SCI INSTITUTE

TECHNICAL REPORT

Efficient Update of Persistent Patches in the Berger Rigoutsous Algorithm

Oren E. Livne

UUSCI-2006-003

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

January 23, 2006

Abstract:

We consider the problem of how to update across timesteps the set of bounding boxes (patches) around cells marked for refinement. Our current algorithm does not re-grid (re-generate the entire set of boxes) at every timestep. Rather, it tries to add patches around cells added at the new timestep, keeping the rest of the boxes fixed, and possibly deleting unused boxes. If the update procedure fails, we re-grid. We study several illustrious test cases, focusing on the statistics of re-griding and box efficiency.

THE UNIVERSITY OF UTAH

Efficient Update of Persistent Patches in the Berger Rigoutsous Algorithm

Oren E. Livne *

January 23, 2006

Abstract

We consider the problem of how to update across timesteps the set of bounding boxes (patches) around cells marked for refinement. Our current algorithm does not re-grid (re-generate the entire set of boxes) at every timestep. Rather, it tries to add patches around cells added at the new timestep, keeping the rest of the boxes fixed, and possibly deleting unused boxes. If the update procedure fails, we re-grid. We study several illustrious test cases, focusing on the statistics of re-griding and box efficiency.

Key words. Updating clusters, time-stepping, cluster efficiency, re-griding, persistent refinement patches.

1 Introduction

In Report 2 we showed how to obtain a set of bounding boxes (patches), given a set of flagged cells ("cells that need further refinement"). This happens at every time step of the ICE code; however, we would ideally not like to re-generate these boxes at every time step. Rather, we would like to re-grid only if the geometry of the flagged cells significantly changed across timesteps;

*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`

else, we would want to use information from the previous time step to *update* the set of patches.

Moreover, we would like to keep as many "old patches" as possible intact. The reason is that the computational bottleneck in updating patches is the data transfer across processors, when a new patch is created. Hence, we would like to minimize the number of patches created at every time step, and prolong the use of existing patches for as many timesteps as possible. The clustering algorithm itself is not expected to be a bottleneck, hence we can afford to spend more time finding a good patch configuration.

Our updating strategy is as follows: we employ two routines, one that creates patches and one that updates them. At the initial time, we create a patch set. At each subsequent timestep, we first try to update, and if we fail, we re-grid (create a new set of patches). To the best of our knowledge, current patch generation algorithms re-grid at every time step, thus our novel approach may as efficient or better.

In §2 we discuss our goals for the time-dependent clustering algorithm. In §3 we list the measures by which we measure the algorithm's result. In §3.1 we describe the updating strategy and parameters. In §4, we study the algorithm's efficiency and updating efficiency statistics (e.g., how many timesteps can use the same patches without re-griding). We summarize our findings and discuss future work in §5.

2 Goals for Time-Stepping

In addition to Objectives 1–5 that we set for a box covering at a single timestep (Reports 1–2), we would like to have the following in time-dependent problems:

1. *Objective 6 - Fast update:* if the flagged cells describe a moving shock front, we would like to use the box covering from the previous timestep, and make minor modifications to it to fit it to the new timestep's flagged cells.

2. *Objective 7 - Persistent Patches:* a patch should be used as long as possible. That is, we would like to keep old patches instead of deleting or even modifying them. This is because the data flow associated in creating or modifying patches is the expected computational bottleneck of the adaptive mesh part of ICE.

We present an algorithm that addresses Objectives 1–7 (again, we do not treat Objective 4, assuming that a prior dilation has been already performed). As we will see, we cannot expect an efficiency (Objective 1) as high as for a single timestep clustering problem. However, the gain in Objectives 6 and 7 overshadows this degradation.

3 Indicators

We assess the quality of the time-dependent constructed set of boxes by the following measures. The measures are naturally related to the objectives listed in §2.

1. Efficiency versus time: ratio of the number of flagged cells to the total box area [dimensionless] (Objective 1).
2. Number of boxes [dimensionless] versus time.
3. Average box volume [cell] versus time (related to Objective 2 and 3 and to load balancing).
4. Average ratio of the shorter to the longer side ratio [dimensionless], versus time. As we have seen, this relates to the total mutual boundary length among patches (Objective 5).

In addition, we define the following quantities that are averages over all timesteps:

1. Averages number of timesteps between re-griding (i.e., the number of times we needed to create a new set of boxes, divided by the number of timesteps - Objective 6).
2. Average number of timesteps for which a patch exists (i.e., the average "life span" of a patch - Objective 7).

3.1 Updating Strategy

The core of the updating strategy is described in the following pseudo-code. We use two functions: `create-cluster` and `update-cluster`. At the initial timestep, we generate a set of boxes with `update-cluster`. Then, we loop over timesteps, and for each new set of points, we first try to update the set from the previous timestep, using `update-cluster`. If this fails, we re-grid using `create-cluster`. The array `points` denotes the list of flagged cells at any given timestep. For simplicity, we assume the timestep is uniform in the code below.

```
t   = 0;
box = create-cluster(points);

for count = 1:num-timesteps
    t           = t+delta-t;
    fprintf('time step = %f\n',t);
    points      = flagged-cells-in-old-timestep;
    points-new  = flagged-cells-in-new-timestep;
    box-new     = update-cluster(points,points-new);

    if (update-succeeded)
        box     = box-new;
        regrid  = 0;
    else
        box     = create-cluster(points-new);
    end if

    Accumulate-statistics;
end for
```

3.2 The Update Function

The update function assumes three inputs: the old set of points `points`, the old set of boxes `box`, and a set of points to be added to the old set, `points-new`. Note that we do not include the points to be removed from the old set. Before activating the update function, we can delete from `points` the set `points-old - points`, so that `points` is contained in `points-old`; if some boxes in `box` become empty, we delete them.

Next, we create a set of boxes around the new points, `box-new`. If it does not overlap `box`, we are done, and output the union of the two sets as the final set of boxes. If not, we loop over new boxes, and try to shift each of them to eliminate overlap with the old and other new boxes, while covering all the new points that were covered by that new box before shifting it. We succeed in updating the old set if we can find a proper shift for all new boxes; if we fail to find a shift for one box, we declare failure.

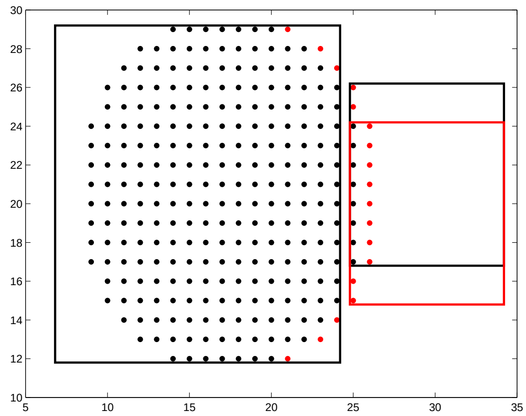
Note that the permitted shifts $s = (s_x, s_y)$ of a new box are limited to those for which the shifted box would still contain the *tight bounding box around the flagged cells in the original new box.* It is easy to see that s belongs to a box, $[SX_{left}, SX_{right}] \times [SY_{left}, SY_{right}]$, that is determined by the sizes of the new box and the tight bounding box. When we look for a "good shift", we search these permitted s's by ascending order of magnitude, that is, by ascending $s_x^2 + s_y^2$. This is done to minimize search time, hoping that most of the shifts of new boxes will be of only few cells in every direction. Naturally, $s = (0, 0)$ is searched first, so if the original new box does not overlap any other box, it is readily accepted to the final set of boxes. See Fig. 1 for an illustration.

Finding a shift can be regarded as a local optimization process: applying a transformation to one new box, keeping the rest fixed. When we proceed to optimize a second, we use the updated location of the first box; and so on. This resembles Gauss-Seidel relaxation (or point-by-point optimization) described in [Bra84, Chap. 1] in the context of multigrid methods for solution of PDEs, and in[BR03] , for optimization. We could go to more sophisticated local (e.g., try to stretch the box as well) or even multi-scale optimization; but that would (for instance) tamper with the minimum and maximum box size requirements, thus we decided to stay with the simple variant of shift only.

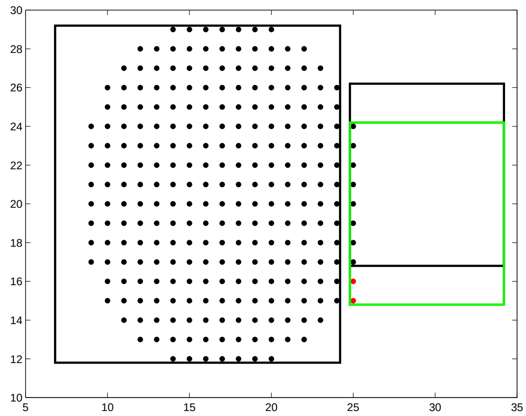
3.3 Full MATLAB Code

In this section we include two routines: `test-movement` and `update-cluster`. The first is a driver that loops over timesteps and moves a certain object in a certain direction inside the domain of the coarse grid in ICE. In this case, we horizontally move a sphere across the domain. The second routine is the update code, that takes a current set of boxes and points to be added to the old set of flagged points. If it succeeds, it returns the status code $status = 0$ with the updated set of boxes. If it fails, $status = -1$, and we are required to run `create-cluster`.

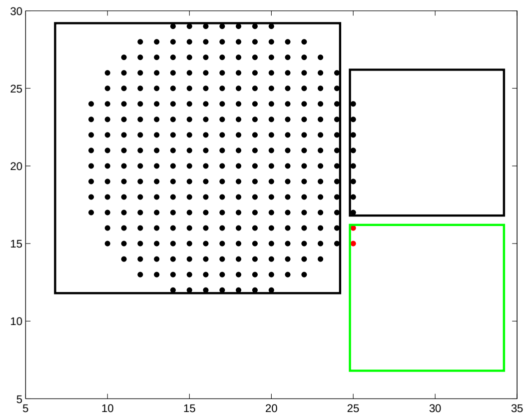
```
%function test_movement
% TEST_MOVEMENT
% A driver for CREATE_CLUSTER and UPDATE_CLUSTER functions, that takes a shape of flagged cells (e.g.,
% a circle), and moves it over the domain as in "time-stepping". For each timestep, we generate the
% boxes around the flagged cells by CREATE_CLUSTER, or only update them with UPDATE_CLUSTER.
% Statistics on how many re-boxing are actually needed, etc., are printed at the end of the run.
```



(a)



(b)



(c)

Figure 1: An example of shifting a new box in the update function. (a) The relevant old flagged cells are the black points, the old set of boxes is denoted by the black rectangles; the new points are in red, and the new set of boxes around the points that lie outside the black boxes, is denoted by the red rectangles (here there is only one red rectangle). (b) We pick a new box (denoted by green), and try to shift it so that it does not overlap any other box. (c) Found a good shift: $s = (0, -8)$.

```
%
% Author: Oren Livne
% Date  : 05/13/2004    Version 1: moving a circle in a horizontal direction

%%%%% Initialize parameters of this run
fprintf('<<<<<<<<<<<<<< TEST_MOVEMENT: an example of moving points and boxes around them >>>>>>>>>>>>>>\n');
plot_flag       = 1;                                        % Flag for generating plots of the boxes and cells
domain_size     = [100 80];                                 % Size of the entire domain we live in [cells]
num_tsteps      = 20;                                       % Number of timesteps to be performed
delay           = 0.;
center          = [15 20];
direction       = [1 1];                                    % Direction of movement
s               = numgrid('D',30);                          % Example: the unit disk is flagged
max_regridding  = 10000;                                    % Maximum number of timesteps allowed between re-gridding

%%%%% Parameters for cluster creation
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   = 10;                                       % Minimum box side length allowed (in all directions)
opts.max_volume = 400;                                      % Maximum box volume allowed. Need to be >= (2*min_side)^dim.
opts.print      = 0;                                        % Printouts flag
opts.plot       = 0;                                        % Plots flag
opts_create     = opts;

%%%%% Parameters for cluster update (efficiency, etc. relate to the generation of the new boxes before merging them into the old set
opts            = opts_create;
opts.print      = 0;                                        % Printouts flag7
opts.plot       = 0;                                        % Plots flag
opts_update     = opts;

%%%%% Initial time step: create the initial boxes
dim             = length(domain_size);                      % Dimension of the problem
points_old      = zeros(domain_size);                       % The array of flagged cells
a               = cell(dim,1);                              % Find the x and y coordinate (a{1} and a{2}, respectively)
[a{:}]          = find(s);                                  % Read the x- and y- from the 2D array s (which has non-zeros integers at the flagged cel
a{1}            = a{1} - size(s,1)/2 + center(1);           % Starting position: move all flagged cells' indices to the right in the x-direction
a{2}            = a{2} - size(s,2)/2 + center(2);           % Starting position: put in the middle of the domain in y, because we don't yet support w
ind             = sub2ind(domain_size,a{:});                % Convert to a 1D array
points_old(ind) = 1;                                        % Put 1's at the flagged cells
[rect,stats]    = create_cluster(points_old,opts_create);   % Create the initial set of boxes (t=0)
count           = 1;                                        % Counter of number of timesteps
t               = 0;                                        % Initial time
dt              = 1;                                        % Delta_t
all_rect        = rect;                                     % Accumulates boxes from all times
all_stats       = stats;                                    % Accumulates statistics from all times
all_stats.t     = t;                                        % Save initial time
all_stats.regrid = 1;                                       % Save initial regridding status

%%%%% Plot-outs of the points and current boxes
fprintf('Initial time = %d\n',t);
if (plot_flag)
    figure(1);
    clf;
    plot_points(points_old,'red');
    hold on;
    plot_boxes(rect,'black');
    axis equal;
    axis([0 domain_size(1) 0 domain_size(2)]);
    %    pause
    shg;
    pause(delay);
    eval(sprintf('print -depsc t%d.eps',t));
end

%%%%% Main loop over time steps
for count = [1:num_tsteps]+1,
    t           = t+dt;
    fprintf('time step = %f\n',t);
    a{1}        = a{1}+direction(1);                        % Move all flagged cells' indices in the x-direction
    a{2}        = a{2}+direction(2);                        % Move all flagged cells' indices in the y-direction
    stats_old   = stats;
    ind_old     = find(points_old);                         % Old flagged cells in a 1D array
    ind_new     = sub2ind(domain_size,a{:});                % New flagged cells in a 1D array
    points_new  = zeros(size(points_old));                  % An array for the cells added in this time step
    points_new(setdiff(ind_new,ind_old)) = 1;               % The additional cells of the new time step in a 2D array
    points_old(setdiff(ind_old,ind_new)) = 0;               % The old cells that are deleted in the new time step
    [rect_new,status,stats] = update_cluster(...
```

```
        points_old,rect,points_new,opts_update);                % Try to add the new points using the update function
    points_old  = points_old + points_new;                      % Put the additional cells in the old array, so now points_old are all the points in the
    if ((status >= 0) & (mod(count,max_regridding) ~= 0))       % Update suceeded and we are not forcing a re-gridding
        rect    = rect_new;                                     % Update box set
        regrid  = 0;
    else                                                        % Update failed, re-grid
        fprintf('Re-gridding\n');
        [rect,stats] = create_cluster(points_old,opts_create);  % Create the set of boxes from the current points
        regrid  = 1;
    end

    %%%%% Accumulate statistics
    all_rect                = [all_rect; rect];
    all_stats.t             = [all_stats.t; t];
    all_stats.regrid        = [all_stats.regrid regrid];
    all_stats.efficiency    = [all_stats.efficiency; stats.efficiency];
    all_stats.num_boxes     = [all_stats.num_boxes ; stats.num_boxes ];
    all_stats.avg_volume    = [all_stats.avg_volume; stats.avg_volume];
    all_stats.avg_side_rat  = [all_stats.avg_side_rat; stats.avg_side_rat];

    %%%%% Plot-outs of the points and current boxes
    if (plot_flag)
        figure(1);
        clf;
        plot_points(points_old,'red');
        hold on;
        plot_boxes(rect,'red');
        axis equal;
        axis([0 domain_size(1) 0 domain_size(2)]);
        %    pause
        shg;
        pause(delay);
        if ((t == 19) | (t == 20) | (t == 21))
            eval(sprintf('print -depsc t%d.eps',t));
        end
    end
end

fig = 1;

fig = fig+1;
figure(fig);
clf;
plot(all_stats.t,all_stats.efficiency);
xlabel('time [sec]');
ylabel('Efficiency [%]');
print -depsc teff.eps

fig = fig+1;
figure(fig);
clf;
plot(all_stats.t,all_stats.num_boxes);
xlabel('time [sec]');
ylabel('Number of boxes');
print -depsc tnbox.eps

fig = fig+1;
figure(fig);
clf;
plot(all_stats.t,all_stats.avg_volume);
xlabel('time [sec]');
ylabel('Average Box Volume [cell^d]');
print -depsc tavgvol.eps

a           = sortrows(all_rect,[1:2*dim]);
base        = max(a(:))+1;
hash        = sum(a.*repmat(base.^[0:size(a,2)-1],size(a,1),1),2);
cross       = find(diff(hash) ~= 0);
histogram   = [cross(1) ; diff(cross) ; length(hash)-cross(length(cross))];
ind         = [cross ; length(hash)];

all_stats.avg_box_life  = mean(histogram);
all_stats.avg_regrid    = (length(find(all_stats.regrid))-1)/num_tsteps;        % -1 because we can ignore the initial time
fprintf('Average efficiency                    : %.1f%%\n',mean(all_stats.efficiency));
fprintf('Average #timesteps of a box''s life    : %f\n',all_stats.avg_box_life);
fprintf('Average #timesteps between regriddings: %f\n',1/all_stats.avg_regrid);
```

```
function [rect,status,stats] = update_cluster(points_old,rect_old,points_new,opts);
% UPDATE_CLUSTER
% Updating a set of covering boxes. If points_old is the old set of flagged cells, and rect_old
% is the covering box set, we add some new flagged cells points_new, and try to update the covering
% set. Possible status code:
% status = 0      We succeeded to update, and the new box set is rect.
% status = -1     We failed tp update. Run CREATE_CLUSTER again.
%
% Author: Oren Livne
% Date  : 05/12/2004    Version 1: move each single new box at a time to eliminate overlap with old+new
%         05/13/2004    Version 2: search in a spiral for a new shift (Dave's brilliant idea)

%%%%%%%%%% Set and print parameters
if (nargin < 4)                                             % Default parameters
    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   = 10;                                   % Minimum box side length allowed (in all directions)
    opts.max_volume = 100;                                  % Maximum box volume allowed
    opts.print      = 0;                                    % Printouts flag
    opts.plot       = 0;                                    % Plots flag
end
fprintf('<<<<<<<<<<<<<< UPDATE_CLUSTER: updating a cluster given more flagged points >>>>>>>>>>>>>>\n');
fprintf('Parameters for creating the new cluster:\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

%%%%% Delete empty boxes from the old box set, in case we deleted points from points_old and some of the boxes are unusable (but might interfer with shif
efficiency      = box_efficiency(points_old,rect_old);          % Compute efficiencies of boxes
empty           = find(efficiency < 1e-13);                     % Empty boxes (efficiency = 0)
rect_old(empty,:) = [];                                         % Remove empty boxes
if ((opts.print) & (~isempty(empty)))                           % Print the boxes that were deleted
    fprintf('Empty boxes deleted:');
    fprintf('%d ',empty);
    fprintf('\n');
end

%%%%%%%%%% Initialize, find the initial new covering, get rid of non-relevant new points
dim         = length(size(points_old));                         % Dimension of the problem
status      = 0;                                                % Default status is success, if we fail we break and status=-1
points      = points_old + points_new;                          % Merge the old and new point sets (for binary images, + = union)
points_new_orig = points_new;
for k = 1:size(rect_old,1)                                      % Get rid of new points that are covered by old boxes
    r = rect_old(k,:);                                          % Old box no. k
    points_new(r(1):r(3),r(2):r(4)) = 0;                        % Delete new points in this box
end
[i,j]       = find(points_new);                                 % Index arrays [i,j] of flagged cells
if (isempty(i))                                                 % No new points are flagged
    rect    = rect_old;                                         % Return the old box set
    status  = 0;                                                % Return a success code
    stats   = final_stats(points,rect,opts);                    % Final printouts (overall statistics) and plot-outs
    fprintf('<<<<<<<<<<<<<< END UPDATE_CLUSTER >>>>>>>>>>>>>>\n');
    return;
end
opts_create = opts;                                             % Options for CREATE_CLUSTER function
opts_create.print = 0;                                          % No printouts from CREATE_CLUSTER, please!
opts_create.plot = 0;                                           % No plots from CREATE_CLUSTER, please!
rect_new    = create_cluster(points_new,opts);                  % Create a set over the new points, might overlap old set
num_new     = size(rect_new,1);                                 % Number of new rectangles
%opts.print  = 1;
%opts.plot   = 1;

%%%%% Plot-outs: plot the points and the current boxes. The considered box is in red.
if (opts.plot)
    figure(1);
    clf;
    plot_points(points_new_orig,'red');
    hold on;
    plot_points(points_old,'black');
    plot_boxes(rect_old,'black');
    plot_boxes(rect_new,'red');
    pause
end

%%%%%%%%%% Loop over new boxes; try to move each one so it does not overlap any old or new one

k       = 1;                                                    % Index of new box to be processed
```

```
while (k <= num_new)                                       % Do until all boxes have been processed
    r               = rect_new(k,:);                       % Box coordinates
    s               = points_new(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
    [i,j]           = find(s);
    tight           = [min(i) min(j) max(i) max(j)] + [r(1) r(2) r(1) r(2)] - 1;
    if (opts.print)
        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)
        fprintf('Tight box = [%3d,%3d,%3d,%3d]\n',tight);
    end

    %%%%% Plot-outs: plot the points and the current boxes. Old is black, new is red. The considered box is in green.
    if (opts.plot)
        figure(1);
        clf;
        plot_points(points_new,'red');
        hold on;
        plot_points(points_old,'black');
        plot_boxes(rect_old,'black');
        plot_boxes(rect_new,'red');
        plot_boxes(rect_new(k,:),'green');
        pause
    end

    %%%%% Initialize parameters for shift loop
    other_rect  = [rect_old; rect_new(setdiff(1:num_new,k),:)]; % All the other rectangles - old+new
    overlap     = 1;                                       % In the loop below: 0 if we overlap no other rectangle, 1 if we do
    within_range= 0;                                       % In the loop below: Check flag - if t contains the tight box around flagged cells (<==>
    t           = r;                                       % t is the new (shifted) box; init t to the old box r
    action      = 0;                                       % Counter of shifts, action=1 is original box (shift=(0,...,0)).

    %%%%% Prepare a list of permitted shifts of the new box under consideration
    shift_line  = cell(dim,1);                             % Permitted shift ranges along the different dimensions
    for d = 1:dim
        shift_line{d} = [tight(d+dim)-t(d+dim):t(d)-tight(d)];  % Permitted shift ranges along dimension d, so that t still contains tight
    end
    [shift_cell{1:dim}] = ndgrid(shift_line{:});           % Prepare a dim-D list of the possible shifts
    shift = [];                                            % 1D list of dim-coordinates of the shifts
    for d = 1:dim                                          % Loop over dimensions
        shift = [shift shift_cell{d}(:)];                  % Concatenate the d-coordinate of all shifts to the big list
    end
    dist        = sum(shift.^2,2);                         % Distance from original rectangle
    [temp,ind]  = sort(dist);                              % Sort by ascending distance
    shift       = shift(ind,:);                            % Apply permutation to the shift list

    %%%%% Main loop over shifts: trying to find a shift for which there's no overlap and we still cover the new points
    for action = 1:size(shift,1),                          % Loop over all permitted shifts
        if ((~overlap) & (within_range))                   % If there's no overlap, accept this rectangle, otherwise, try other actions
            break;
        end
        s       = shift(action,:);                         % Current shift
        if (opts.print)
            fprintf('Action %5d: shift = (%d,%d)',action,s);
        end
        within_range    = box_subset(t,tight);             % Check if t contains the tight box around flagged cells (<==> t contains all flagged ce
        t(1:dim)        = r(1:dim) + s;                    % Shift box lower-left coordinate by s
        t(dim+[1:dim])  = r(dim+[1:dim]) + s;              % Shift box upper-right coordinate by s
        within_range    = box_subset(t,tight);             % Check if we still cover what we should cover
        overlap         = max(box_intersect(other_rect,t));     % 0 if we overlap no other rectangle, 1 if we do
        if (opts.print)
            fprintf('   overlap = %d\n',overlap);
        end
        if (opts.plot)
            figure(1);
            clf;
            plot_points(points_new,'red');
            hold on;
            plot_points(points_old,'black');
            plot_boxes(rect_old,'black');
            plot_boxes(rect_new(setdiff(1:num_new,k),:),'red');
            plot_boxes(t,'green');
            pause
        end
    end
    if (overlap)                                           % Unfortunately we can't find a good shift, so give up and do re-gridding with CREATE_CLU
```

```
        if (opts.print)
            fprintf('No shift found, giving up\n');
        end
        rect    = [];                                           % Return an empty results
        status  = -1;                                           % Negative status = UPDATE_CLUSTER failed
        stats   = [];
        fprintf('<<<<<<<<<<<<<< END UPDATE_CLUSTER >>>>>>>>>>>>>>\n');
        return;
    else
        rect_new(k,:) = t;                                      % Replace the kth new rectangle with the shifted one, t
    end
    k = k+1;                                                    % Consider the next new rectangle
    if (opts.print)
        fprintf('\n');
    end
end

%%%%% Unify old and new sets, and delete empty boxes
rect            = [rect_old; rect_new];                         % Merge the old and new box sets
efficiency      = box_efficiency(points,rect);                  % Compute efficiencies of boxes
empty           = find(efficiency < 1e-13);                     % Empty boxes (efficiency = 0)
rect(empty,:)   = [];                                           % Remove empty boxes
if ((opts.print) & (~isempty(empty)))                           % Print the boxes that were deleted
    fprintf('Empty boxes deleted:');
    fprintf('%d ',empty);
    fprintf('\n');
end

stats = final_stats(points,rect,opts);                          % Final printouts (overall statistics) and plot-outs
fprintf('<<<<<<<<<<<<<< END UPDATE_CLUSTER >>>>>>>>>>>>>>\n');
```

4 Numerical Experiments

Each test case describes a movement of a set of points as in "time-stepping". We performed 30 timesteps. The maximum box volume was set to 400, and the minimum side length to 10 (Note that in general, it is prudent to have $max_volume \geq (2min_side_length)^d$, where d is the dimension of the problem. Otherwise, big boxes cannot be dissected as the resulting boxes are too small, yielding a contradiction). For each case, we show several snapshots at some timesteps, and compare (in the summarizing statistics tables) the results when re-griding is forced at every timestep ($regrid = 1$), versus a full utilization of the update routine ($regrid = \infty$). Unless otherwise noted, figures describe the tests where full update was operational ($regrid = 1$).

4.1 Sphere Moving in Positive-x

This test case is denoted "Sphere-x". We start with a sphere near the left x-boundary of the domain (and in the middle in y), and move it one cell to the right in the x-direction. In this case, we need no re-griding: we can create the set of boxes at the initial time, and only update it at all subsequent timesteps.

4.2 Sphere Moving in Negative-x

This test case is denoted "Sphere-mx". It is identical to Sphere-x, except that we now start with the sphere near the right x-boundary of the domain (and in the middle in y), and move it one cell to the left in the x-direction. By symmetry, we need no re-griding here too.

4.3 Sphere Moving in Positive-y

This test case is denoted "Sphere-y". It is identical to Sphere-x, with the x and y directions roles reversed. By symmetry, we need no re-griding here too.

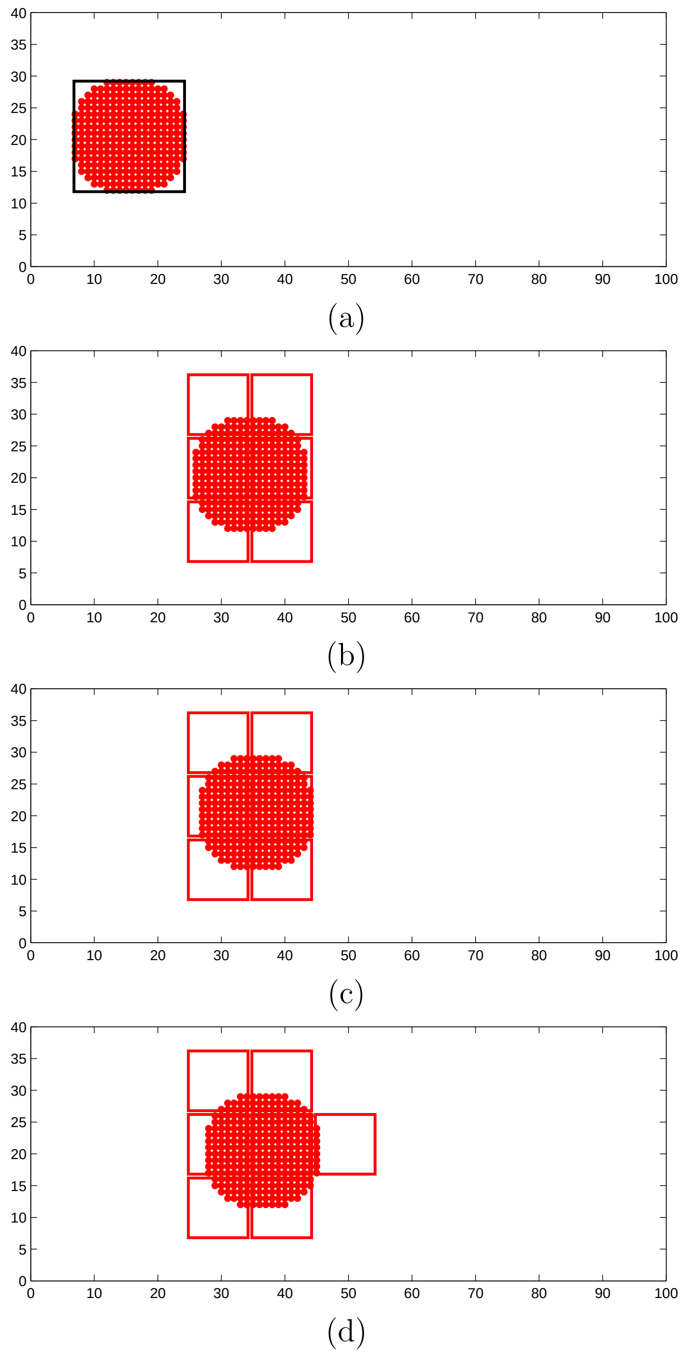
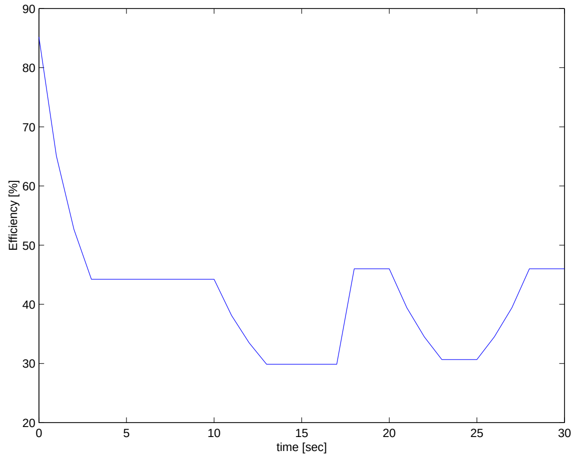
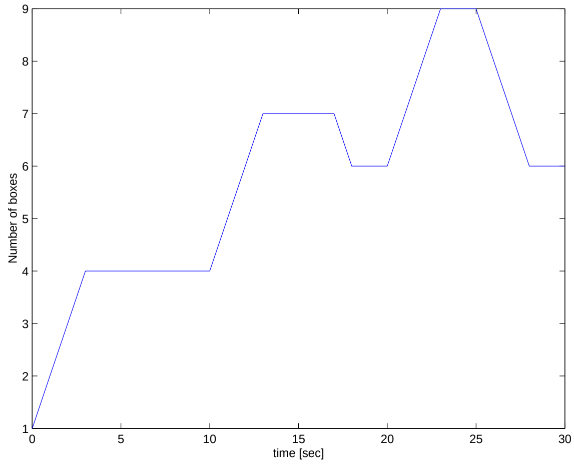


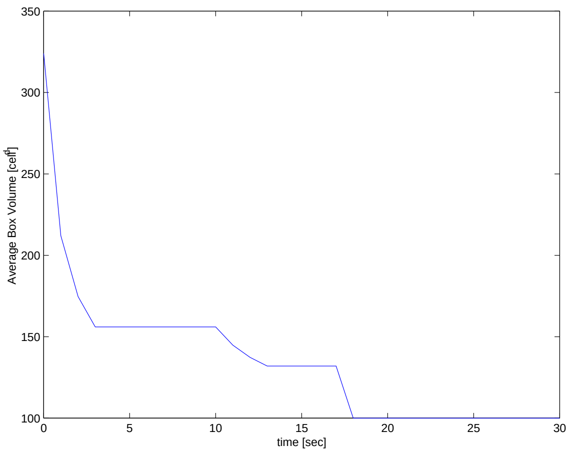
Figure 2: Sphere-x test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) Initial timestep. (b) $t = 19$. (c) $t = 20$. (d) $t = 21$.



(a)

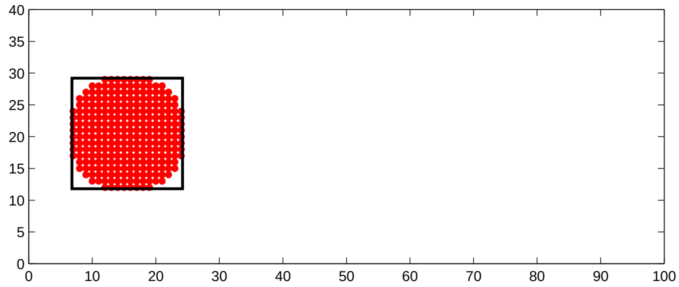


(b)

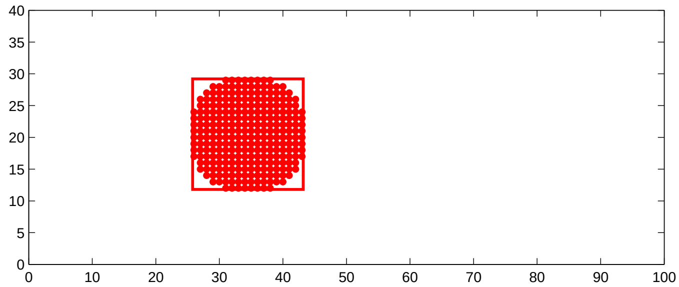


(c)

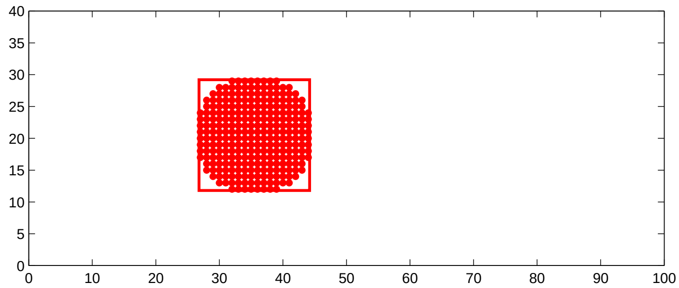
Figure 3: Sphere-x test case: statistics. (a) Efficiency versus time. (b) Number of boxes versus time. (c) Average box volume versus time.



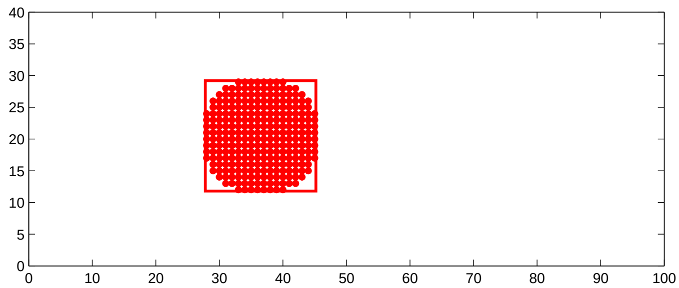
(a)



(b)

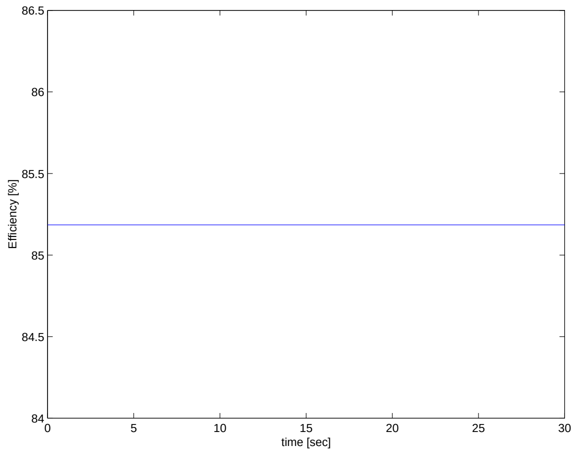


(c)

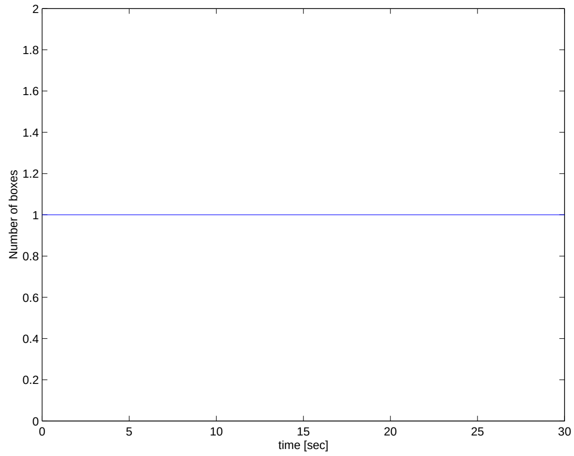


(d)

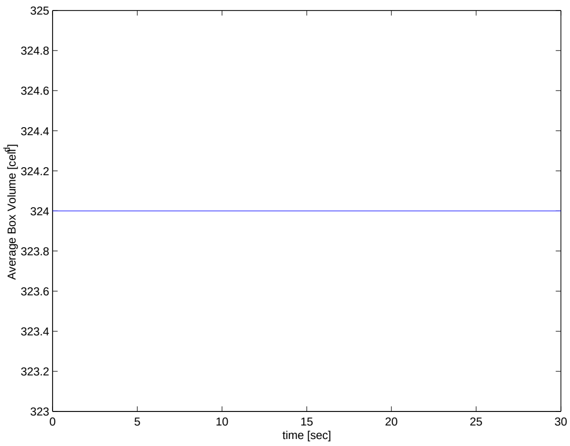
Figure 4: Sphere-x test case with re-griding at every timestep: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) Initial timestep. (b) $t = 19$. (c) $t = 20$. (d) $t = 21$.



(a)

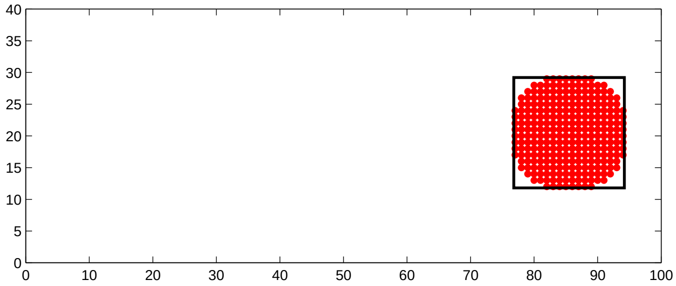


(b)

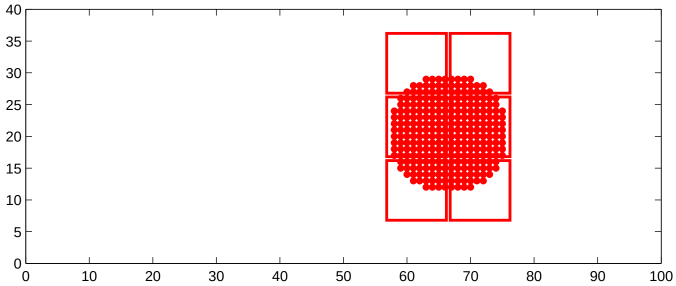


(c)

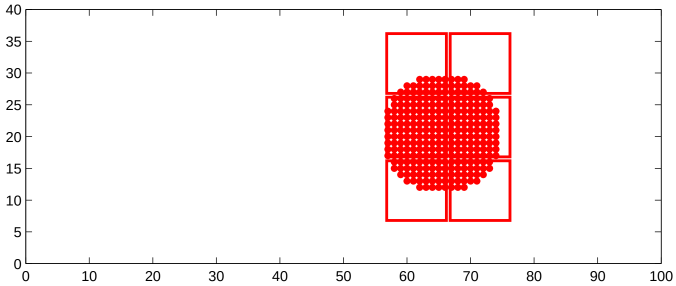
Figure 5: Sphere-x test case with re-griding at every timestep: statistics. (a) Efficiency versus time. (b) Number of boxes versus time. (c) Average box volume versus time.



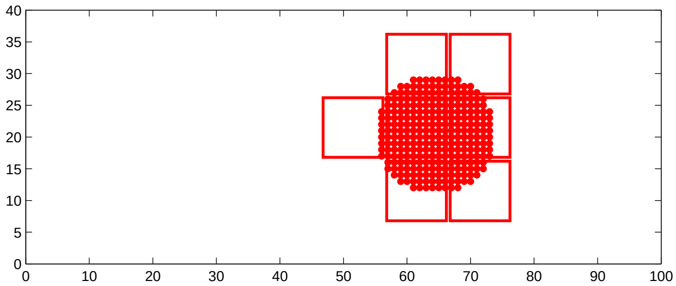
(a)



(b)

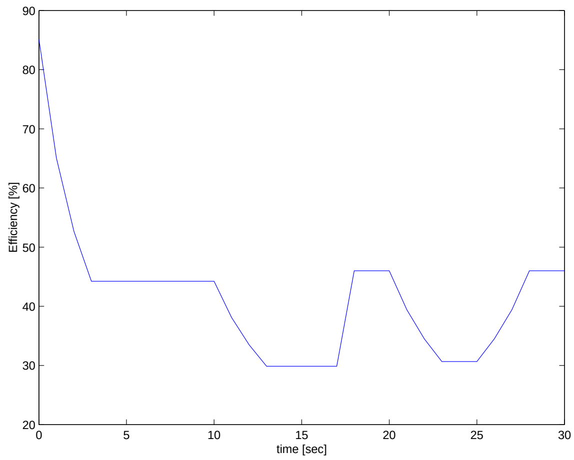


(c)

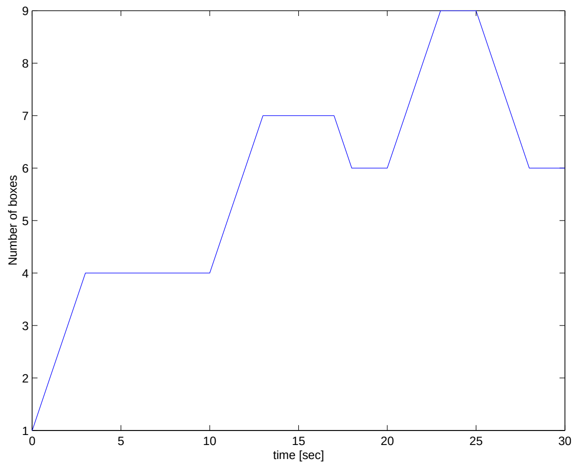


(d)

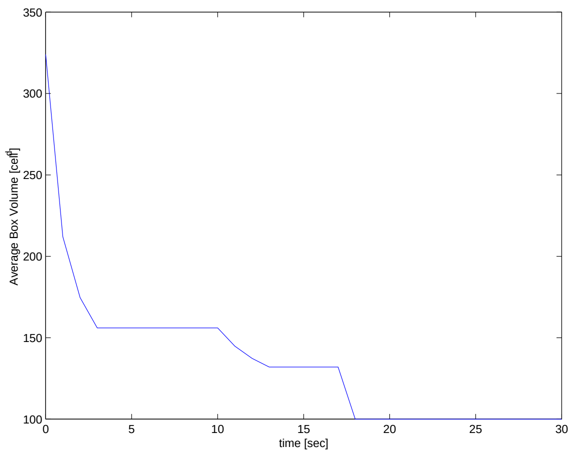
Figure 6: Sphere-mx test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) Initial timestep. (b) $t = 19$. (c) $t = 20$. (d) $t = 21$.



(a)

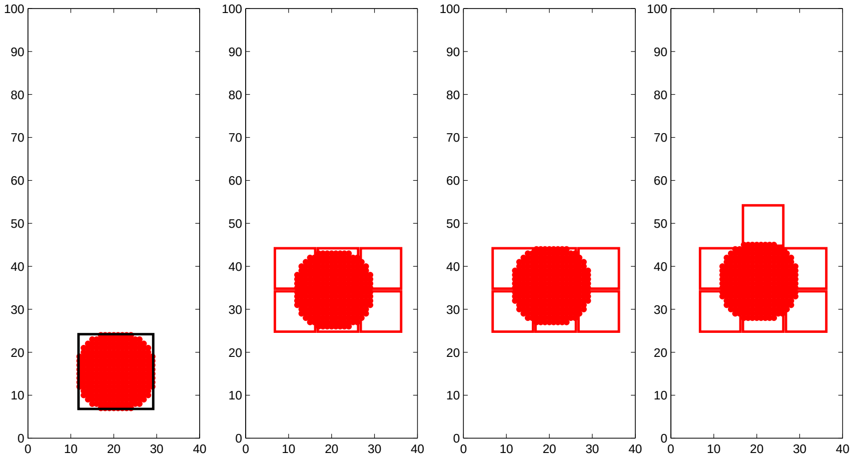


(b)



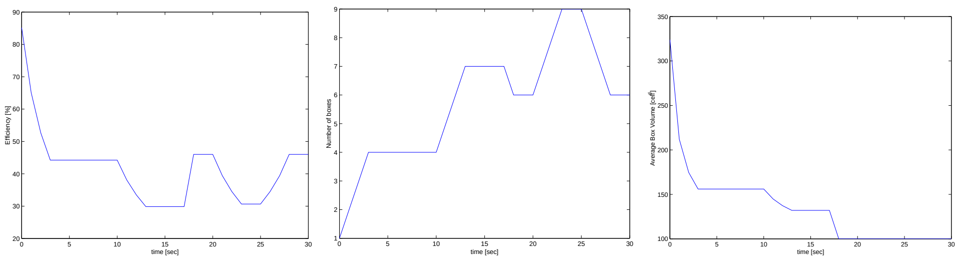
(c)

Figure 7: Sphere-mx test case: statistics. (a) Efficiency versus time. (b) Number of boxes versus time. (c) Average box volume versus time.



(a) (b) (c) (d)

Figure 8: Sphere-y test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) Initial timestep. (b) $t = 19$. (c) $t = 20$. (d) $t = 21$.



(a) (b) (c)

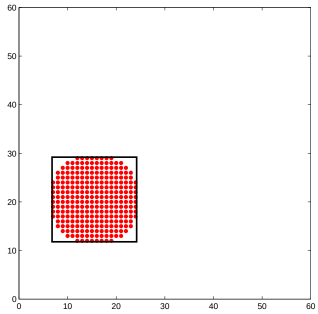
Figure 9: Sphere-y test case: statistics. (a) Efficiency versus time. (b) Number of boxes versus time. (c) Average box volume versus time.

4.4 Sphere Moving in Diagonal

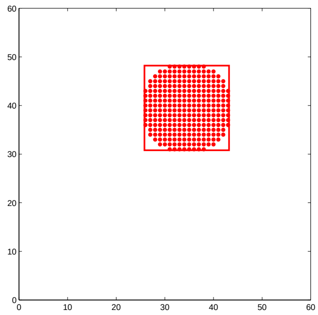
This test case is denoted "Sphere-diag". This time, we start at the left bottom part of the domain, and move the sphere in a diagonal direction (one cell in x and one cell in y at every timestep). This is the opposite extreme to the horizontal movement case: because we align our patches with the grid lines, a diagonal movement requires re-griding at every time step. This also relates to the size of the sphere relative to the minimum box size: when the sphere is large compared with the minimum side length, we might be able to use the update routine more frequently than in the following example.

4.5 Sphere in a Circular Motion

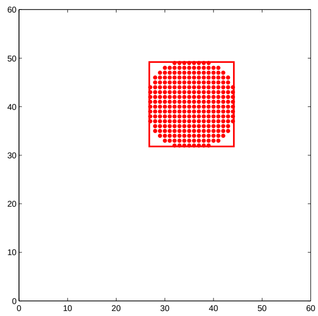
This test case is denoted "Sphere-circ". We specify a certain circle in the domain (in this example, centered at $(100, 100)$ with radius $R = 30$), and move the sphere's center along this circle. At every time step, we update the angle θ of the sphere's center with respect to the circle's center by an increment that results in about one Cartesian cell shift (in x and y combined) in the sphere's location. For instance, we used $\Delta\theta = 0.1 \arccos(1/R) \approx .152$ radians. This is also a hard case, and re-griding is needed every ≈ 1.5 steps.



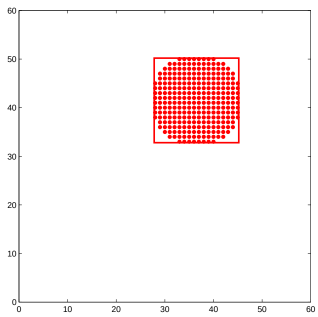
(a)



(b)

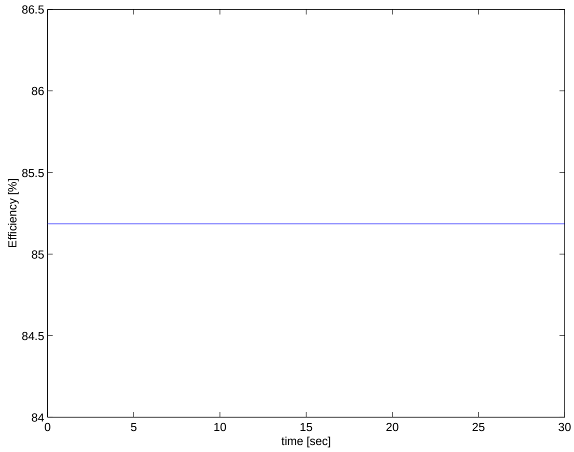


(c)

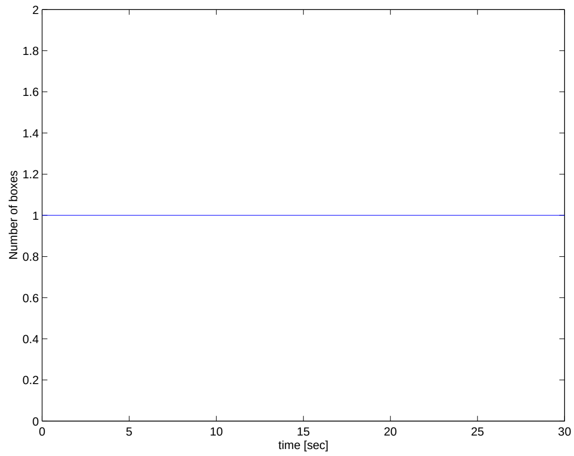


(d)

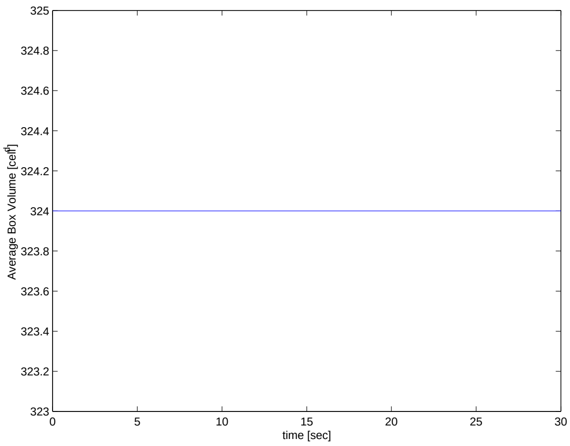
Figure 10: Sphere-diag test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) Initial timestep. (b) $t = 19$. (c) $t = 20$. (d) $t = 21$.



(a)



(b)



(c)

Figure 11: Sphere-diag test case: statistics. (a) Efficiency versus time. (b) Number of boxes versus time. (c) Average box volume versus time.

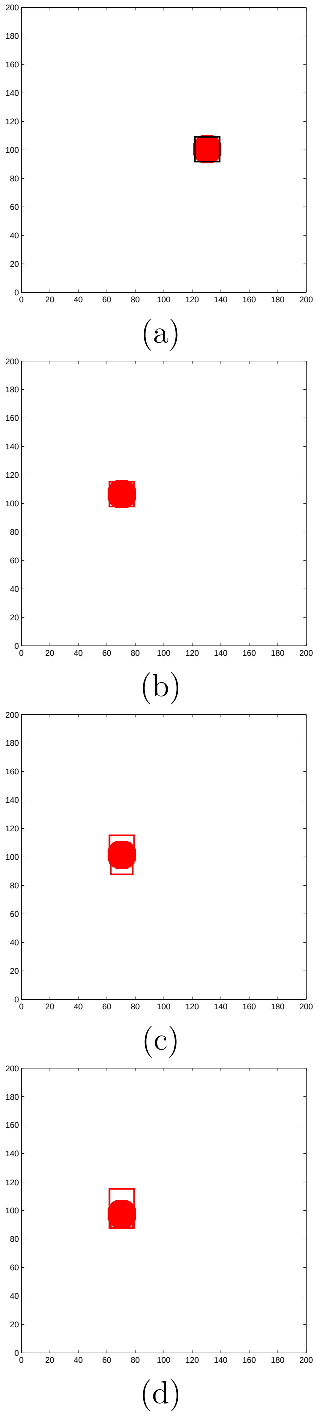
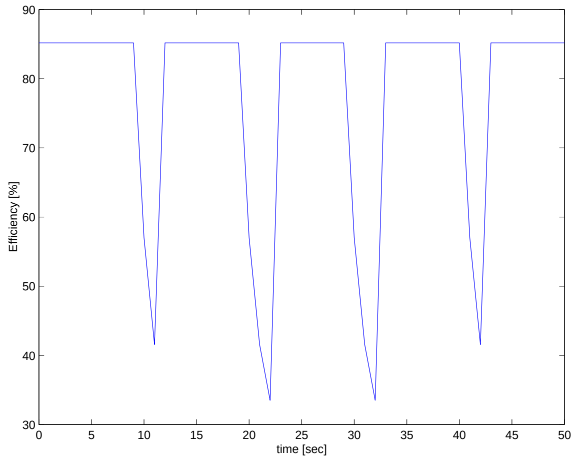
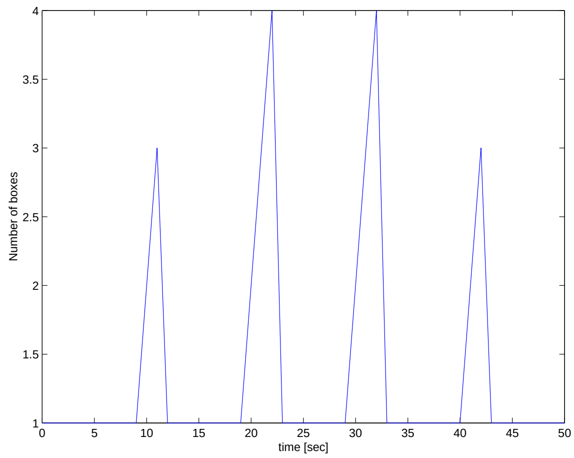


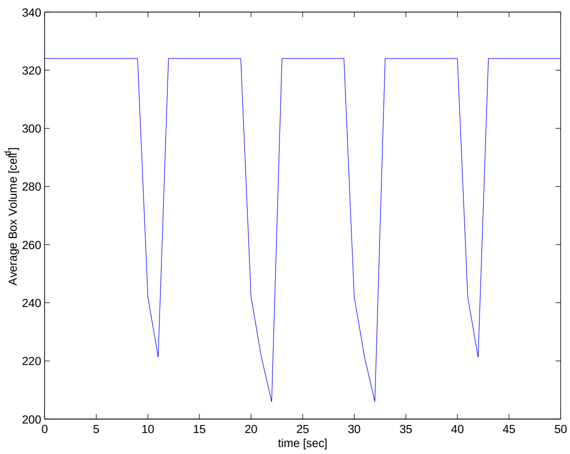
Figure 12: Sphere-circ test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) Initial timestep. (b) $t = 19$. (c) $t = 20$. (d) $t = 21$.



(a)



(b)



(c)

Figure 13: Sphere-circ test case: statistics. (a) Efficiency versus time. (b) Number of boxes versus time. (c) Average box volume versus time.

4.6 Summarizing Statistics

The following tables compare two sets of tests: the first table refers to the algorithm that re-grids at every timestep. The second table updates whenever it can.

Table 1: Summarizing Statistics, no update: $regrid = 1$

Case	Avg. Efficiency	Avg. Patch Life	Avg. Time Re-grid
Sphere-x	.852	1.0	1.00
Sphere-mx	.852	1.0	1.00
Sphere-y	.852	1.0	1.00
Sphere-diag	.852	1.0	1.00
Sphere-circ	.852	1.06	1.00

Table 2: Summarizing Statistics, update: $regrid = \infty$

Case	Avg. Efficiency	Avg. Patch Life	Avg. Time Re-grid
Sphere-x	.417	17.7	∞
Sphere-mx	.417	17.7	∞
Sphere-y	.417	17.7	∞
Sphere-diag	.852	1.0	1.00
Sphere-circ	.775	1.44	1.25

5 Discussion and Summary

We developed a variant of the Berger-Rigoustos clustering algorithm [BR91] that is able to use "history" from previous timesteps for updating the set of boxes, rather than plainly re-grid at every timestep. However, this work is far from complete.

First, it is clear that when there are several disjoint (and say, far enough apart) areas of flagged cells, the algorithm should ideally treat each area separately from all others. Thus, even if we fail to update the box set in one area, that should not affect the decision in other areas (unlike the current algorithm).

Furthermore, the diagonal and circular movement cases did not result in satisfactory updates. We needed to re-grid at almost every timestep. The updating technique can be improved to treat such cases much more efficiently. This work will appear in a future report.

Importantly, we can see in Tables 1–2 that the efficiency of the sets generated by updating an old set is generally much lower than when we re-grid at every timestep. This is to be expected, as we do not optimize from scratch but confine ourselves to improving an existing covering boxes solution. However, we should not worry about that: efficiency plays a major role only when the areas of flagged cells are large, and if they move continuously, changes in the efficiency due to updating would be felt only at the boundaries of the flagged areas. Overall, that would not change the efficiency very much (unlike the test cases on this report, that had fairly small flagged areas).

Another important addition is the treatment of boxes near the physical boundaries (the boundaries of the "coarse grid"). Although seemingly technical, we should in a future research make sure that boxes adjacent or near the boundary are correctly generated.

In sum, this is a first version of an algorithm that can "move" patches over time. Our next development stages will be concerned with improving its updating performance, and analyzing representative test cases of practical movement (e.g., colliding or separating spheres).

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.

[BR03] A. Brandt and D. Ron. Multigrid solvers and multilevel optimization strategies. In J. Cong and J. R. Shinnerl, editors, *Multilevel Optimization in VLSICAD*, pages 1–93. Kluwer, Dordrecht, Holland, 2003.

[Bra84] A. Brandt. *Multigrid techniques: 1984 guide with applications to fluid dynamics*. GMD–Studien Nr. 85. Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, 1984.