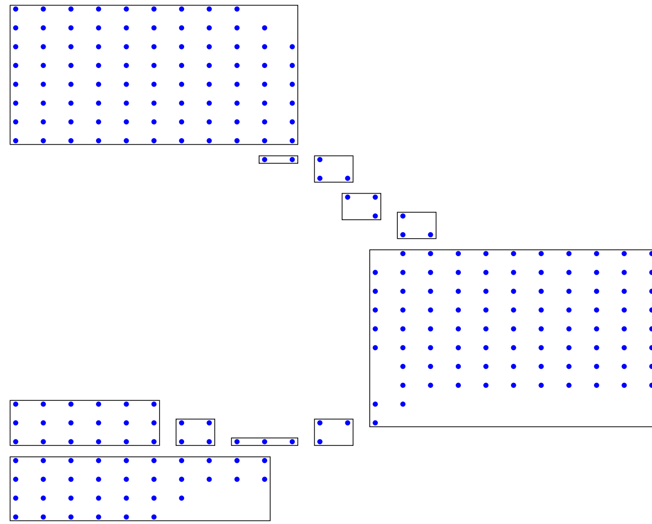


(a)

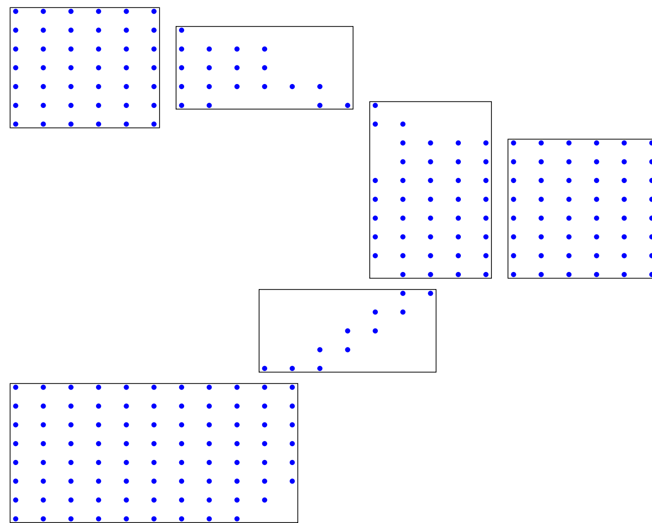


(b)

Figure 3: Test Case G2 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

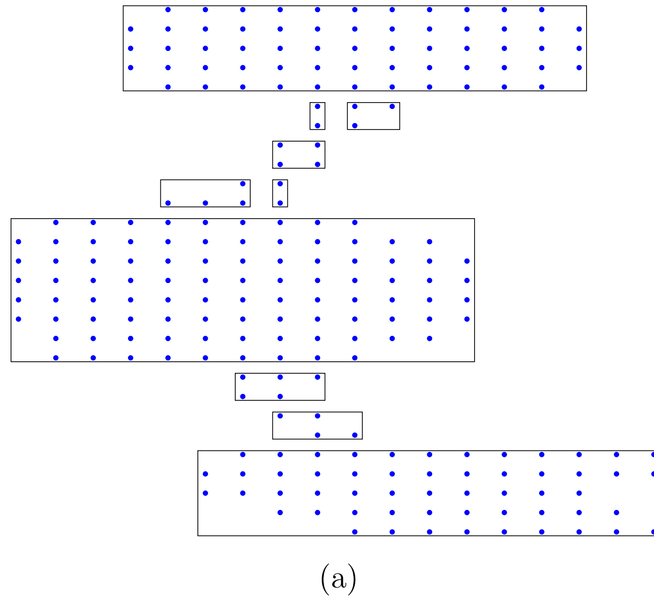


(a)

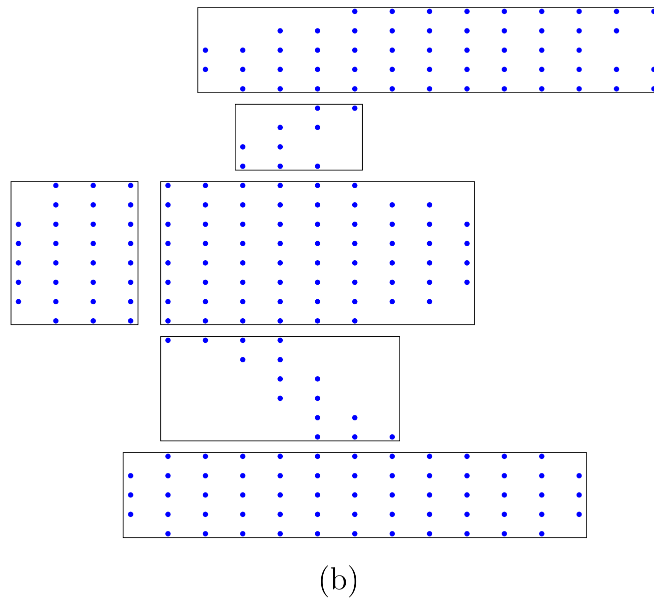


(b)

Figure 4: Test Case G4 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

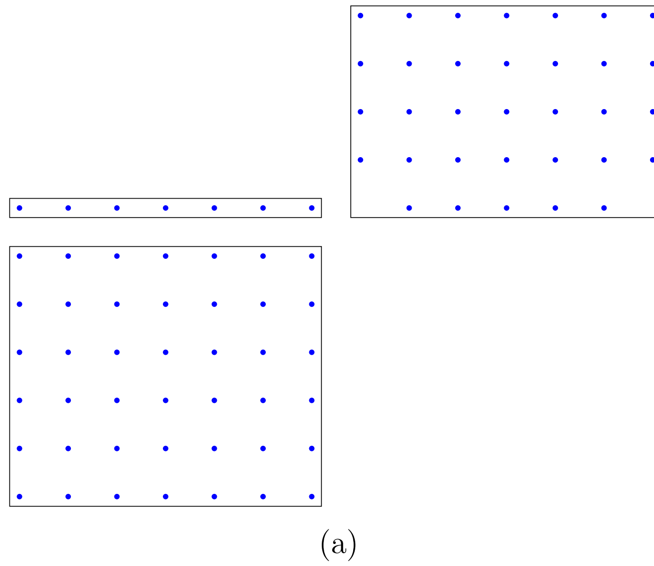


(a)

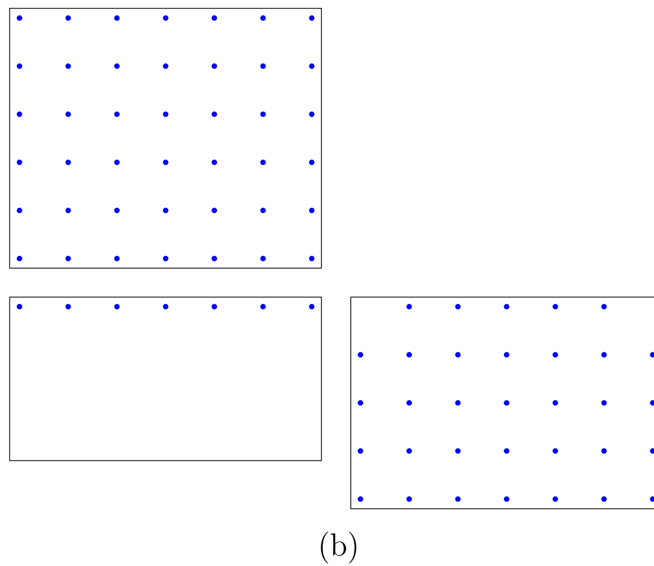


(b)

Figure 5: Test Case G5 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

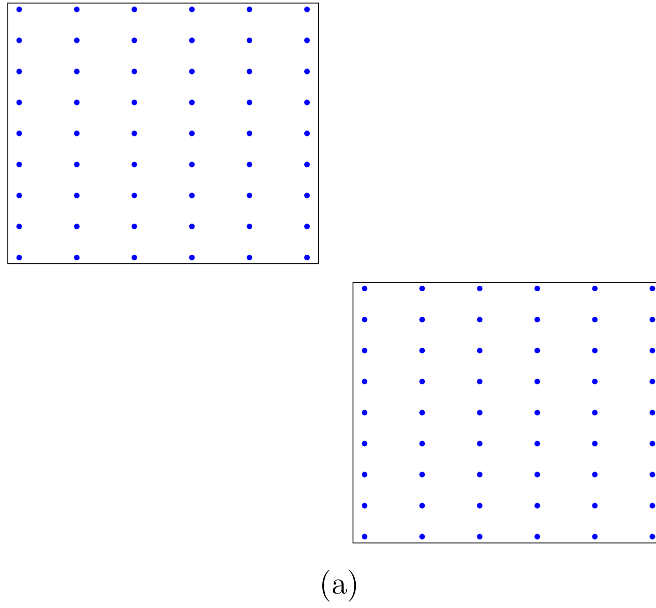


(a)

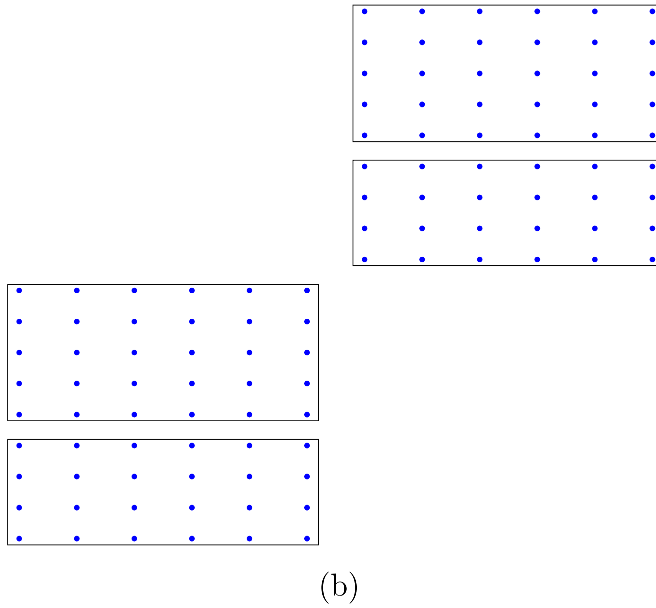


(b)

Figure 6: Test Case H1 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

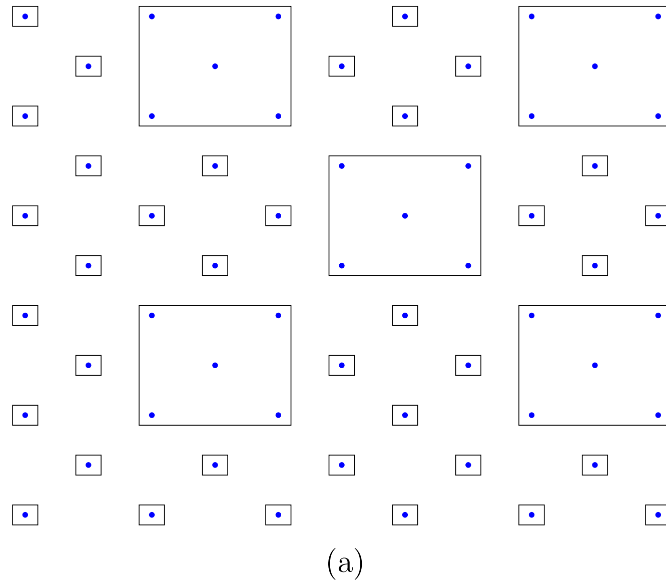


(a)

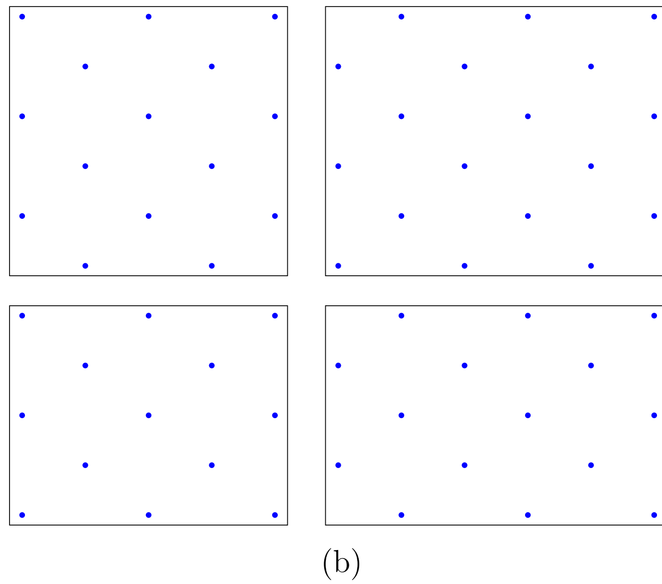


(b)

Figure 7: Test Case H2 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

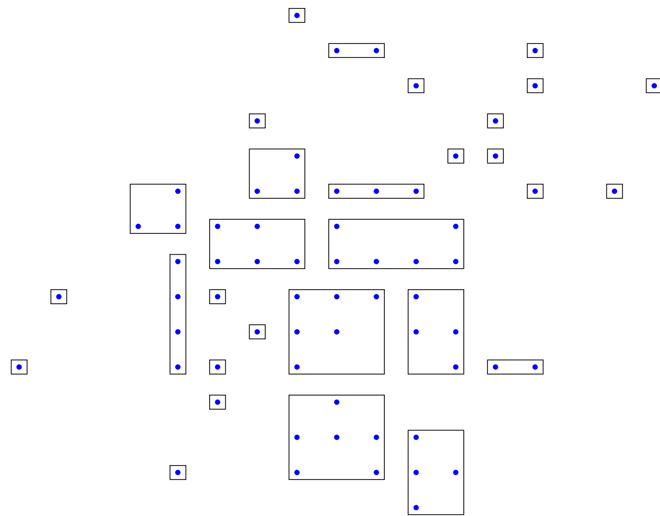


(a)

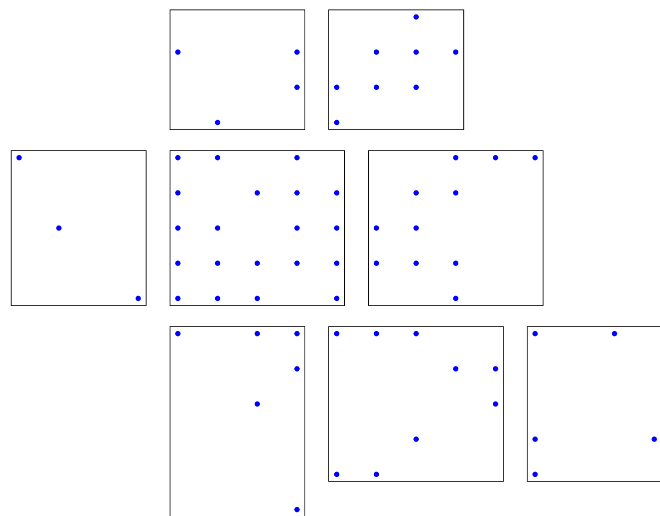


(b)

Figure 8: Test Case H3 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.



(a)



(b)

Figure 9: Test Case H4 results. Upper figure: the old algorithm covering. Lower figure: the new algorithm covering.

3.3 Discussion

There are a few important points to note in the comparison of the two algorithms:

- The minimum and maximum patch size constraints are satisfied for all cases (an evidence of the code's correctness).
- The efficiency threshold is almost always attained by the old algorithm; however, the new algorithm fails to meet this requirement. Especially hard cases are H3 and H4. This can be explained by the geometry of these two cases. in H3, we have points that are positioned in a way that the total area covered by them is twice their number. A related phenomenon occurs in H4: the randomly distributed points are scattered on a very large area, with some points isolated from the rest. While the old algorithm can afford to create small boxes around these isolated points, the new algorithm is required to create larger patches. Depending on the sparsity of the points, these larger patches can be very wasteful, as in H4. However, in cases where the points' "cloud" is more dense, as expected in the ICE application (and will anyway happen if we dilate to points to attain Objective 4 in the future), this is unlikely to happen. The other test cases provide some evidence for this prediction.
- The boxes tend to be more "cubical" (indicated by the higher average edge ratio) in the new algorithm. This an interesting consequence of adding the extra constraint on the minimum size of the patch (possibly also resulting from the maximum patch size, if a very big and stretched patch is initially pondered on). We come to the interesting observation that Objectives 2 and 3 do not contradict Objective 5, rather "live harmoniously". Note that more-cubical boxes tend to minimize mutual patch boundaries

4 Summary

We developed a new variant of the Berger-Rigoustos clustering algorithm, and compared it with the old Berger-Rigoustos used in Report 1, for several test cases. The new algorithm has a similar efficiency to the old algorithm, for most cases. Exceptions with low efficiency are observed when the flagged

cells cover a "checkerboard" space or are scattered at large distances from each other. On the other hand, the new algorithm satisfies additional constraints on the minimum and maximum patch size; interestingly, the added constraints (Objectives 2 and 3) help attain Objective 5 as well, because larger boxes than the old algorithm are likely to be generated, and because of the nature of the dissection, they tend to be more "cubical", hence minimize mutual patch boundaries.

It seems that the new algorithm satisfies Objectives 1,2,3 and 5 to a reasonable tolerance. Next, we will turn our attention in the next reports to dealing with Objective 4 - dilating the flagged cells to keep them away from the box boundaries. This is strongly related to the generation of multiple refinements within each other.

References

[BR91] M. J. Berger and I. Rigoustos. An algorithm for point clustering and grid generation. *IEEE. Trans. Sys. Man Cyber.*, 21 (5):1278–1286, 1991.

[JBW94] J. Saltzman J. Bell, M. J. Berger and M. Welcome. Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM. J. Sci. Comput.*, 15 (1):127–138, 1994.