SCI INSTITUTE
TECHNICAL REPORT

Clustering Patches for Multi-Level Refined Grids: a Hierarchical Approach

Oren E. Livne

UUSCI-2006-004

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

January 23, 2006

Abstract:

We present an alternative approach to Adaptive Mesh Refinement (AMR) to the Berger-Rigoustos concept of floating patches. The current approach create patches of a prescribed size, on a prescribed lattice, at any level of refinement. Similar to cell dissection refinement, this approach is hierarchical, allowing easy implementation of multi-level refinement, treatment of safety layers, and automatic update over timesteps. We explain the implementation, study several illustrious test cases, and monitor our AMR levels efficiency. The algorithm is presented in 2D, but readily extends to 3D.

THE UNIVERSITY OF UTAH

Clustering Patches for Multi-Level Refined Grids: a Hierarchical Approach

Oren E. Livne *

January 23, 2006

Abstract

We present an alternative approach to Adaptive Mesh Refinement (AMR) to the Berger-Rigoustos concept of "floating patches". The current approach create patches of a prescribed size, on a prescribed "lattice", at any level of refinement. Similar to cell dissection refinement, this approach is hierarchical, allowing easy implementation of multi-level refinement, treatment of "safety layers", and automatic update over timesteps. We explain the implementation, study several illustrious test cases, and monitor our AMR levels' efficiency. The algorithm is presented in 2D, but readily extends to 3D.

Key words. hierarchical levels, re-griding, persistent refinement patches.

1 Introduction

Reports 1–3 discussed the Berger-Rigoustos (BR) algorithm [BR91] for clustering cells that are flagged by a time-stepping code as "needing refinement". The advantage of the BR approach is patch efficiency: patches tightly bound the areas of flagged cells. However, a multi-level implementation of BR is

*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

quite involved, it does not update patches efficiently over time-steps, and would be as wasteful as any other AMR approach that uses rectilinear patches only, in the presence of thin shock fronts that do not align with the original grid.

In this report we present an alternative approach, that uses a hierarchy of refinement level. Many papers have been devoted to such approaches (e.g., [Ber86], [Nee96]). In our case, every level is a union of patches. A patch can exist only at certain alignments and can have a prescribed size. The levels are organized so that each patch can be dissected into the next-finer level patches. This approach is easier to implement for multi-level refinement, naturally supports updates of patches over time-steps, and should not have a much smaller efficiency than BR in most scenarios, let alone thin shock fronts.

Our necessary requirements of the AMR level hierarchy are as follows.

1. *Objective 1 - Maximum efficiency:* the ratio of the number of flagged cells to the total patch 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 patch area.
2. *Objective 2 - Minimum flag distance from patch boundary:* because we are not sure whether flagged cells are due to effects in neighboring cells, and because of the reason for Objective 4 below, we want to keep several "layers of cells" between the flagged cells and patch boundaries, at any level.
3. *Objective 3 - Minimum distance between boundaries of patches at consecutive levels:* because of the ICE's computational framework, the boundary of a fine level $(k+1)$ patch should be separated from coarse level (k) patch boundary by at least one layer of cell at a coarse level cells (otherwise, there are problems in defining the boundary conditions at levels $k+1$).
4. *Objective 4 - Fast update:* if the flagged cells describe a moving shock front, we would like to use the patch covering from the previous timestep, and make minor modifications to it to fit it to the new timestep's flagged cells. This is in fact another way of looking at Objective 3; but it also relates to the time required to generate the updated AMR levels at a new timestep, from an AMR hierarchy at the previous timestep.

5. *Objective 5 - Maximum patch volume:* patches that are 32^3 or $64\times64\times8$ are equivalent in terms of memory and cost, but we would not want very large patch volumes in light of a worse load balancing between processors, in a parallel processing framework.

6. *Objective 6 - cell alignment:* cell boundaries at level $k+1$ must align with cell boundaries at the coarser level k. and cost, but we would not want very large patch volumes in light of a worse load balancing between processors, in a parallel processing framework.

   In addition, we would like to have the following objectives, which are however not to be strictly enforced:

7. *Objective 7 - Minimum patch size:* the smallest patch 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 patches.

8. *Objective 8 - Minimum patch mutual-boundary area:* to minimize processor communication, we would like the patches to have as low mutual edges as possible. One way to indirectly achieve this is by trying to construct more "cubic" patches than "thin" ones, thereby reducing the edge area of each patch (independently of the other patches' edge area, though).

All of Objectives 1–8 are address by the hierarchical approach described in this report. Thus, it seems that this is a good approach that we should next implement in the Uintah framework.

This report is organized as follows. In §2 we list the measures by which we measure the algorithm's result. In §2.1 we describe the griding algorithm. §3 contains a MATLAB implementation of the two main routines of our code. 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 Indicators

We assess the quality of the time-dependent constructed set of patches by the following measures. The measures are naturally related to the objectives listed in §1.

1. Global indicators (computed versus time):
   (a) Total number of active patches at all levels.
   (b) Total number of active cell at all levels.
   (c) Total number of patch creations at all levels.
   (d) Total number of patch deletions at all levels.
2. Level-dependent indicators (computed versus time):
   (a) Number of active patches.
   (b) Number of active cells.
   (c) Percent of active patches out of all possible patches.
   (d) Average patch efficiency: ratio of the number of flagged cells to the patch volume, averaged over patches (this may also be viewed as the level efficiency, as most patches have equal volume; see §2.1).
   (e) Median patch efficiency.
   (f) Maximum patch efficiency (it better be close to 1).
   (g) Number of empty patches (patches with no flagged cells in them).
   (h) Number of patch creations.
   (i) Number of patch deletions.

2.1 The Hierarchical Griding Algorithm

The hierarchical griding algorithm is easily incorporated as a module of time-stepping. Prior to time-stepping, we initialize a structure of `max-levels` levels. At each time, we loop over levels from coarsest (level o) to one-before-the-finest (level $max - levels + o - 2$), and determine the patches of the next-finer levels, if there are flagged cells in them; or, if they are needed for keeping Objectives 3–5. This strategy is described in the following pseudo-code. The function `levels-create` initializes levels; `mark-patches` determines the patches of the next-finer levels from the given flagged cells $Lk.cell - err$ at level k (denoted Lk). In the process of creating level k, all levels coarser than k (i.e. l with $o \geq l < k$) can change.

```
levels-create(max-levels);     % Initialize levels
for timestep = 0:num-tsteps-1,
    Advance to time t;
    for k = o:max-levels+o-2   % Mark patches for refinement at all levels
        % RENDERING AND MARKING PATCHES FOR REFINEMENT, LEVEL K
        Get input: flagged cells array L{k}.cell-err at this level & time
        dilate L{k}.cell-err as required;
        mark-patches(k);       % Mark patches that need refinement
    end
    Accumulate statistics;
end
```

2.2 Level Definition: Lattice

The input for the algorithm is the coarsest grid L_o, defined in the Uintah framework. L_o is uniform and extends over the entire physical domain. We will restrict ourselves to a 2D description for simplicity; however, the code and concepts work in any dimension. Let $n_1 \times n_2$ be the size in cells of L_o. To address Objectives 5 and 7, we also assume that we are given a patch size $p_1 \times p_2$ (cells). If n_i is not divisible by p_i, the last patch is slightly bigger (if the remainder is less than $p_i/2$) or slightly smaller (if the remainder is greater or equal to $p_i/2$) in direction i, $i = 1, 2$. Each patch is processed by a different processor, thus it is up to the user to define patch size to comply with the minimum/maximum size requirements.

Unlike standard bisection algorithms, we allow a more general framework for refining each patch. We assume to be given an input "lattice resolution" $m_1 \times m_2$, that determines the size of the next-level (L_{o+1}) patches. The Uintah framework requires that L_{o+1} cells align with L_o cells (Objective 6). Thus, if the cell refinement ration between levels o and $o+1$ is denoted by r_i in direction i, $r_i p_i/m_i$ must be an integer, for $i = 1, 2$. $\{r_i\}_i$ is also assumed to be an input to our algorithm. The inputs discussed here (m, p, r and the size of L_i) are assumed to be given for the entire level hierarchy, and remain unchanged over timesteps. Fig. 1 illustrates a possible L_o and its lattice. Note that m, p, r control the patch size and cell size at all levels. These numbers have to be coordinates by the user to avoid level cell mis-alignments, or too small/too big patches.

The reason for this generalized dissection strategy, as opposed to the bisection strategy, stems from complexity considerations. With bisection

and a 1 : 2 cell refinement ratio between each two consecutive levels, each patch of L_{k+1} has the same number of cells as in a patch of L_k. Hence, when refining a singularity at a corner, we would have (say) one patch per level, and the total work would be unbounded when the number of levels would be increased. A finer lattice than 2×2 (per patch) makes the work at L_{k+1} patch smaller than a coarser patch, but the number of cells in a patch decreases when k increases. This means that at a certain level we would have patches of size 1×1 cells that cannot be further sub-divided, which forces us to go back to the 2×2 bisection lattice from that point on; but at least, that happens at a fine grid, whose associated complexity is (hopefully) equivalent to few percents of the complexity of processing the coarsest level. By limiting the number of levels to the reasonable number usually permitted by the other considerations of the ICE framework, bisecting from that point on would not accumulate to as much work as with bisection at all levels.

2.3 The Dilation Operation

Given a set of flagged cells on a grid (or equivalently, a binary image that has 1 for flagged cells, 0 otherwise), a dilated list is a convolution with a filter that determines "which neighbors count" for the dilation. This convolution may be applied n times to obtain n "safety layers" from the original list of cells. An example of a convolution with a 5-point (nearest neighbors) and a 9-point filter is given in Fig. 2.

2.4 Level Refinement

At a fixed timestep, we loop over all levels (coarsest to one-before-the-finest). At each level (k), we keep a list of coordinates of the active patches. The lower-left corner patch of the lattice is numbered $(0, 0)$ for simplicity. The process of level refinement marks which sub-patches of each L_k-patch will be active at L_{k+1} (these will be called sub-patches that are "marked for refinement").

We keep four lists of flagged cells: (a) the original one; (b) dilated one for creating patches; (c) dilated for deleting patches' and (d) dilated list for keeping fine patches from boundaries of coarse patches. (b) is a dilation by typically $2 - 5$ cell layers of (a). If there exists a cell of (b) in a sub-patch, it is marked for refinement. (c) is typically a larger dilation ($5 - 10$ layers).

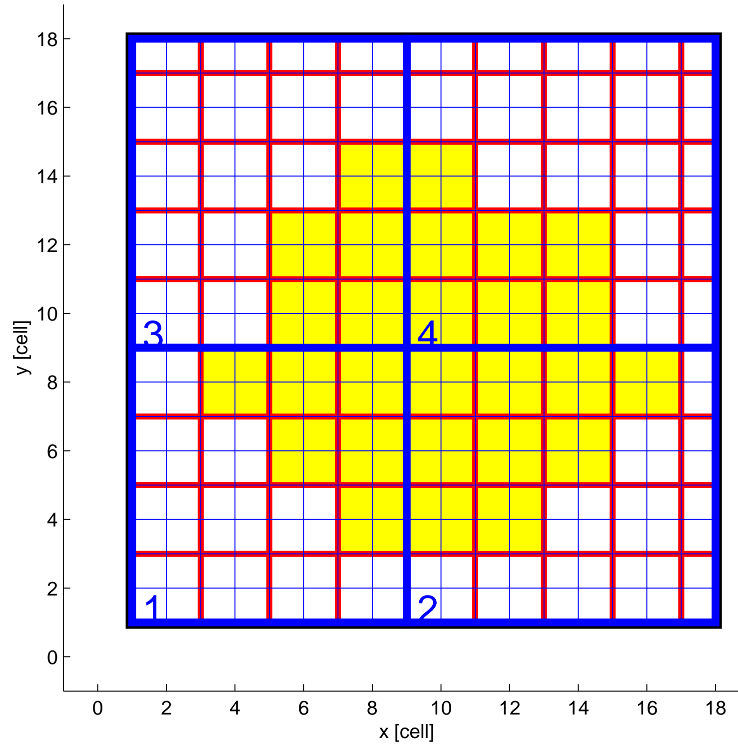
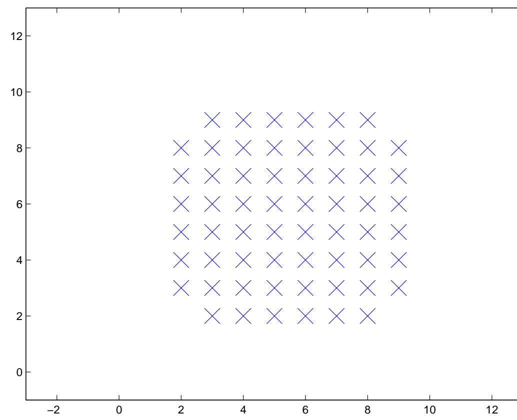
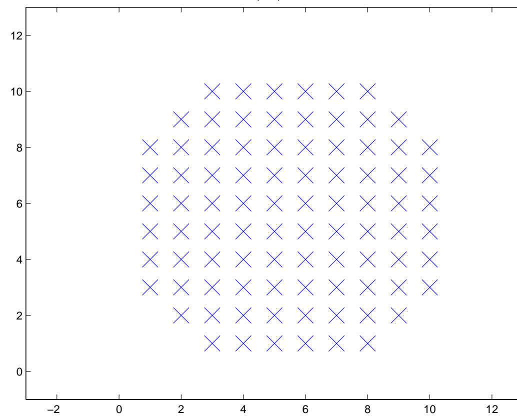


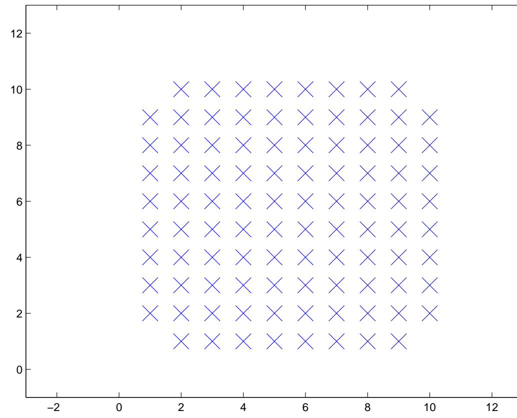
Figure 1: An example of the coarsest level L_o. Here L_o has 17×17 cells (the blue lines), divided into four patches $1-4$ (the size of the "standard" patch is 8×8 cells; the thick blue lines). Each patch may be divided into 4×4 sub-patches of size 2×2 L_o-cells (the red lines). For instance, L_{o+1} can consist of all the highlighted patches. If L_{o+1} is twice finer, each of its patches is 4×4 cells, except near the boundaries of the physical domain.



(a)



(b)



(c)

Figure 2: An example of a dilation. (a) The original list of flagged cells. (b) Dilation with a 5-point ("star") filter - diagonal neighbors don't count. (c) Dilation with a 9-point ("box") filter - diagonal neighbors count.

If there exists a cell of (c) in a sub-patch, it is not deleted (even if there are no cells of (b) in it). (b) and (c) determine L_{k+1}.

Once L_{k+1} has been computed, (d) is defined as a dilation by one or few cell layers of the all cells encompassed by existing L_{k+1}, after it is determined by (b) and (c). If there exists a cell of d that is not covered by a L_k-patch, this patch is added to level k. Then we compute a similar (d)-list for L_{k-1} and possibly expand its patches to cover it, and so on, until the coarsest grid (that already extends over the full domain).

3 Full MATLAB Code

In this section we include two routines: `test-movement` and `mark-patches`. The first is a driver that loops over timesteps and moves a certain (synthetic) object in a certain direction inside the domain of the coarse grid in ICE. The second routine is the main code for refining level k into level $k+1$, possibly updating all levels coarser than $k+1$ in the process.

```
%function tout = test-movement(tin)
%TEST-MOVEMENT Driver for moving objects and creating moving patch collections around them.
%   TOUT = TEST-MOVEMENT(TIN) is a driver for CREATE-CLUSTER and UPDATE-CLUSTER functions,
%   that takes shapes of flagged cells (e.g., circles, boxes) and their direction of movement,
%   and moves them over a domain as in "time-stepping". The objects and their movements are
%   specified by the structure TIN, that can be prepared by the function TEST-CASE. For each
%   timestep, we generate the boxes around the flagged cells by MARK-PATCHES.
%   Statistics on how many re-boxing are actually needed, etc.,
%   are printed and plotted at the end of the run. A sample of timestep configurations
%   of the cells and their covering boxes, is also plotted. The statistics and original
%   parameters of TIN are output in the structure TOUT.
%
%   See also MARK-PATCHES, OBJECT-RENDER, TEST-CASE.

%%%%% Set and print parameters
global-params;
o               = 1;                                    % List offset for MATLAB; this is the coarsest grid index
tin             = test-case('ball-x');                  %---------- Load parameters of this test case ----------

fprintf('<<<<<<<<<<<<<< TEST-MOVEMENT: moving flagged and boxes around them >>>>>>>>>>>>>>\n');
if (tin.print)
    fprintf('Parameters:\n');                           % Print parameters
    fprintf('\tTest case title   : %s\n',tin.title);    % Title string
    fprintf('\tDomain size\t\t= ');
    print-vector(tin.domain,'x','float');
    fprintf('\n');
    fprintf('\tNumber of timesteps\t= %d\n' ,tin.num-tsteps);
    fprintf('\tInitial time\t\t= %f [sec]\n',tin.init-t);
    fprintf('\tDelta-t\t\t\t= %f [sec]\n',tin.dt);
    fprintf('\n');
    fprintf('\t# safety layers, boundary= %d [cell]\n',tin.safe-bdry);
    fprintf('\t# safety layers, create\t= %d [cell]\n',tin.safe-create);
    fprintf('\t# safety layers, delete\t= %d [cell]\n',tin.safe-delete);
    fprintf('\n');
    fprintf('\t# processors\t\t= %d\n',tin.num-procs);
    fprintf('\tMin. patch side length\t= %d [cell] \n',tin.min-side);
    fprintf('\tMax. patch volume\t= %d [cell^d]\n',tin.max-volume);
    fprintf('\n');
    fprintf('\tPlot flag\t\t= %d\n'        ,tin.plot);
    fprintf('\tPrint flag\t\t= %d\n'        ,tin.print);
    fprintf('\tDelay time\t\t= %f [sec]\n'   ,tin.delay);
    fprintf('\n');
    fprintf('\tMax. # levels\t\t= %d\n' ,tin.max-levels);
    fprintf('---------------------------------------------------------------------------------------\n');
    fprintf('\n');
```

```
end

%%%%% Create the list of levels and init stats array (tout)
dim                 = length(tin.domain);                       % Dimension of the problem
L                   = cell(tin.max-levels,1);                   % List of levels; L{o} is the coarsest, L{o+1}-finer, etc.
levels-create(tin.max-levels);                                  % Init levels o..max-levels+o-1
print-level-info;
tout                = [];
tout.in             = tin;                                      % Save input parameters in the output data
tout.t              = zeros(tin.num-tsteps,1);
tout.tstep          = zeros(tin.num-tsteps,1);
tout.data           = zeros(tin.num-tsteps,length(L),9);
tout.sum-data       = zeros(tin.num-tsteps,4);

%%%%% Main loop over time steps
for count = [0:tin.num-tsteps-1],
    t = tin.init-t+count*tin.dt;
    fprintf('Time step = %d, Time = %f [sec]\n',count+o,t);
    for k = o:length(L)+o-1                                     % Initialize some counters
        L{k}.num-created    = 0;
        L{k}.num-deleted    = 0;
    end

    for k = o:length(L)+o-2                                     % Mark patches for refinement at all levels
        fprintf('---- Level %d: RENDERING AND MARKING PATCHES FOR REFINEMENT ----\n',k);
        L{k}.cell-err           = object-render(t,k);           % Synthesize the flagged cells array at this level
        L{k}.cell-err-create    = dilate-list(L{k}.cell-num,...
            L{k}.cell-err,tin.safe-create);                     % Dilate by a big amount when creating patches
        L{k}.cell-err-delete    = dilate-list(L{k}.cell-num,...
            L{k}.cell-err,tin.safe-delete);                     % Dilate by a small amount when checking patches
        if (tin.plot >= 2)
            fprintf('Before marking\n');
            for l = o:length(L)+o-1                             % Display all finer grids
                plot-grid(l);
            end
%            pause
        end
        mark-patches(k);                                        % Mark patches that need refinement, based on L{k}.flagged
        if (tin.plot >= 2)
            fprintf('After marking\n');
            for l = o:length(L)+o-1                             % Display all finer grids
                plot-grid(l);
            end
%            pause
        end
    end

    %%%%% Accumulate statistics, printout
    plot-composite-grid(t);
    if (tin.plot >= 1)
        if (ismember(count+o,tin.tsteps-save))
            fprintf('Saving grid\n');
            eval(sprintf('print -depsc %s-grid-t%d.eps',tin.title,count+o));
        end
    end
    if (tin.print >= 2)
        print-level-info;
    end
    tout.t(count+o)             = t;
    tout.tstep(count+o)         = count+o;
    tout.data(count+o,:,:)      = levels-stats;
    tout.sum-data(count+o,:)    = sum(squeeze(tout.data(count+o,:,[1 2 8 9])),1); % Sum over levels the following columns of tout.data: 1. #patches 2. #c
    fprintf('##########################################################################################\n');
end

%%%%% Final plots, prepare output structure
eval(sprintf('save %s-data.mat tout',tin.title));
final-plots(tout);

function mark-patches(k)
%MARCH-PATCHES Mark patches for refinement.
%   MARK-PATCHES(K) marks patches for refinement at level K, based on the flagged
%   cells that are stored at L{K} based on the information from TOUT and object
%   rendering (movement) information.
%
%   See also OBJECT-RENDER, TEST-MOVEMENT.
```

```
global-params;

dim                = length(L{k}.cell-num);                     % Dimension of the problem
b                  = box-list(zeros(1,dim),L{k}.rat-patch);     % Prepare all combinations for sub-patch index offsets
patch-new          = zeros(0,dim);                              % List of newly created patches at k+1
L{k}.deleted-patches= zeros(0,dim);
siz-sub            = L{k+1}.patch-size./L{k}.rat-cell;                  % Size of a sub-patch [L{k} cells]

for p = o:size(L{k}.patch-active,1)+o-1,                        % Loop over patches at this level
    j       = L{k}.patch-active(p,:);                           % d-D coordinates of the patch
    start   = patch-start(k,j);                                 % Start cell of patch j
    finish  = patch-finish(k,j);                                % Finish cell of patch j
    siz     = finish - start + 1;                               % Size of patch

    %%%%% Check whether there are flagged cells in each quadrant, create/delete patches on L{k+1} accordingly
    for q = o:size(b,1)+o-1,                                    % Loop over all possible patches; that might be 3x3 for a "normal" patch and 4x4 for a "b
        start-sub   = start + b(q,:).*siz-sub;                  % Sub-patch start cell [L{k} cells]
        if (~isempty(find(start-sub > finish)))                 % We are outside the L{k}-patch, skip this one
            continue;
        end
        start-f     = convert-c2f(k,start-sub,'cell');          % Sub-patch start cell [L{k+1} cells]
        j-f         = convert-c2p(k+1,start-f);                 % Sub-patch patch coordinate [L{k+1} patches]
        finish-sub  = start-sub + siz-sub - 1;
        last        = find(j-f == L{k+1}.patch-num);
        finish-sub(last) = L{k}.cell-num(last);
        ind         = patch-find(k+1,j-f);                      % Index of sub-patch at level k+1 (if exists)
        a           = find(check-range(...
            L{k}.cell-err-create,start-sub,finish-sub) > 0);    % Flagged cells in this patch (including small dilation), for creating patches
        ad          = find(check-range(...
            L{k}.cell-err-delete,start-sub,finish-sub) > 0);    % Flagged cells in this patch (including big dilation), for deleting patches
        if (~isempty(a) & (ind < 0))                            % There exist flagged cells of small dilation area and child doesn't exist => create it c
            if (tin.print >= 2)
                fprintf('-CREATE-  Patch [ ');
                fprintf('%d ',j);
                fprintf('], sub-patch [ ');
                fprintf('%d ',b(q,:));
                fprintf('] has flagged cells => refined to Level %d, Patch [ ',k+1);
                fprintf('%d ',j-f);
                fprintf(']\n');
            end
            L{k+1}.patch-active = [L{k+1}.patch-active; j-f];   % Add to level k+1 patches
            patch-new = [patch-new; j-f];                       % Add to list of newly created patches
            L{k+1}.num-created = L{k+1}.num-created+1;
        end
        if (isempty(ad) & (ind >= o))                           % No flagged cells of big dilation area and child exists => delete child and everything b
            if (tin.print >= 2)
                fprintf('-DELETE-  Patch [ ');
                fprintf('%d ',j);
                fprintf('], quadrant %d has no flagged cells => delete Patch %d,%d [ ',q,k+1,ind);
                fprintf('%d ',j-f);
                fprintf('] + childs\n');
            end
            start-f     = j-f;
            finish-f    = j-f+1;
            for l = k+1:length(L)+o-1
                c = L{l}.patch-active;
                a = find(check-range(c,start-f,finish-f-1) > 0);
                if (tin.print >= 2)
                    fprintf('\tLevel %d: start-f=(%d,%d), finish-f=(%d,%d), children indices=',l,start-f,finish-f);
                    if (size(a,1) > 1)
                        a = a';
                    end
                    fprintf('%d ',a);
                    fprintf('\n');
                end
                if (isempty(a))
                    break;
                end
                L{l}.deleted-patches = [L{l}.deleted-patches; L{l}.patch-active(a,:)];  % Save deleted patches for plots
                L{l}.patch-active(a,:)  = [];                   % Delete patch children from level k+1 children list
                L{l}.num-deleted = L{l}.num-deleted + length(a);

                if (l < length(L)+o-1)
                    start-f  = convert-c2f(l,start-f,'patch');  % Convert from L{l-1} to L{l} coordinate
                    finish-f = convert-c2f(l,finish-f,'patch');
                end
            end
```

```
        end
    end
end

%%%%% Add cells at coarser levels to ensure that L{k+1} patches have safety layers from the
%%%%% boundaries of patches at this levels.
for l = k+1:-1:o+1,                                         % Loop from finest to coarsest
    patch-new   = L{l}.patch-active;
    if (isempty(patch-new))                                 % Until there are no patches to be added
        break;
    end
    patch-new-cells = zeros(0,dim);                         % Mark ALL L{l-1}-cells within the new L{l} patches
    for i = 1:size(patch-new,1),
        pf          = patch-new(i,:);                       % L{l} patch coords
        start       = convert-f2c(l,patch-start(l,pf),'cell');  % Starting cell of pf in L{l-1} cell coordinates
        finish      = convert-f2c(l,patch-finish(l,pf),'cell'); % Finishing cell of pf in L{l-1} cell coordinates
        patch-new-cells = union(patch-new-cells,box-list(start,finish),'rows');
    end
    patch-new-cells-d = dilate-list(L{l-1}.cell-num,patch-new-cells,...
        tin.safe-bdry,'box');                               % Dilate these cells by the #safety L{l-1}-cell layers we want; diagonal nbhrs count ('bo
    patch-needed    = unique(convert-c2p(l-1,patch-new-cells-d),'rows');        % L{l-1} patches that cover the L{l-1}-cells of interest
    add             = find(patch-find(l-1,patch-needed) < 0);   % Need to add these patches to L{l-1} because they don't exist
    patch-new       = patch-needed(add,:);                  % Added patches L{l-1} coordinates; update patch-new for yet-coarser level (l-1->l-2) upc
    L{l-1}.patch-active = [L{l-1}.patch-active; patch-new];     % Add to level l-1

    L{l-1}.new-patch        = patch-new;                    % Save added patches
    L{l-1}.new-cells        = patch-new-cells;              % Save cells in new patches
    L{l-1}.new-cells-bdry   = patch-new-cells-d;            % Save dilated cells from new patches

    new     = size(patch-new,1);
    rein    = length(find(ismember(L{l-1}.deleted-patches,patch-new,'rows')));        % Reincarnated patches (deleted and then created => they were in f
    L{l-1}.num-created  = L{l-1}.num-created + new - rein;
    L{l-1}.num-deleted  = L{l-1}.num-deleted - rein;

    if (~isempty(add))                                      % Print info on patches added to L{l-1}
        if (tin.print >= 2)
            fprintf('-SAFE CREATE-  Level %d\n',l-1);
            for i = 1:size(patch-new,1),
                fprintf('\tPatch ');
                print-vector(patch-new(i,:),',','int');
                fprintf(' starts at ');
                print-vector(patch-start(k,patch-new(i,:)));
                fprintf('\n');

            end
        end
    end
end
```

4 Numerical Experiments

Each test case describes a movement of a set of points as in "time-stepping". The size physical size of the domain was 16×16 meters. We performed 30 timesteps, starting from $t = 0[s]$ till $t = 30[s]$, with $\Delta t = 1[s]$. We generated four levels of refinement. Their sizes, cell sizes and lattice sizes are summarized in Table 1. "Cell ratio" is the cell refinement ratio between a level and the next-finer level. "Lattice ratio" is the lattice refinement ratio, that is, the number of sub-patches in each patch of the current level.

Table 1: Levels Information

Level	Cell Size [meter]	Grid Size [cell]	Patch Size [cell]	Lattice Size [patch]	Cell Ratio	Lattice Ratio
1	1.882×1.882	17×17	8×8	2×2	2×2	4×4
2	0.941×0.941	34×34	4×4	9×9	2×2	2×2
3	0.471×0.471	68×68	4×4	17×17	2×2	2×2
4	0.235×0.235	136×136	4×4	34×34	-	-

Simulating four levels required generating a list of flagged cells for the first three levels, $o, o+1, o+2$. Thus, when we refer to a "moving ball", we mean that a full disk of flagged cells that is moving in a certain direction, constitutes the data at level L_o that defines L_{o+1}; to simulate a practical situation, we defined only an annulus that is contained in that ball, to be the list of flagged cells at level L_{o+1}. The flagged cells list at L_{o+2} is the "right half" of this annulus. This is not a very practical scenario, because there are relatively many cells that are flagged at the finer levels. Consequently, efficiency results are to be taken as illustrations only, not as meaningful absolute numbers.

The dilation parameters were: 1 layer for (b) (creating patches), 2 layers for (c) (deleting patches), and 1 layer for coarse-fine patch boundary separation (list (d)).

4.1 Example: Ball Moving in Positive-x

This test case is denoted “Ball-x”. We start with a ball 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. Fig. 3 show the active patches at the four levels of refinement, their sub-patches that were marked for refinement, and the flagged cells (without dilation). See also §4.2.

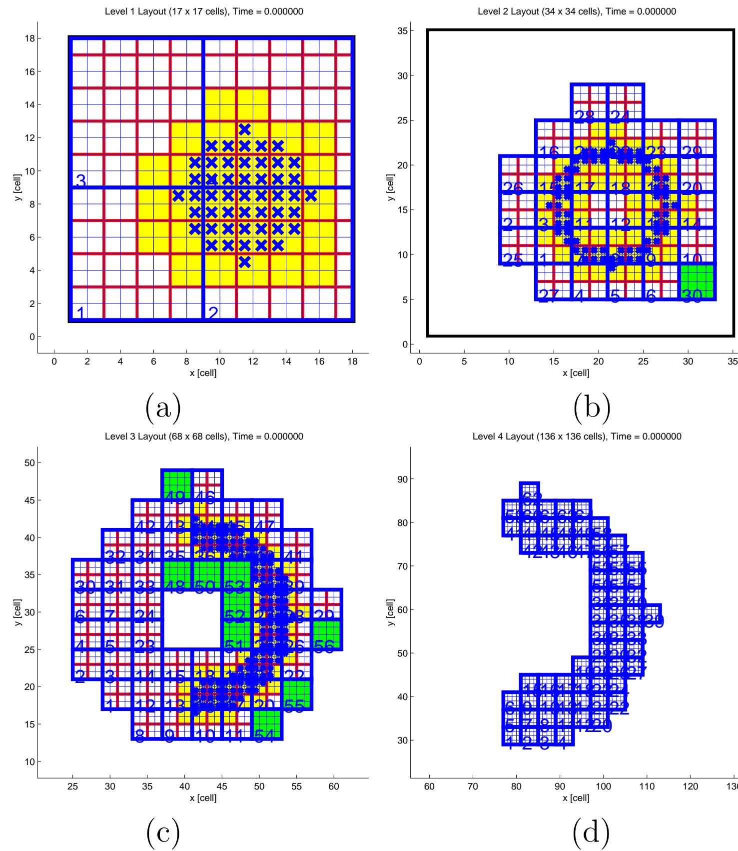


Figure 3: Ball-x test case at a certain timestep: different levels and their flagged cells. (a) is L_1 (here $o = 1$), (b) is L_2, (c) is L_3, and (d) is L_4. For each level we show its active patches (thick blue lines) and their cells (thin blue lines), lattice lines (red lines), flagged cells (marked by “x”’s), and sub-patches that are marked for refinement (highlighted in yellow). Patches that are added because of the (d)-list (boundary separation) are highlighted by green.

The composite grid and the flagged cells at all levels are plotted in Fig. 4.

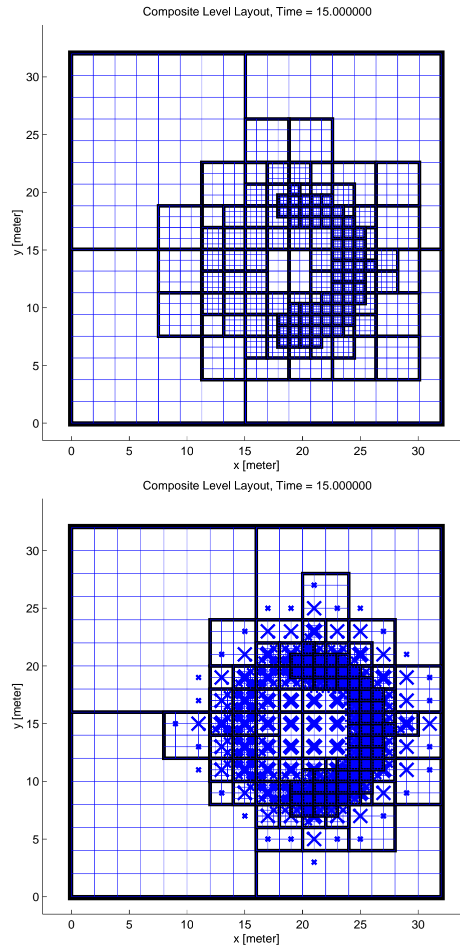


Figure 4: Upper figure: the composite grid. Lower figure: the composite grid with the flagged cells at all levels (marked by "x"'s).

4.2 Ball Moving in Positive-x

We start with a ball near the left x-boundary of the domain (and approximately in the middle in y), and move it one cell to the right in the x-direction. The schematic movement is shown in Fig. 5. Fig. 6 shows several snapshots of the generated AMR grids at several timesteps. Fig. 7 contains some summarizing statistics of this test.

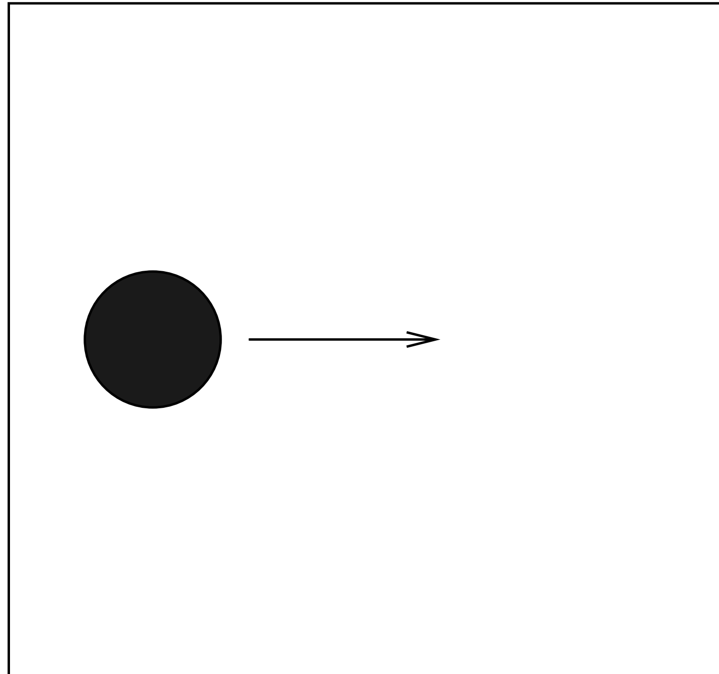


Figure 5: Ball-x test case: a ball moving in positive x-direction.

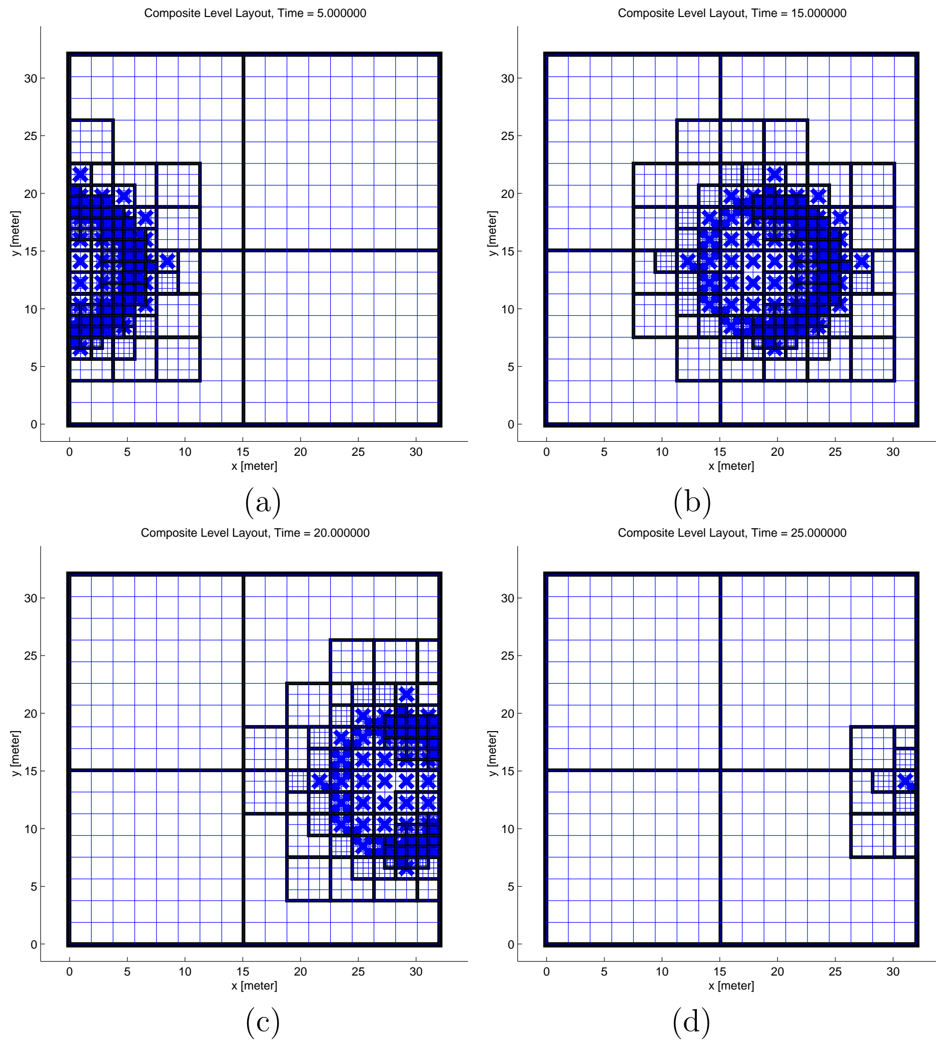
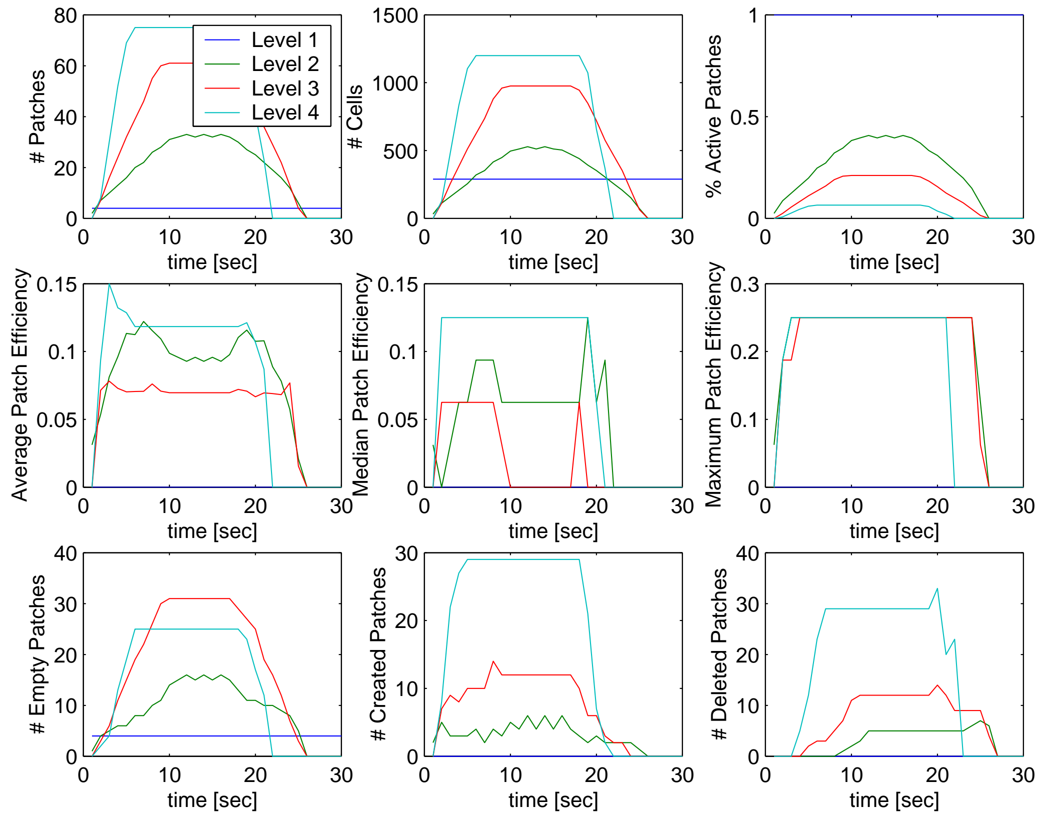
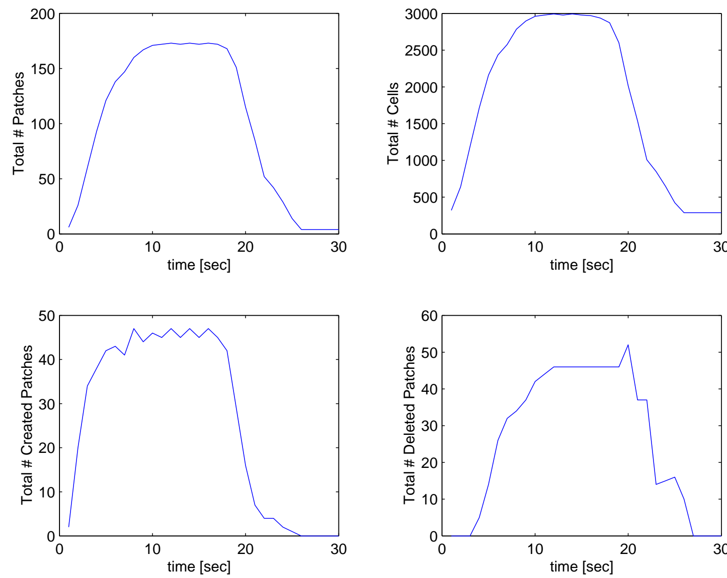


Figure 6: Ball-x test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 5$. (b) $t = 15$. (c) $t = 20$. (d) $t = 25$.



(a)



(b)

Figure 7: Ball-x test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

4.3 Ball Moving in Negative-x

This test case is denoted “Ball-mx”. It is identical to Ball-x, except that we now start with the ball 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. The results are similar to Ball-x, as may be expected.

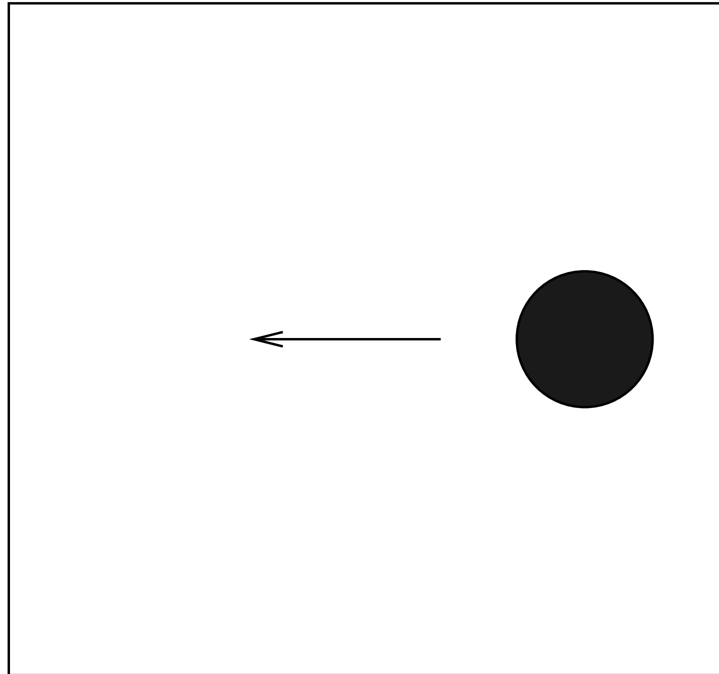


Figure 8: Ball-mx test case: a ball moving in positive x-direction.

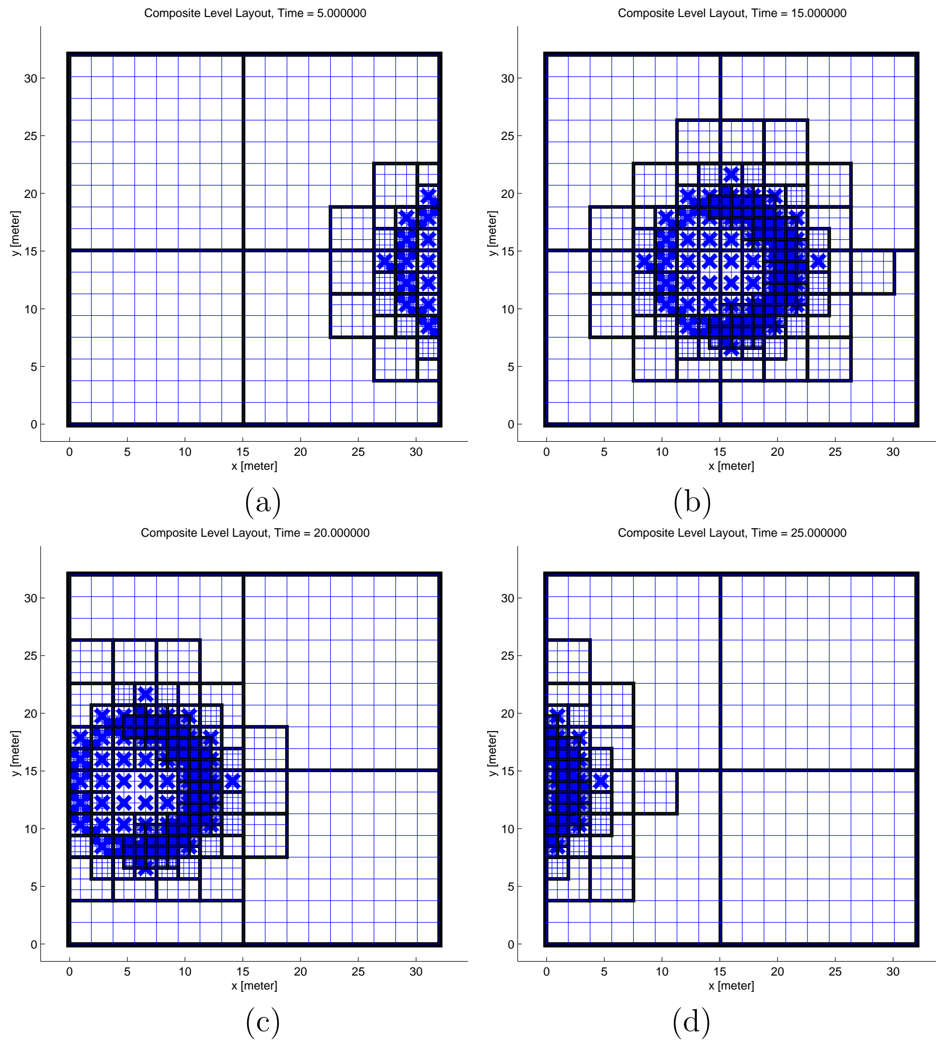
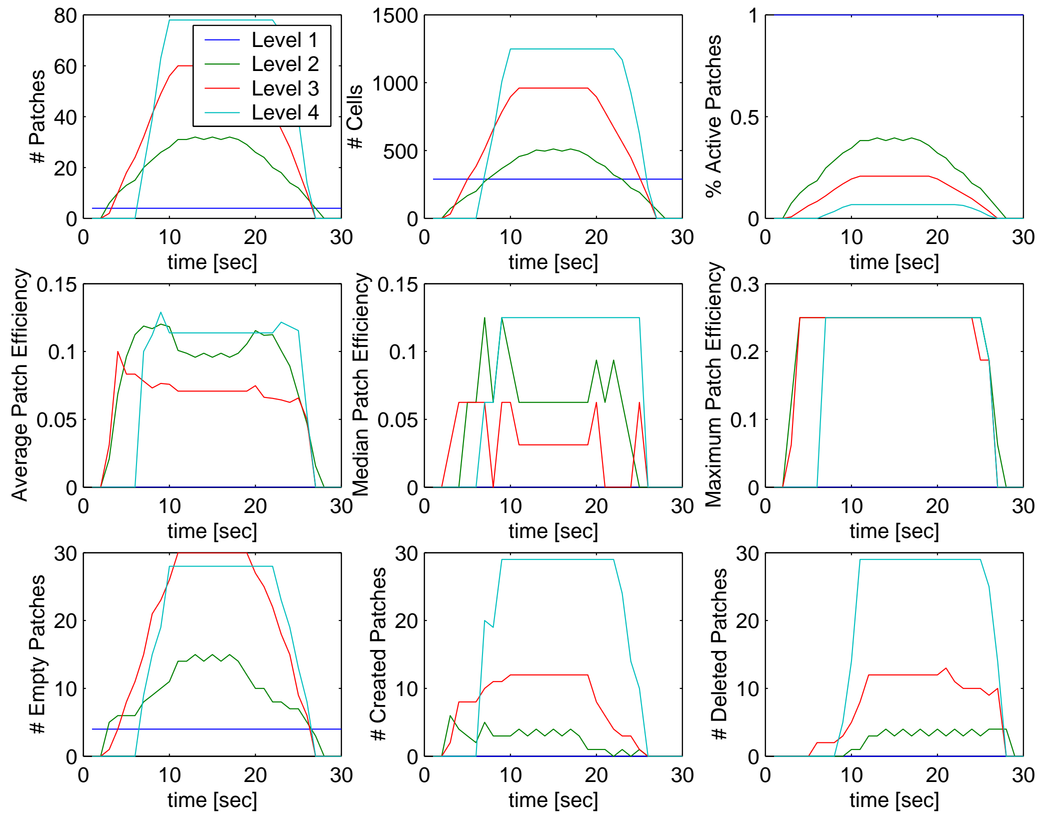
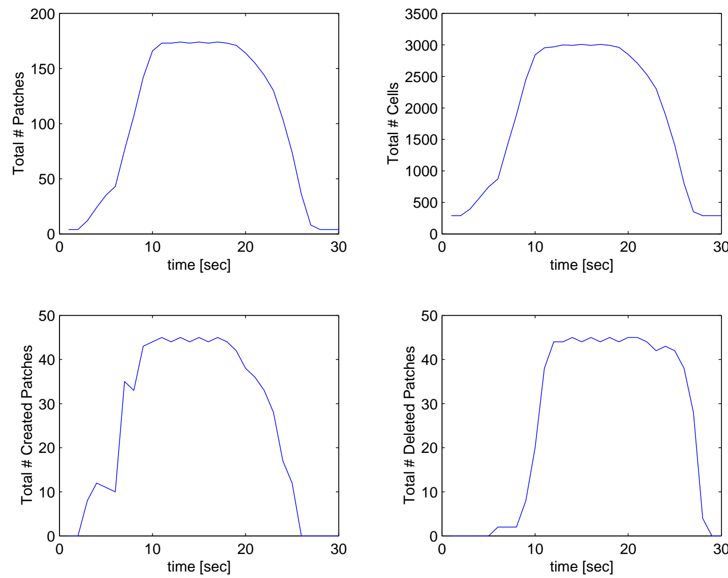


Figure 9: Ball-x test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 5$. (b) $t = 15$. (c) $t = 20$. (d) $t = 25$.



(a)



(b)

Figure 10: Ball-x test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

4.4 Ball Moving in Positive-y

This test case is denoted "Ball-y". It is identical to Ball-x, with the x and y directions roles reversed. The results are indeed exactly as for Ball-x, with x and y reversed.

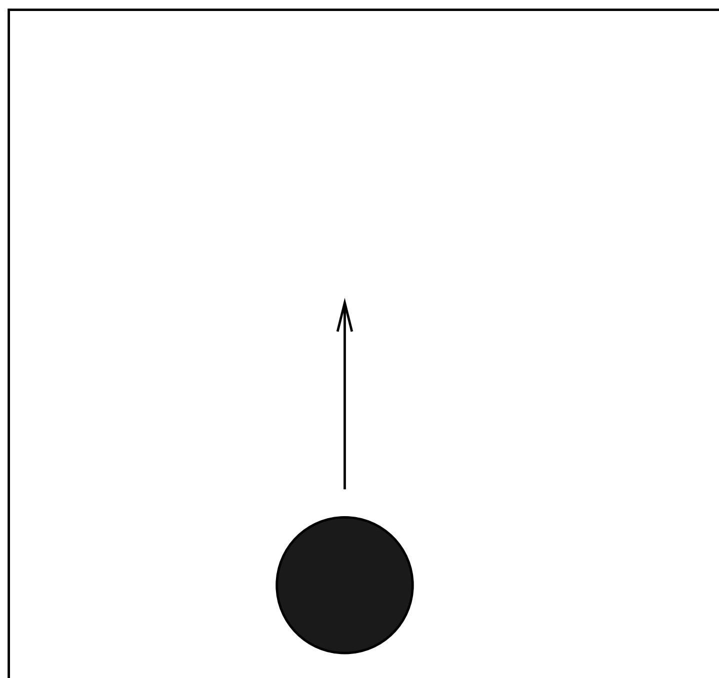


Figure 11: Ball-y test case: a ball moving in positive y-direction.

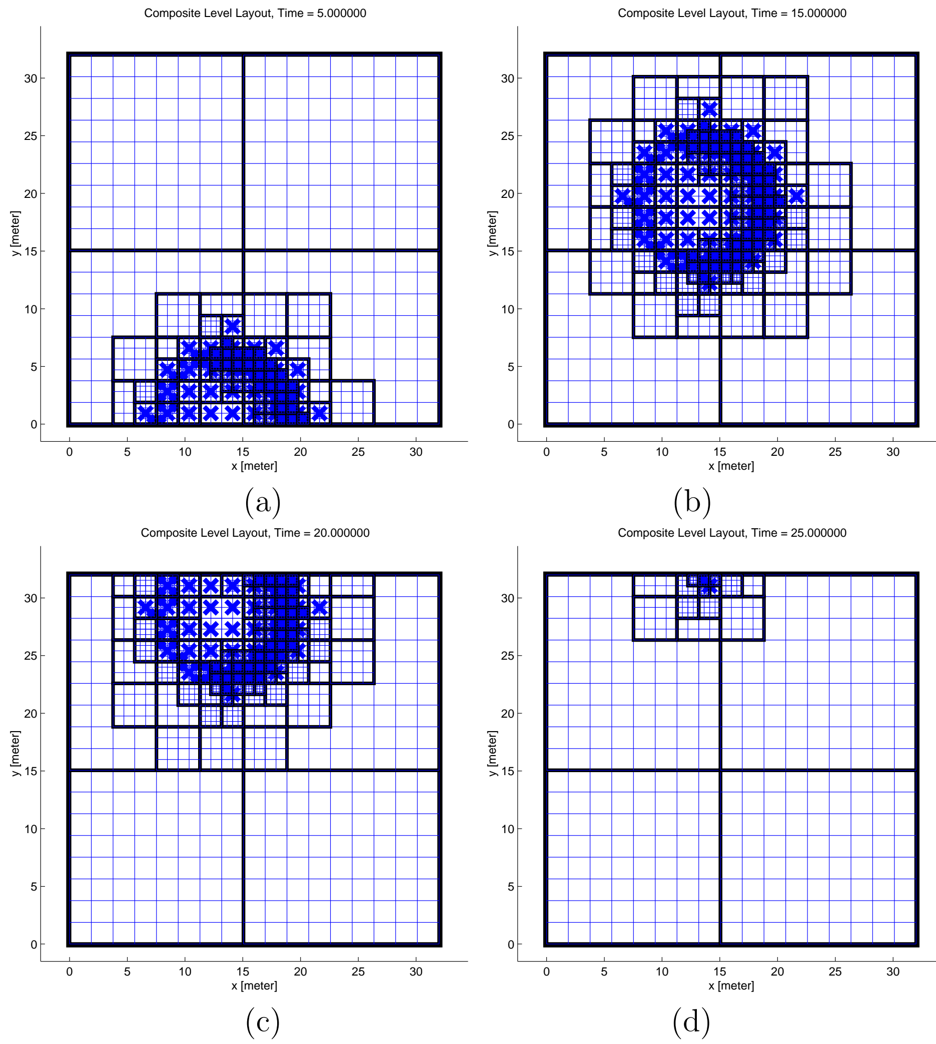
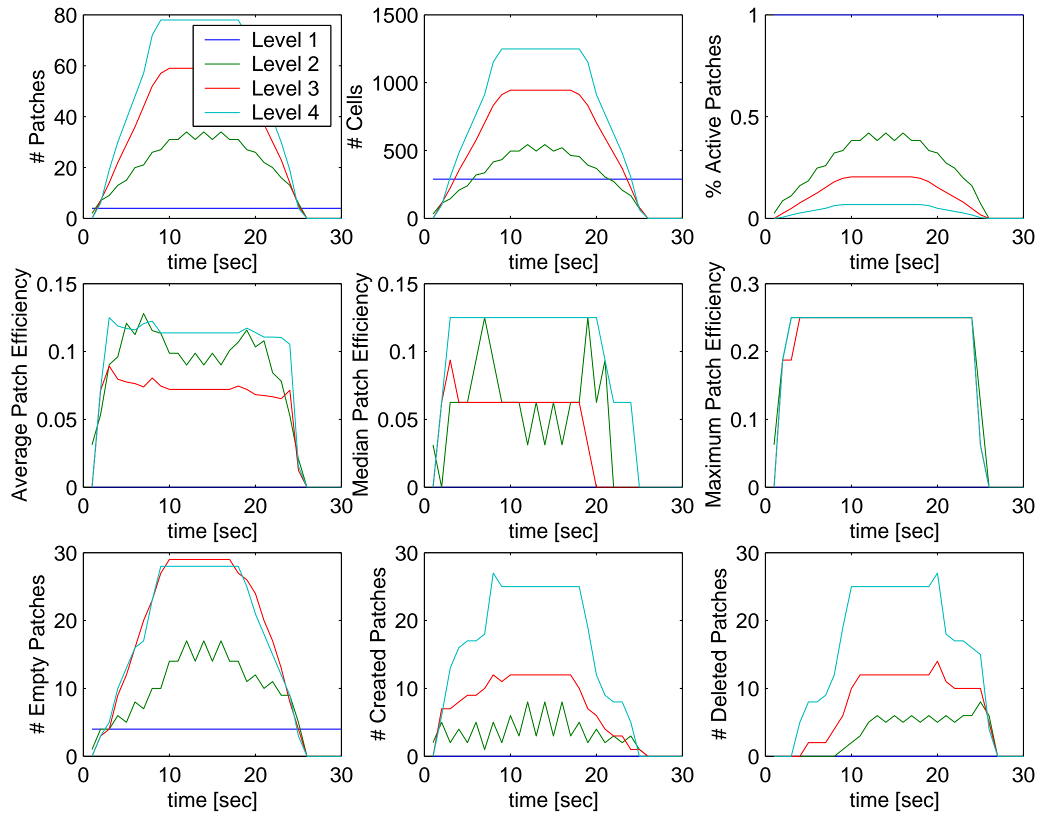
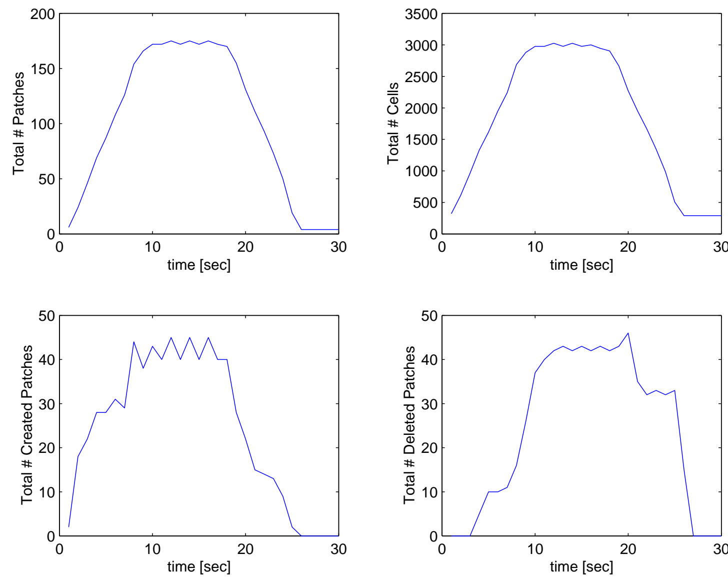


Figure 12: Ball-y test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 5$. (b) $t = 15$. (c) $t = 20$. (d) $t = 25$.



(a)



(b)

Figure 13: Ball-y test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

4.5 Ball Moving in Diagonal

This test case is denoted "Ball-diag". This time, we start at the left bottom part of the domain, and move the ball 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, the patches are very wasteful (they have a lot of empty cells in them). However, the results are as reasonable as we can expect, taking into account this drawback of any rectilinear-patch-based approach.

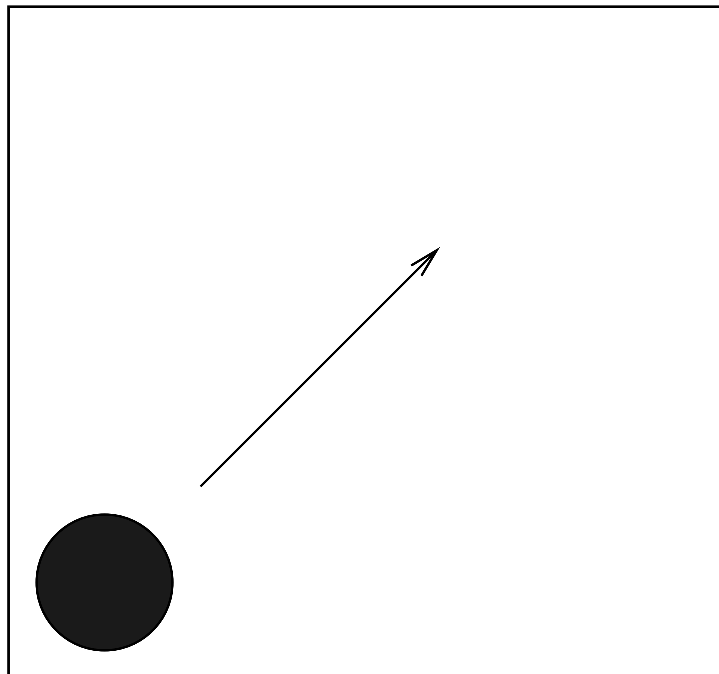


Figure 14: Ball-diag test case: a ball moving in positive y-direction.

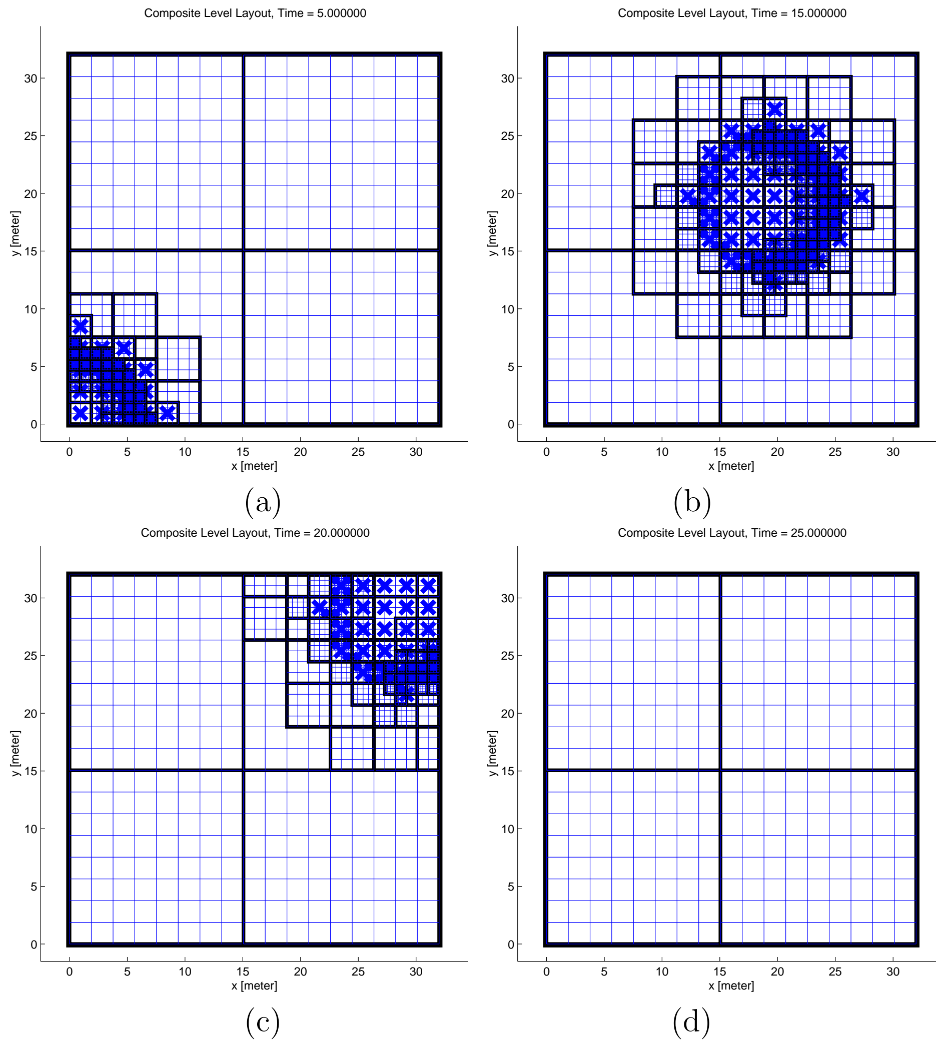
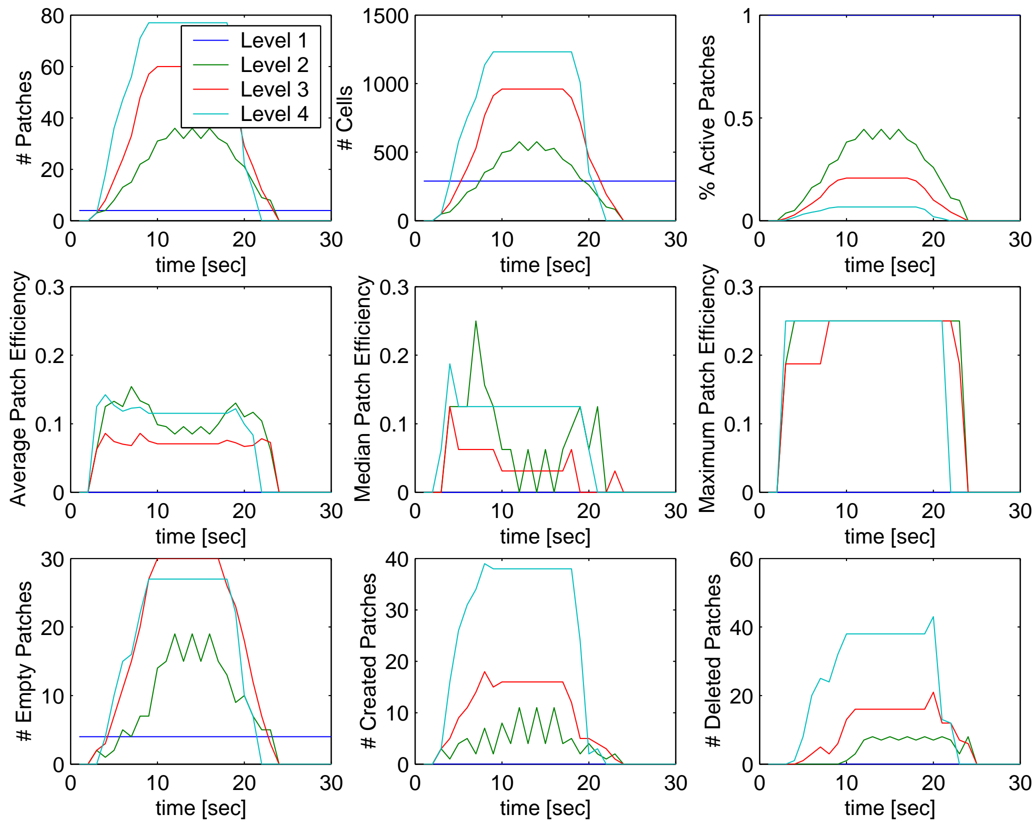
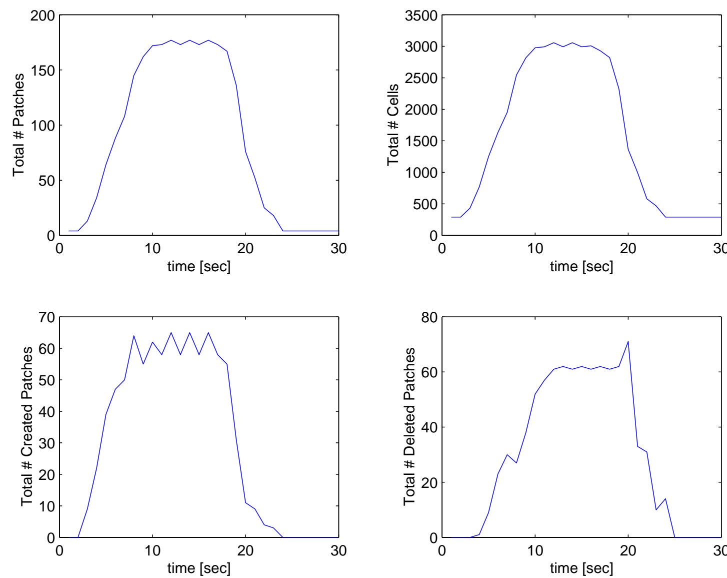


Figure 15: Ball-diag test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 5$. (b) $t = 15$. (c) $t = 20$. (d) $t = 25$.



(a)



(b)

Figure 16: Ball-diag test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

4.6 Ball in a Circular Motion

This test case is denoted “Ball-circ”. We specify a certain circle in the domain (in this example, centered at $(8, 8)$ with radius $R = 8$), and move the ball’s center along this circle. At every time step, we update the angle θ of the ball’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 ball’s location. For instance, we used $\Delta\theta = 0.1 \arccos(1/R) \approx .144$ radians. This is considered a another hard case for griding, because the circular motion does not align with grid-lines.

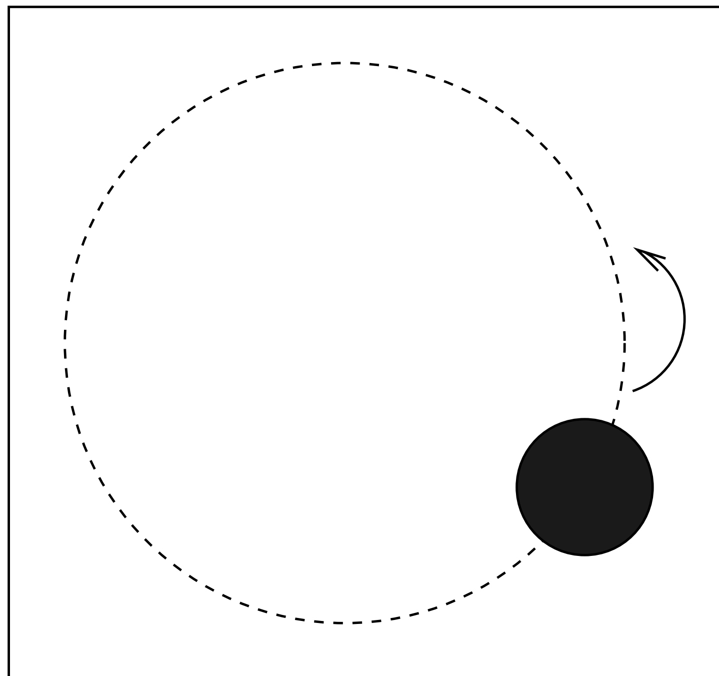


Figure 17: Ball-circ test case: a ball moving in positive y-direction.

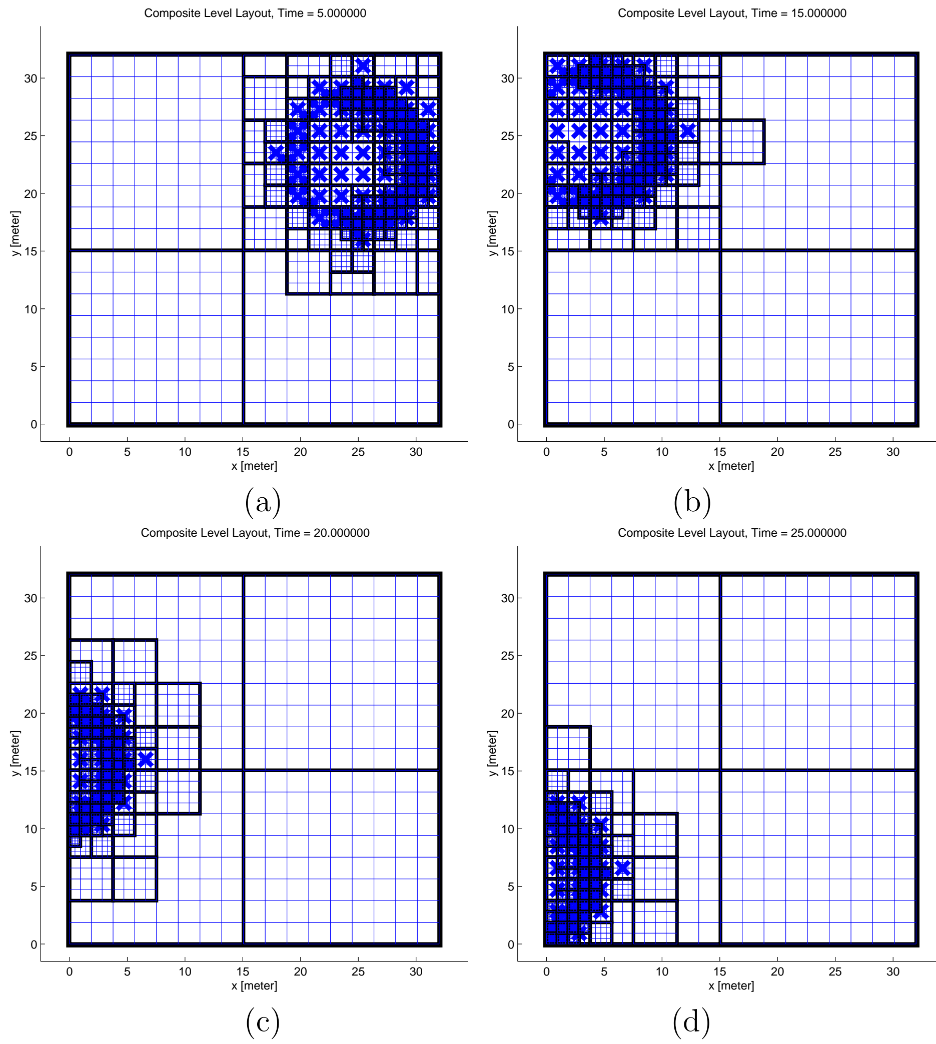
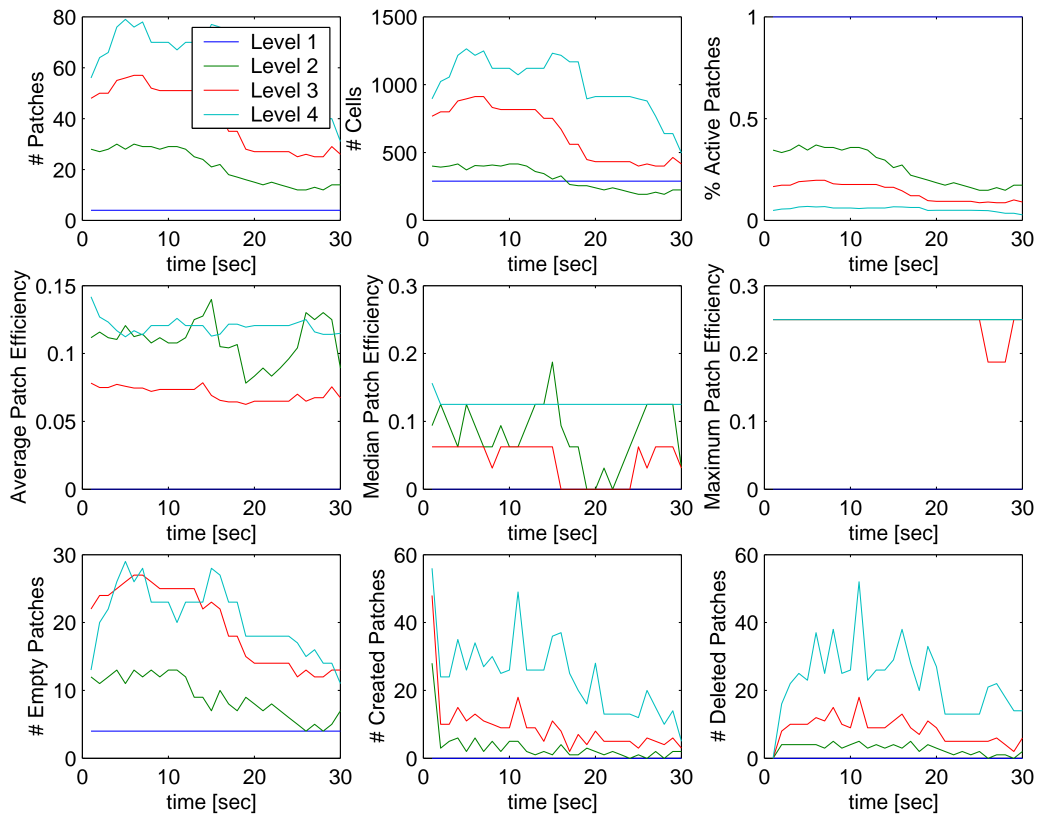
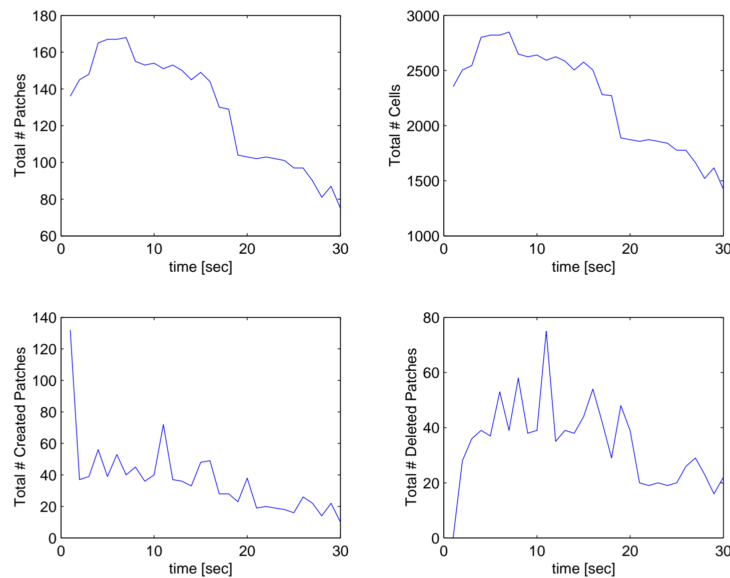


Figure 18: Ball-circ test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 5$. (b) $t = 15$. (c) $t = 20$. (d) $t = 25$.



(a)



(b)

Figure 19: Ball-circ test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

4.7 Colliding Balls

This test case is denoted "Ball-collide". We consider two colliding balls, one moving in positive x-direction, starting on the left of the domain, and a second one that starts on the right of the domain, and moves in negative x-direction. In the region of collision, the patches are observed to be more efficient. Indeed, there is effectively one "local region" in which there exist flagged cells, in that case, as opposed to two separate regions when the balls are apart.

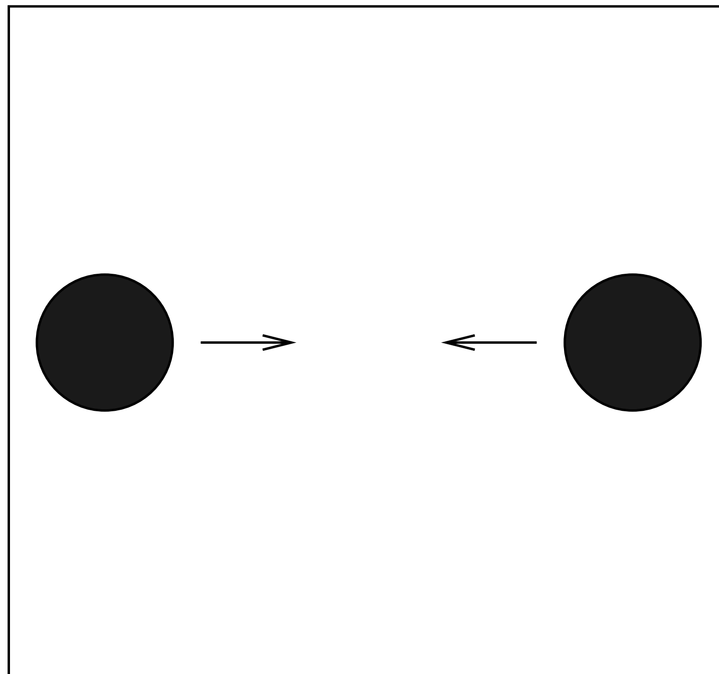


Figure 20: Ball-collide test case: a ball moving in positive y-direction.

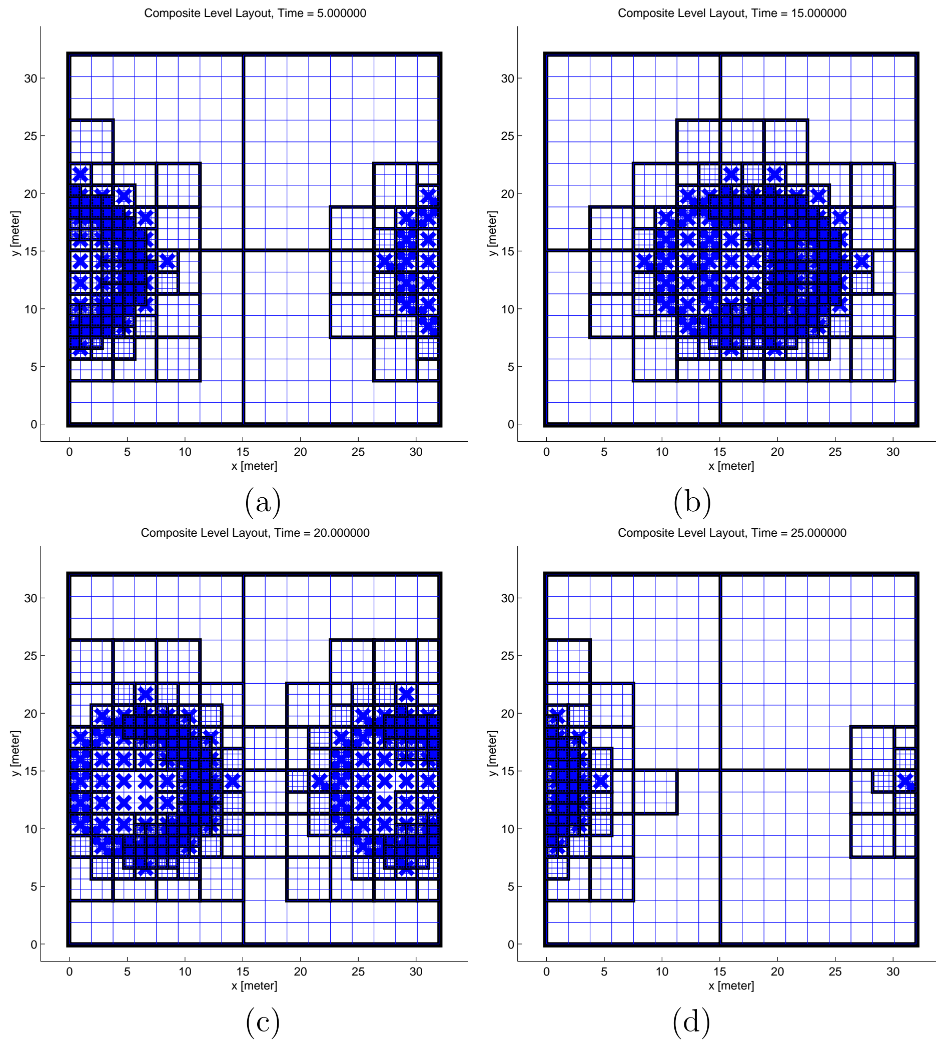
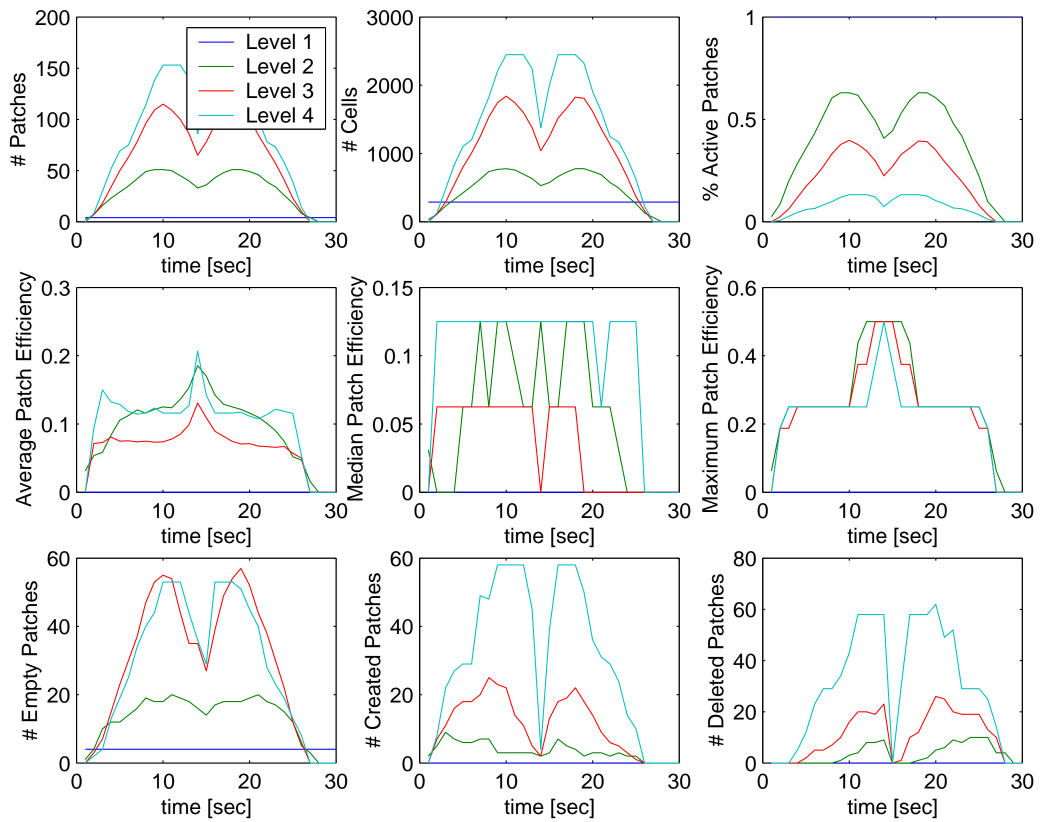
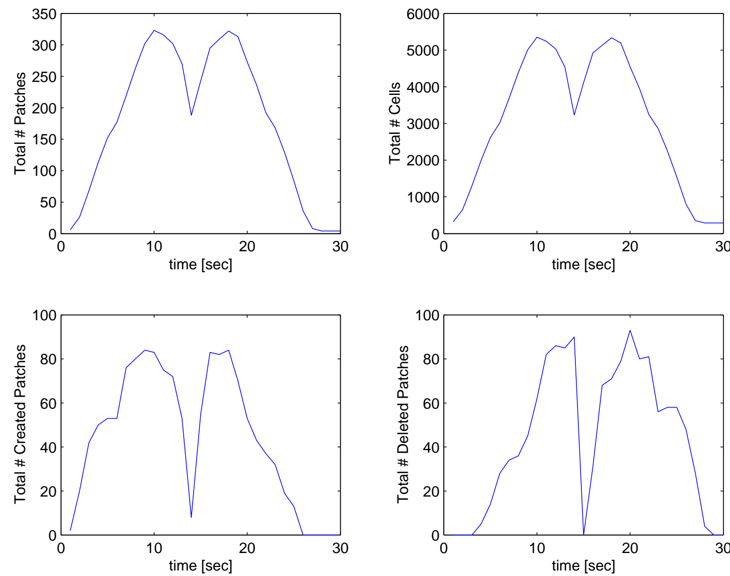


Figure 21: Ball-collide test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 5$. (b) $t = 15$. (c) $t = 20$. (d) $t = 25$.



(a)



(b)

Figure 22: Ball-collide test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

4.8 An Expanding Ball

This test case is denoted "Ball-expand". This is analogous to Ball-x, except that at L_o we only consider an annulus instead of the full ball of flagged cells. Instead of moving its location, this ball expands in radius with time (the radius increases by one cell per timestep). This simulates an expanding shock wave.

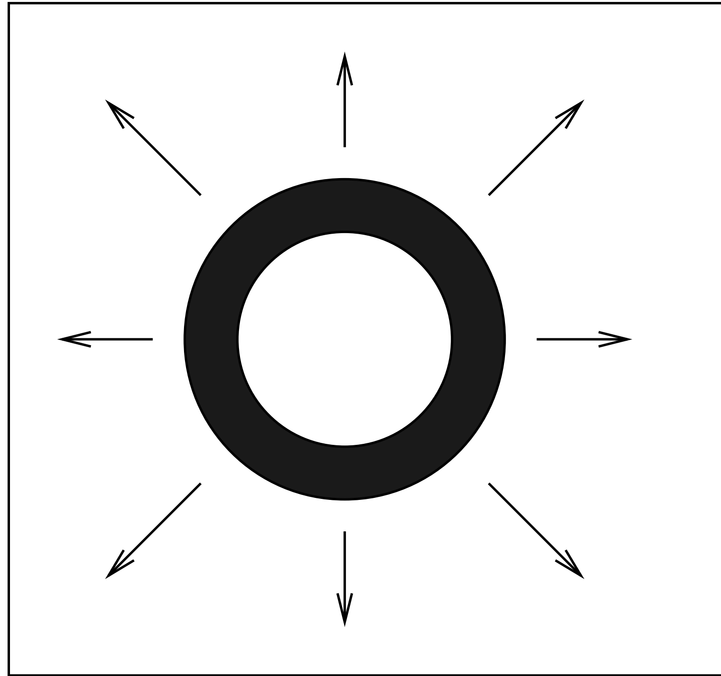


Figure 23: Ball-expand test case: a ball moving in positive y-direction.

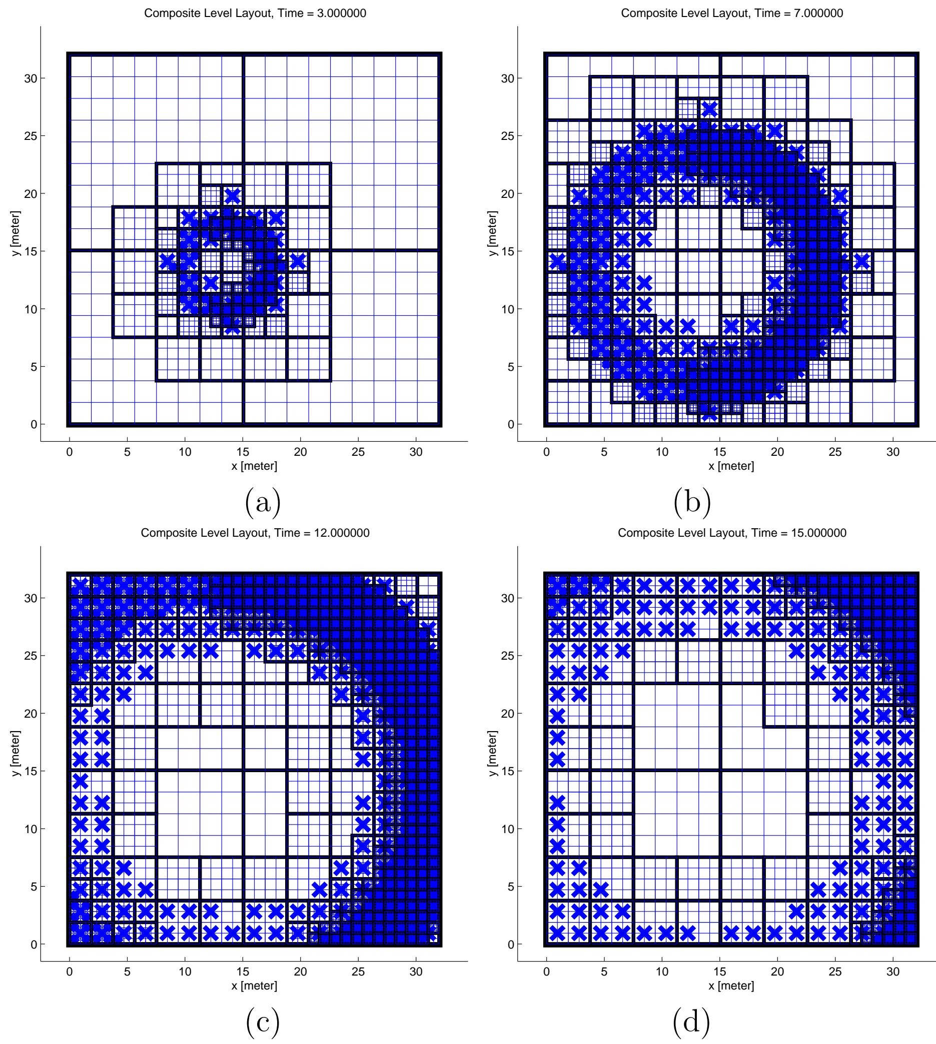
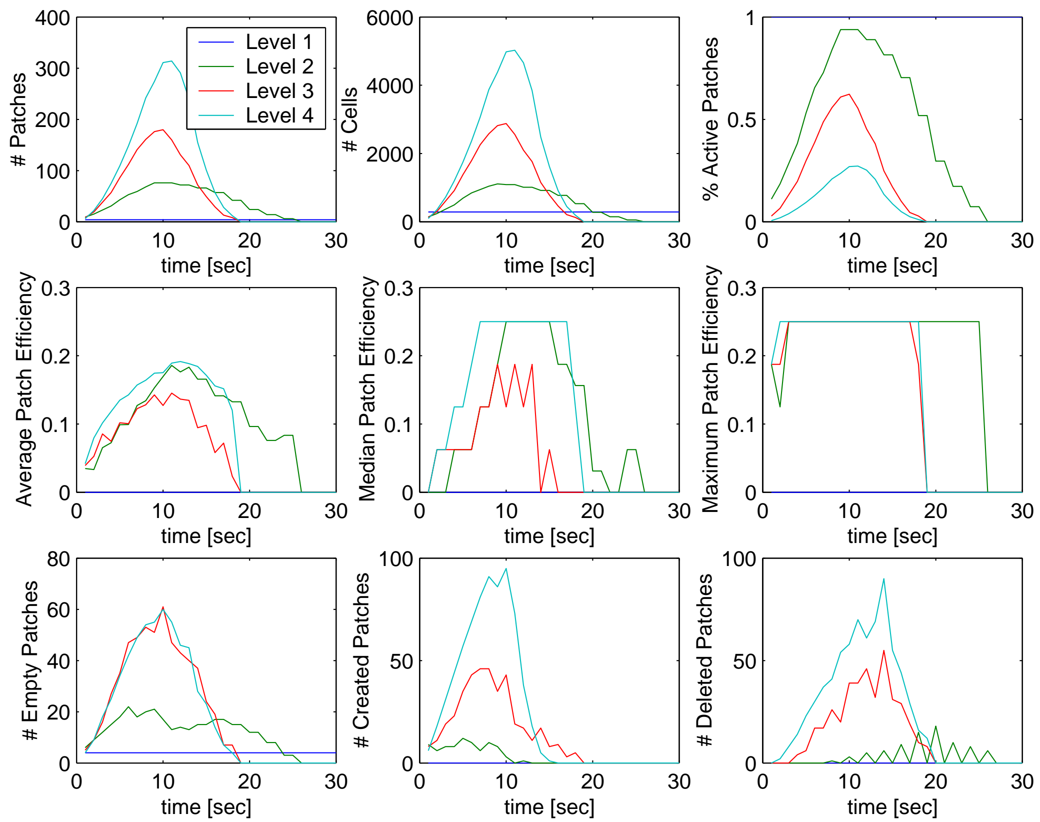
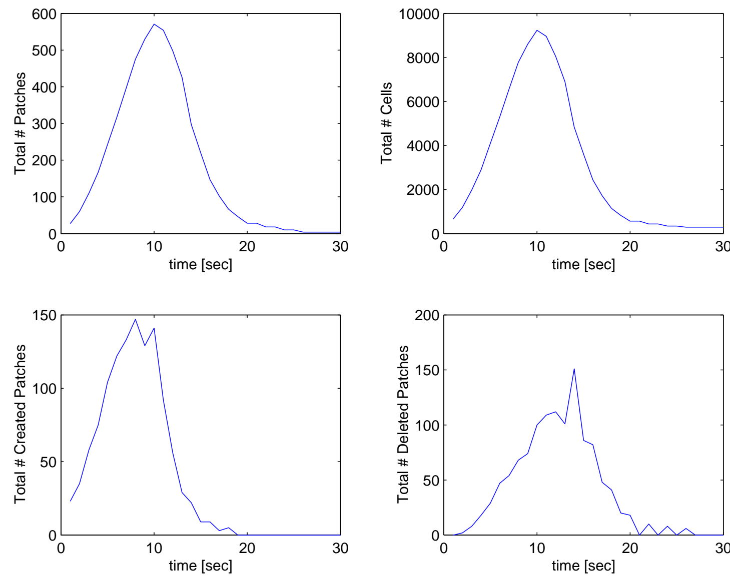


Figure 24: Ball-expand test case: snapshots of the flagged cells (red points) and covering boxes (black or red rectangles). (a) $t = 3$. (b) $t = 7$. (c) $t = 12$. (d) $t = 15$.



(a)



(b)

Figure 25: Ball-expand test case: statistics. (a) Level-dependent indicators. (b) Global indicators.

5 Summary

We developed a generalized version of a hierarchical bisection approach for AMR. Each patch can not only be bisected, but dissected to sub-patches that align with a certain lattice, to better control the computational cost of the entire AMR hierarchy. All the objectives that the AMR hierarchy should satisfy, were attained using the current system: automatic re-griding, reasonably-size patches at all levels, and the safety layers requirements. The user is provided with convenient parameters that control the size of patches and grids.

The algorithm and data structures have been demonstrated on various test cases in 2D. It is easy to generalize them to 3D (in fact, the code is already prepared for a general dimension, except for the auxiliary display/plotting routines).

The complexity of the algorithm does not seem to be large compared with the main ICE computation on the generated grids, but we should monitor the run times of the griding algorithm in the Uintah frameworks.

The next natural step of development seems to be the implementation of this system in the Uintah framework. We can then test it on various practical scenarios, and learn more about its strengths and weaknesses in our applications. Another important possible step is merging adjacent patches to bigger patches before sending them to a processor, if the resulting patch is not too big. In this way, we minimize the number of "too small patches". However, this point might be ignored for the moment, and alternatively treated inside the load balancer.

References

[Ber86] M. J. Berger. Data structures for adaptive grid generation. *SIAM J. Sci. Stat. Comp.*, 7:904–916, 1986.

[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.

[Nee96] H. J. Neeman. *Autonomous hierarchical adaptive mesh refinement for multiscale simulations.* PhD thesis, University of Illinois at Urbana-Champaign, 1996.