Top Math Summer School on

Adaptive Finite Elements: Analysis and Implementation
Organized by: Kunibert G. Siebert

Pedro Morin

Instituto de Matemática Aplicada del Litoral
Universidad Nacional del Litoral

Santa Fe - Argentina

July 28 – August 2, 2008
Frauenchiemsee, Germany

Outline

Regular Refinement

Aim: Given a conforming triangulation $\mathcal{T}_k$ construct a refinement $\mathcal{T}_{k+1}$ of $\mathcal{T}_k$ such that

- some or all elements of $\mathcal{T}_k$ are refined, i. e. decomposed into sub-simplices;
- $\mathcal{T}_{k+1}$ is again conforming and shape-regular.

Regular Refinement

Aim: Given a conforming triangulation $\mathcal{T}_k$ construct a refinement $\mathcal{T}_{k+1}$ of $\mathcal{T}_k$ such that

- some or all elements of $\mathcal{T}_k$ are refined, i. e. decomposed into sub-simplices;
- $\mathcal{T}_{k+1}$ is again conforming and shape-regular.

Regular Refinement: A simplex T is cut into 2^d smaller simplices:

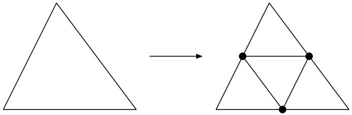
Regular Refinement

Aim: Given a conforming triangulation $\mathcal{T}_k$ construct a refinement $\mathcal{T}_{k+1}$ of $\mathcal{T}_k$ such that

- some or all elements of $\mathcal{T}_k$ are refined, i. e. decomposed into sub-simplices;
- $\mathcal{T}_{k+1}$ is again conforming and shape-regular.

Regular Refinement: A simplex T is cut into 2^d smaller simplices:

2d: 4 congruent triangles



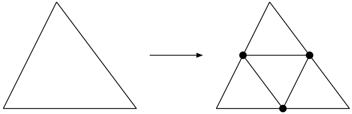
Regular Refinement

Aim: Given a conforming triangulation $\mathcal{T}_k$ construct a refinement $\mathcal{T}_{k+1}$ of $\mathcal{T}_k$ such that

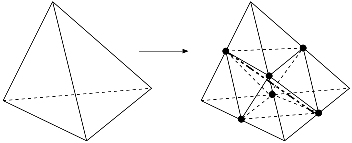
- some or all elements of $\mathcal{T}_k$ are refined, i. e. decomposed into sub-simplices;
- $\mathcal{T}_{k+1}$ is again conforming and shape-regular.

Regular Refinement: A simplex T is cut into 2^d smaller simplices:

2d: 4 congruent triangles



3d: 4 congruent tetrahedra plus 4 additional tetrahedra



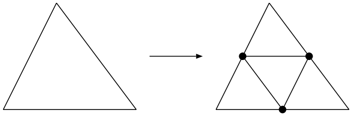
Regular Refinement

Aim: Given a conforming triangulation $\mathcal{T}_k$ construct a refinement $\mathcal{T}_{k+1}$ of $\mathcal{T}_k$ such that

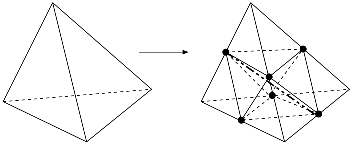
- some or all elements of $\mathcal{T}_k$ are refined, i. e. decomposed into sub-simplices;
- $\mathcal{T}_{k+1}$ is again conforming and shape-regular.

Regular Refinement: A simplex T is cut into 2^d smaller simplices:

2d: 4 congruent triangles



3d: 4 congruent tetrahedra plus 4 additional tetrahedra



Very well suited for global refinement. Induces a very regular structure. This refinement is also called red refinement.

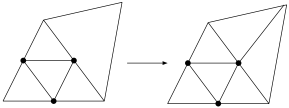
Green Closure

Problems when dealing with local refinement of conforming triangulations: Using only regular refinement leads always to global refinement. Needs an additional refinement rule.

Green Closure

Problems when dealing with local refinement of conforming triangulations: Using only regular refinement leads always to global refinement. Needs an additional refinement rule.

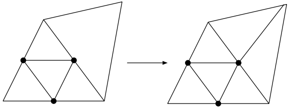
Green closure for removing one single hanging node in 2d:



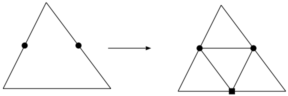
Green Closure

Problems when dealing with local refinement of conforming triangulations: Using only regular refinement leads always to global refinement. Needs an additional refinement rule.

Green closure for removing one single hanging node in 2d:



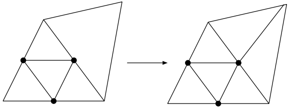
Regular refinement for removing more than one hanging node in 2d, may create new hanging nodes



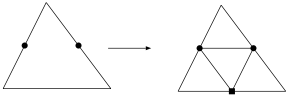
Green Closure

Problems when dealing with local refinement of conforming triangulations: Using only regular refinement leads always to global refinement. Needs an additional refinement rule.

Green closure for removing one single hanging node in 2d:



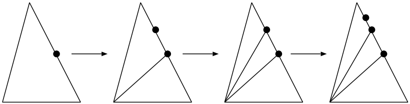
Regular refinement for removing more than one hanging node in 2d, may create new hanging nodes



In 3d, there are several refinement rules, depending on the number and location of the hanging nodes (bisection, cutting into 4 or 8).

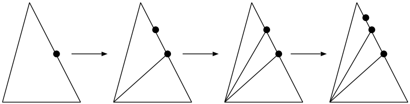
Removing of the Green Closure

We have to remove the green closure before refining a "special" element again. Without "undo"-step shape-regularity of the sequence $\{\mathcal{T}_k\}_{k\geq 0}$ can not ensured:

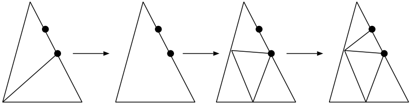


Removing of the Green Closure

We have to remove the green closure before refining a "special" element again. Without "undo"-step shape-regularity of the sequence $\{\mathcal{T}_k\}_{k\geq 0}$ can not ensured:



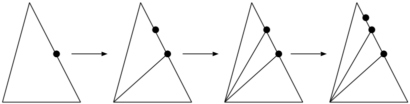
Refinement of a "special" element:



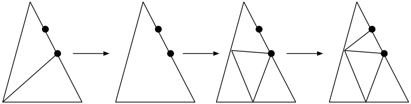
This is a very elaborate procedure ...

Removing of the Green Closure

We have to remove the green closure before refining a “special” element again. Without “undo”-step shape-regularity of the sequence $\{\mathcal{T}_k\}_{k\geq 0}$ can not ensured:



Refinement of a “special” element:



This is a very elaborate procedure . . .

Without bisectioning of elements, local refinement is impossible.

Refinement by Bisection

Use a refinement procedure that only relies on bisectioning of elements.

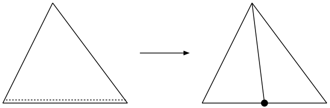
Problem: Avoid degenerate situations, i. e. avoid that elements are getting flat.

Refinement by Bisection

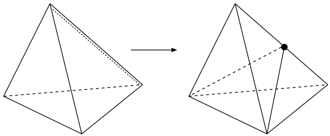
Use a refinement procedure that only relies on bisectioning of elements.

Problem: Avoid degenerate situations, i. e. avoid that elements are getting flat.

Solution: Assign to all elements of the initial triangulation $\mathcal{T}_0$ a refinement edge.
An element is always bisected by inserting a new vertex in the midpoint of the refinement edge:



Refinement of a triangle



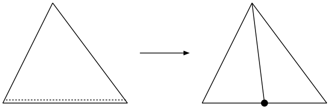
Refinement of a tetrahedron

Refinement by Bisection

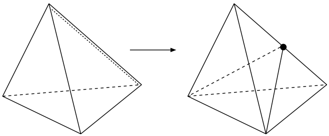
Use a refinement procedure that only relies on bisectioning of elements.

Problem: Avoid degenerate situations, i. e. avoid that elements are getting flat.

Solution: Assign to all elements of the initial triangulation $\mathcal{T}_0$ a refinement edge.
An element is always bisected by inserting a new vertex in the midpoint of the refinement edge:



Refinement of a triangle



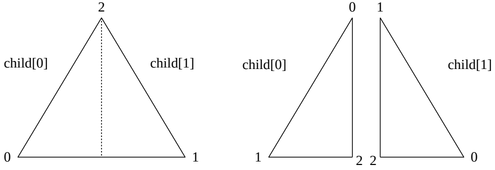
Refinement of a tetrahedron

The algorithm then assigns the refinement edges of the two children such that shape regularity is preserved.

Assignment of Refinement Edges in 2d

Local Numbering of the element's vertices such that the refinement edge is the edge between vertex 0 and 1.

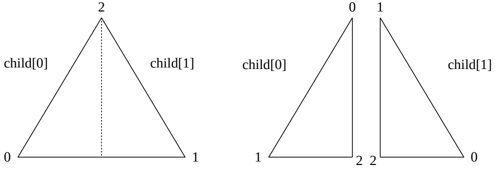
Newest Vertex Bisection: Numbering of nodes on parent and children in 2d:



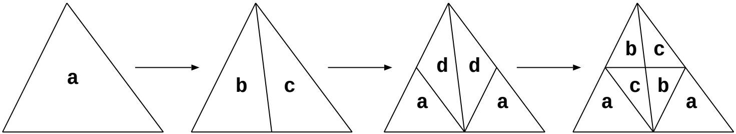
Assignment of Refinement Edges in 2d

Local Numbering of the element's vertices such that the refinement edge is the edge between vertex 0 and 1.

Newest Vertex Bisection: Numbering of nodes on parent and children in 2d:



Successive refinement of a single triangle leads to at most 4 equivalence classes:



Assignment of Refinement Edges in 3d

The assignment is more complicated in 3d and depends on the element type that is a number in $\{0,1,2\}$

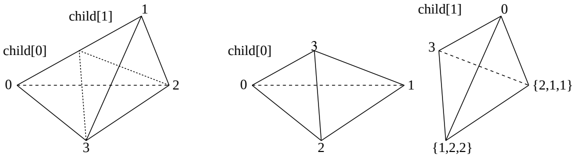
- The type can be freely assigned for all elements of $\mathcal{T}_0$.
- The type of a child is (parent's type $+$ 1) modulo 3.

Assignment of Refinement Edges in 3d

The assignment is more complicated in 3d and depends on the element type that is a number in $\{0,1,2\}$

- The type can be freely assigned for all elements of $\mathcal{T}_0$.
- The type of a child is (parent's type + 1) modulo 3.

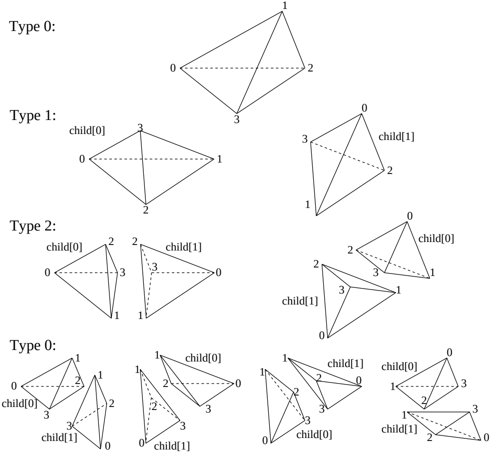
Numbering of nodes on parent and children in 3d:



Convention: For the index set $\{1,2,2\}$ on `child[1]` of a tetrahedron of type 0 we use the index 1 and for a tetrahedron of type 1 and 2 the index 2.

[Kossaczký] and also [Bänsch] and [Maubach]

Successive Refinements of a Type 0 Tetrahedron



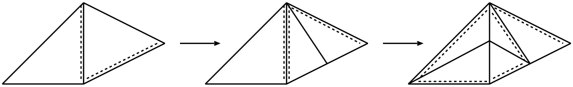
Idea of Recursive Refinement

Problem. In general, two neighboring elements will not share a common refinement edge.

Idea of Recursive Refinement

Problem. In general, two neighboring elements will not share a common refinement edge.

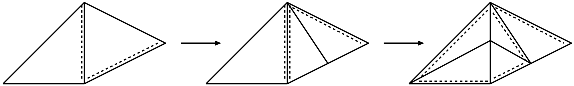
Idea. If two neighboring elements do not share the same refinement edge, recursively refine first the neighbor. Afterwards, the refinement edge is shared.



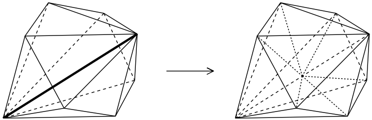
Idea of Recursive Refinement

Problem. In general, two neighboring elements will not share a common refinement edge.

Idea. If two neighboring elements do not share the same refinement edge, recursively refine first the neighbor. Afterwards, the refinement edge is shared.



This also works in 3d but involves all elements at the common edge and may need several recursive refinements of neighbors before performing the atomic refinement operation.



Avoiding hanging nodes in 3d is even more convenient than in 2d!

Recursive Refinement of a Single Element

The recursive refinement of a single element reads:

subroutine recursive_refine(T, $\mathcal{T}$)

$\mathcal{A} := \{T' \in \mathcal{T};\, T'$ is not compatibly divisible with $T\}$

for all $T' \in \mathcal{A}$ do

recursive_refine(T', $\mathcal{T}$);

end for

Recursive Refinement of a Single Element

The recursive refinement of a single element reads:

```
subroutine recursive_refine(T, 𝒯)
  do
    𝒜 := {T′ ∈ 𝒯; T′ is not compatibly divisible with T}
    for all T′ ∈ 𝒜 do
      recursive_refine(T′, 𝒯);
    end for
    𝒜 := {T′ ∈ 𝒯; T′ is not compatibly divisible with T}
  until 𝒜 = ∅
```

Recursive Refinement of a Single Element

The recursive refinement of a single element reads:

```
subroutine recursive_refine(T, 𝒯)
  do
    𝒜 := {T' ∈ 𝒯; T' is not compatibly divisible with T}
    for all T' ∈ 𝒜 do
      recursive_refine(T', 𝒯);
    end for
    𝒜 := {T' ∈ 𝒯; T' is not compatibly divisible with T}
  until 𝒜 = ∅

  𝒜 := {T' ∈ 𝒯; T' is element at the refinement edge of T}
  for all T' ∈ 𝒜
    bisect T' into T'_0 and T'_1
    𝒯 := 𝒯\{T'} ∪ {T'_0, T'_1}
  end for
```

Recursive Refinement of a Triangulation

Denote by $\mathcal{M} \subset \mathcal{T}$ the set of elements marked for refinement. The recursive refinement algorithm reads:

```
subroutine refine(𝒯)
   for all T ∈ ℳ do
      recursive_refine(T, 𝒯)
   end for
```

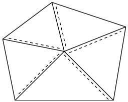
Recursive Refinement of a Triangulation

Denote by $\mathcal{M} \subset \mathcal{T}$ the set of elements marked for refinement. The recursive refinement algorithm reads:

```
subroutine refine(𝒯)
  for all T ∈ ℳ do
    recursive_refine(T, 𝒯)
  end for
```

Remarks.

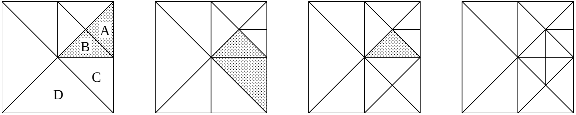
- The recursion terminates on any level iff the initial grid $\mathcal{T}_0$ fulfills certain criteria. A necessary condition is that there is no cycle on $\mathcal{T}_0$.



In 2d, this condition is also sufficient.

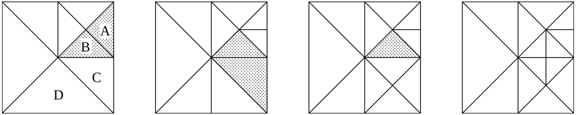
Remarks (Continued)

- Recursion is needed (Triangles A and B were initially marked):



Remarks (Continued)

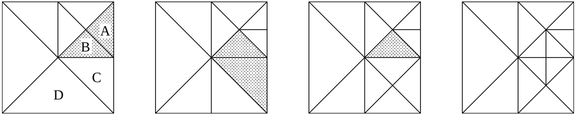
- Recursion is needed (Triangles A and B were initially marked):



- Simplices initially not marked for refinement are bisected, enforced by the refinement of a marked simplex. This is a necessity to obtain a conforming triangulation.

Remarks (Continued)

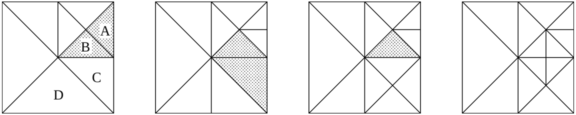
- Recursion is needed (Triangles A and B were initially marked):



- Simplices initially not marked for refinement are bisected, enforced by the refinement of a marked simplex. This is a necessity to obtain a conforming triangulation.
- Usually, a marked element is bisected more than once, where the natural choice are d bisections. Then all edges of the element are bisected.

Remarks (Continued)

- Recursion is needed (Triangles A and B were initially marked):



- Simplices initially not marked for refinement are bisected, enforced by the refinement of a marked simplex. This is a necessity to obtain a conforming triangulation.
- Usually, a marked element is bisected more than once, where the natural choice are d bisections. Then all edges of the element are bisected.
- The assignment of the refinement edges of children by the above algorithm ensures shape-regularity for the sequence of triangulations

$$\sup_{k\geq 0} \max_{T\in\mathcal{T}_k} \overline{h_T}/\underline{h_T} \leq C\left(\mathcal{T}_0\right) < \infty.$$

Complexity of Refinement

Assume a given triangulation $\mathcal{T}_k$ together with a set $\mathcal{M}_k$ of elements marked for refinement. Conformity of $\mathcal{T}_{k+1}$ in general implies that we refine also elements $T \in \mathcal{T}_k \setminus \mathcal{M}_k$.

Complexity of Refinement

Assume a given triangulation $\mathcal{T}_k$ together with a set $\mathcal{M}_k$ of elements marked for refinement. Conformity of $\mathcal{T}_{k+1}$ in general implies that we refine also elements $T \in \mathcal{T}_k \setminus \mathcal{M}_k$.

Important Question. Is there a bound

$$\#\mathcal{T}_{k+1} \le \#\mathcal{T}_k + C\,\#\mathcal{M}_k? \qquad \text{or} \qquad \#\mathcal{T}_{k+1} - \#\mathcal{T}_k \le C\,\#\mathcal{M}_k?$$

Complexity of Refinement

Assume a given triangulation $\mathcal{T}_k$ together with a set $\mathcal{M}_k$ of elements marked for refinement. Conformity of $\mathcal{T}_{k+1}$ in general implies that we refine also elements $T \in \mathcal{T}_k \setminus \mathcal{M}_k$.

Important Question. Is there a bound

$$\#\mathcal{T}_{k+1} \leq \#\mathcal{T}_k + C\,\#\mathcal{M}_k? \qquad \text{or} \qquad \#\mathcal{T}_{k+1} - \#\mathcal{T}_k \leq C\,\#\mathcal{M}_k?$$

In general not, since refinement can spread drastically in one single step! But on the other hand, refinement by bisection creates local structures.

Complexity of Refinement

Assume a given triangulation $\mathcal{T}_k$ together with a set $\mathcal{M}_k$ of elements marked for refinement. Conformity of $\mathcal{T}_{k+1}$ in general implies that we refine also elements $T \in \mathcal{T}_k \setminus \mathcal{M}_k$.

Important Question. Is there a bound

$$\#\mathcal{T}_{k+1} \le \#\mathcal{T}_k + C\,\#\mathcal{M}_k? \qquad \text{or} \qquad \#\mathcal{T}_{k+1} - \#\mathcal{T}_k \le C\,\#\mathcal{M}_k?$$

In general not, since refinement can spread drastically in one single step! But on the other hand, refinement by bisection creates local structures.

Theorem (Complexity of Refinement). A proper choice of refinement edges on the initial triangulation $\mathcal{T}_0$ implies

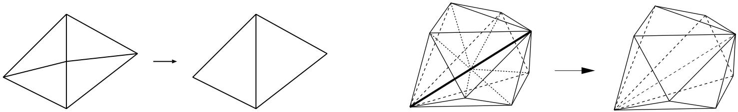
$$\#\mathcal{T}_{k+1} - \#\mathcal{T}_0 \le C_0 \sum_{n=0}^{k} \#\mathcal{M}_n.$$

[Binev,Dahmen,DeVore] in 2d and [Stevenson] in any dimension.

Coarsening of Elements

Coarsening is mainly the inverse operation to refinement with the following restriction:

Collect all children that were created in one atomic refinement operation. If all these children are of finest level and if all children are marked for coarsening, undo the atomic refinement operation.



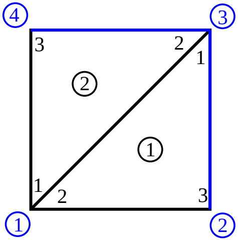
Atomic coarsening operation in 2d and 3d

Outline

A simple 2d implementation in OCTAVE

We will use a data structure for the mesh, similar to the previous one, but with adjacency and boundary information, as follows:

- `mesh.vertex_coordinates` as before
- `mesh.element_vertices` as before
- `mesh.element_neighbours` with information about the neighbours
- `mesh.element_boundary` with boundary type information



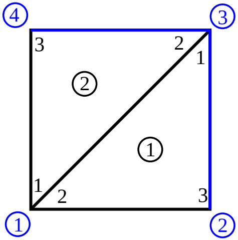
A simple 2d implementation in OCTAVE

We will use a data structure for the mesh, similar to the previous one, but with adjacency and boundary information, as follows:

- mesh.vertex_coordinates as before
- mesh.element_vertices as before
- mesh.element_neighbours with information about the neighbours
- mesh.element_boundary with boundary type information

mesh.vertex_coordinates

```
0.0  0.0
1.0  0.0
1.0  1.0
0.0  1.0
```



A simple 2d implementation in OCTAVE

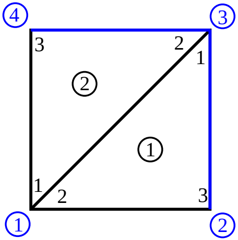
We will use a data structure for the mesh, similar to the previous one, but with adjacency and boundary information, as follows:

- `mesh.vertex_coordinates` as before
- `mesh.element_vertices` as before
- `mesh.element_neighbours` with information about the neighbours
- `mesh.element_boundary` with boundary type information

`mesh.elem_vertices`

In counter clockwise order.
Refinement edge first.

3	1	2
1	3	4



A simple 2d implementation in OCTAVE

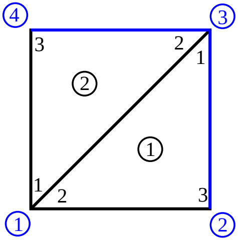
We will use a data structure for the mesh, similar to the previous one, but with adjacency and boundary information, as follows:

- `mesh.vertex_coordinates` as before
- `mesh.element_vertices` as before
- `mesh.element_neighbours` with information about the neighbours
- `mesh.element_boundary` with boundary type information

`mesh.elem_neighbours`

Neighbouring elements. The i-th side is the one opposite to the local i-th vertex. Boundary sides are marked with 0.

```
0 0 2
0 0 1
```



A simple 2d implementation in OCTAVE

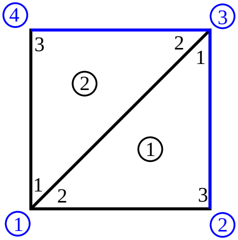
We will use a data structure for the mesh, similar to the previous one, but with adjacency and boundary information, as follows:

- `mesh.vertex_coordinates` as before
- `mesh.element_vertices` as before
- `mesh.element_neighbours` with information about the neighbours
- `mesh.element_boundary` with boundary type information

mesh.elem_boundaries

Boundary types. Dirichlet: `1`, Neumann: `-1`, Interior: `0`.

```
-1  1  0
 1 -1  0
```



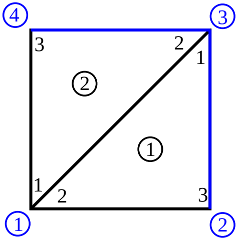
A simple 2d implementation in OCTAVE

We will use a data structure for the mesh, similar to the previous one, but with adjacency and boundary information, as follows:

- `mesh.vertex_coordinates` as before
- `mesh.element_vertices` as before
- `mesh.element_neighbours` with information about the neighbours
- `mesh.element_boundary` with boundary type information

Mesh Generation

So far, by hand.
One can use mesh generators like `triangle` and then convert them into this format.



`refine_mesh`

This function receives a mesh with marked elements and calls `refine_element` in order to refine them.

- `mesh.mark` is a vector of length `mesh.n_elem` indicating the number of bisections that should be done to each element

refine_mesh

This function receives a mesh with marked elements and calls `refine_element` in order to refine them.

```
function n_refined = refine_mesh

% we use global variables to save memory space
global mesh uh fh

n_refined = 0;

if (max(mesh.mark)==0)
  % no elements marked, doing nothing
  return
end

while (max(mesh.mark) > 0)
  first_marked = min(find(mesh.mark > 0));
  n_refined = n_refined + refine_element(first_marked);
end
```

refine_element

This function receives an element to refine. Checks if the corresponding neighbour is compatibly divisible or if the refinement edge is at the boundary. If this is so, proceeds to the atomic refinement. Otherwise, calls recursively refine_element in order to refine the neighbour.

```
function n_refined = refine_element(elem_to_refine)

global mesh uh fh

n_refined = 0;

neighbour = mesh.elem_neighbours(elem_to_refine, 3);

if (neighbour > 0) % not a boundary side
  if (mesh.elem_neighbours(neighbour, 3) != elem_to_refine)
    % not compatible
    n_refined = n_refined + refine_element(neighbour);
  end
end
% now the neighbour is compatible and comes the refinement
```

refine_element

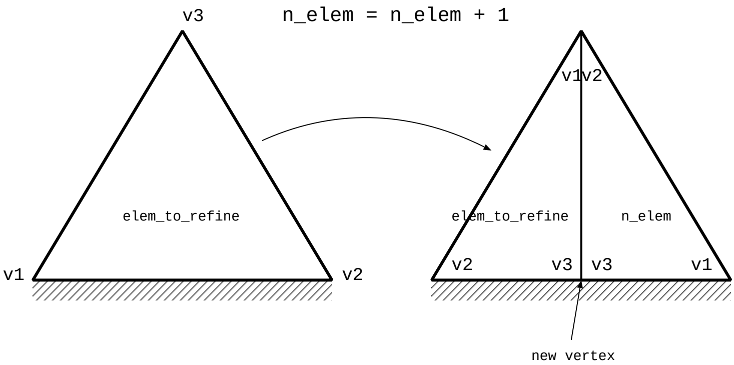
```
% now the neighbour is compatible and comes the refinement
neighbour = mesh.elem_neighbours(elem_to_refine, 3);

v_elem = mesh.elem_vertices(elem_to_refine,:);
neighs_elem = mesh.elem_neighbours(elem_to_refine,:);
bdries_elem = mesh.elem_boundaries(elem_to_refine,:);

n_vertices = n_vertices + 1;
mesh.vertex_coordinates(n_vertices, :) ...
 = 0.5*(mesh.vertex_coordinates(v_elem(1),:)
        + mesh.vertex_coordinates(v_elem(2),:));
uh(n_vertices, :) = 0.5*(uh(v_elem(1), :) + uh(v_elem(2), :));
fh(n_vertices, :) = 0.5*(fh(v_elem(1), :) + fh(v_elem(2), :));

[...]
```

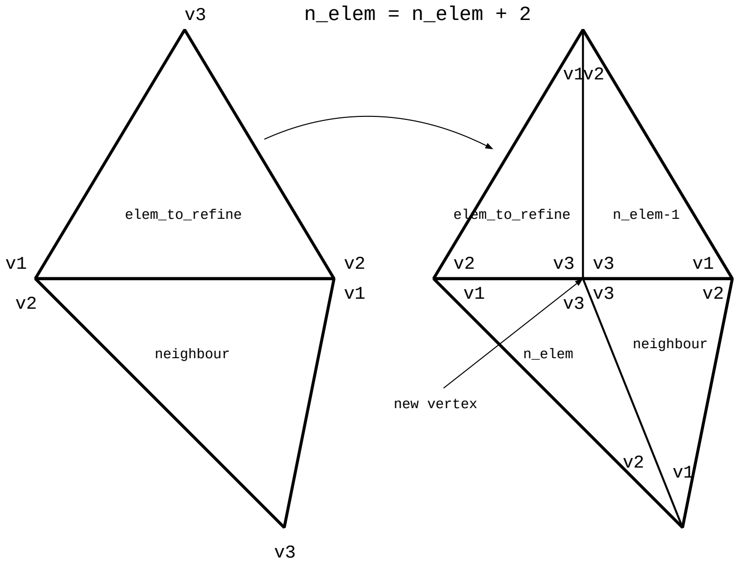
refine_element



Besides defining the new elements we must:

- Set the neighbours and boundary information of the two new elements.
- Set the neighbours of the old neighbours of the elem_to_refine

refine_element



Exercise 4

Read the mesh from the folder adapt/square_all_dirichlet with the following lines (you can copy them from adapt/afem.m):

```
domain = 'square_all_dirichlet';

global mesh uh fh

read the mesh from 'domain'
mesh = struct();
mesh.elem_vertices       = load([domain '/elem_vertices.txt']);
mesh.elem_neighbours     = load([domain '/elem_neighbours.txt']);
mesh.elem_boundaries     = load([domain '/elem_boundaries.txt']);
mesh.vertex_coordinates = load([domain '/vertex_coordinates.txt']);
mesh.n_elem     = size(mesh.elem_vertices, 1);
mesh.n_vertices = size(mesh.vertex_coordinates, 1);

uh = zeros(mesh.n_elem,1); fh = zeros(mesh.n_elem,1);
```

Exercise 4 (continued)

Now repeat the following two steps N times:

- Mark the elements touching the origin for one refinement (remember that vertex of the mesh corresponding to the origin will always have the same index). For this you have to set `mesh.mark` to all zeros and then put one on the elements that contain the origin as a vertex (check usage of `find`).
- Refine the mesh (using `refine_mesh`).

After this, mark (for one refinement) the first element to the right of the lowest element that touches the origin. Now refine the mesh, and check how many elements needed to be refined.
Repeat the exercise (from scratch) for $N = 5, 10, 15$, and convince yourself that it is not possible to obtain a bound as

$$\#\mathcal{T}_{k+1} - \#\mathcal{T}_k \leq C\mathcal{M}_k.$$