Construction of general hierarchical matrices for multi-dimensional problems

Steffen Börm

Christian-Albrechts-Universität, Kiel

Winterschool on Hierarchical Matrices

General cluster tree

Goal: Cluster tree $\mathcal{T}_\mathcal{I}$ for general index set $\mathcal{I}$.

Definition: A labeled tree $\mathcal{T}_\mathcal{I}$ is a cluster tree for $\mathcal{I}$ if

- its root is labeled by $\mathcal{I}$,
- if $t \in \mathcal{T}_\mathcal{I}$ has sons, its label is the union of its sons' labels

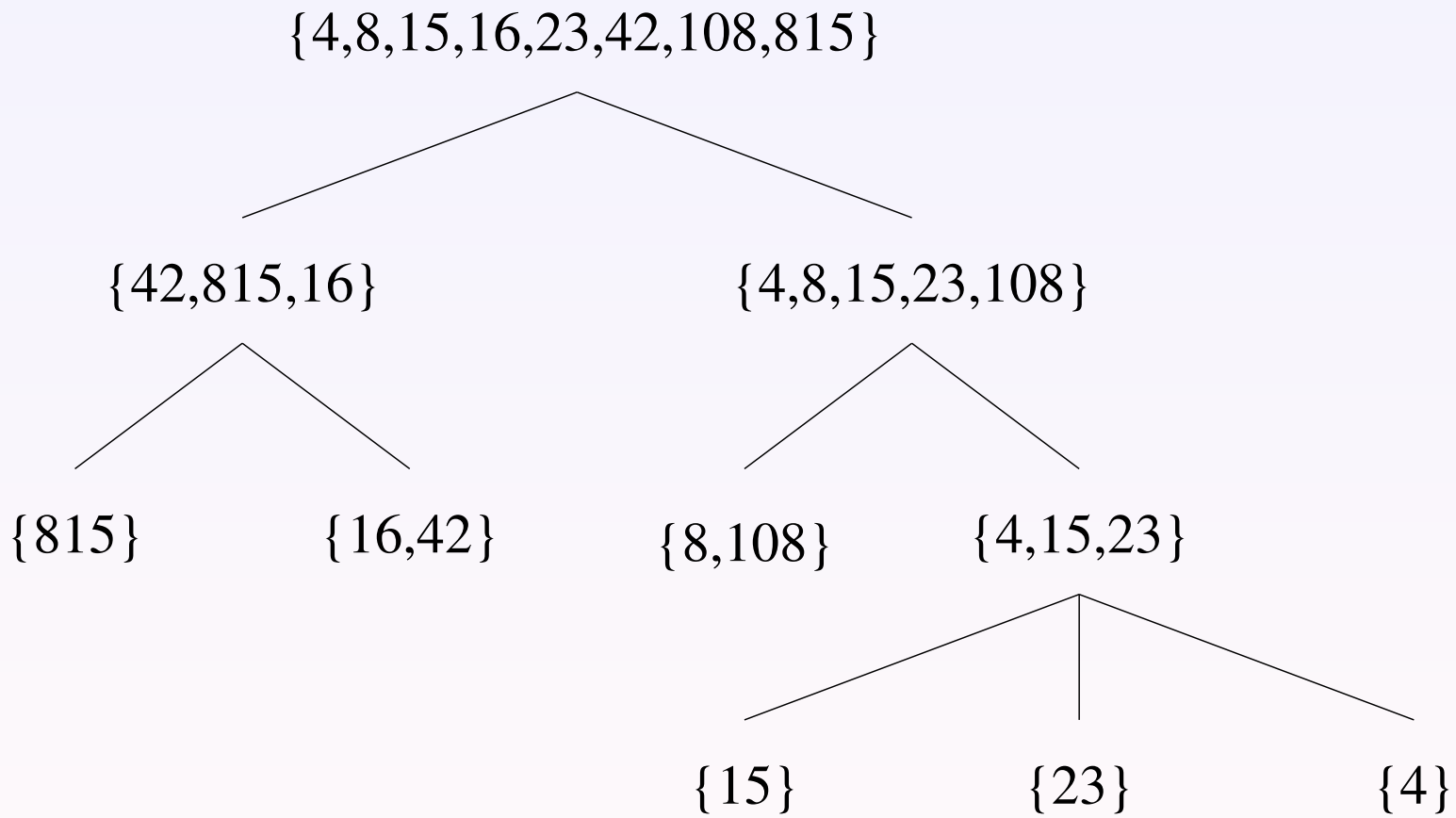
$$\hat{t} = \bigcup_{t' \in \operatorname{sons}(t)} \hat{t}',$$

- if $t \in \mathcal{T}_\mathcal{I}$ has sons, the labels of its sons are disjoint

$$t_1 \neq t_2 \Rightarrow \hat{t}_1 \cap \hat{t}_2 = \emptyset \qquad \text{for all } t_1, t_2 \in \operatorname{sons}(t).$$

Simple example

An acceptable cluster tree for the general index set $\mathcal{I} = \{4, 8, 15, 16, 23, 42, 108, 815\}$ could look as follows:



Properties

Levelwise disjoint: We have

$$t \neq s \Rightarrow \hat{t} \cap \hat{s} = \emptyset \qquad \text{for all } t, s \in \mathcal{T}_{\mathcal{I}} \text{ with } \operatorname{level}(t) = \operatorname{level}(s).$$

Inclusion: By a simple induction this implies

$$\hat{t} \cap \hat{s} \neq \emptyset \Rightarrow s \in \operatorname{sons}^*(t) \text{ or } t \in \operatorname{sons}^*(s) \qquad \text{for all } t, s \in \mathcal{T}_{\mathcal{I}}.$$

Partition: For all $i \in \mathcal{I}$, there is a $t \in \mathcal{T}_{\mathcal{I}}$ with $i \in \hat{t}$.

Leaf partition: Labels of the leaf clusters $\mathcal{L}_{\mathcal{I}}$ form disjoint partition of $\mathcal{I}$

$$\mathcal{I} = \dot{\bigcup_{t \in \mathcal{L}_{\mathcal{I}}}} \hat{t}.$$

How to construct?

In theory there are more than $n!$ possible cluster trees for an index set $\mathcal{I}$ with n elements.

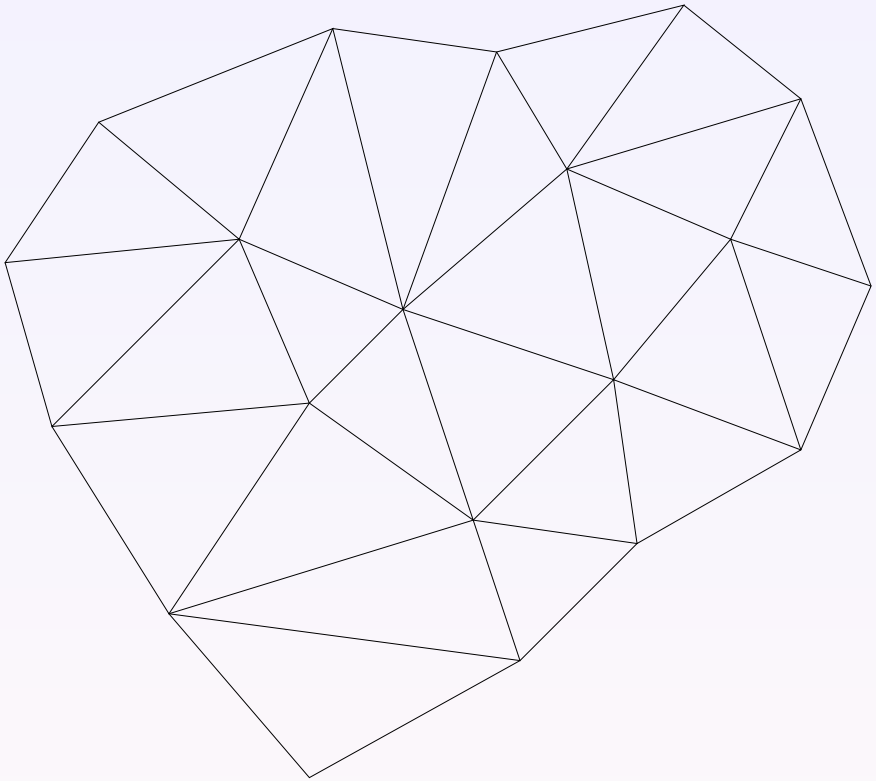
Goal: Construct cluster tree that is well-suited for low-rank approximations.

Approximation quality: Small diameters lead to good approximations.

Complexity: Small number of clusters and small number of levels lead to good efficiency.

General construction

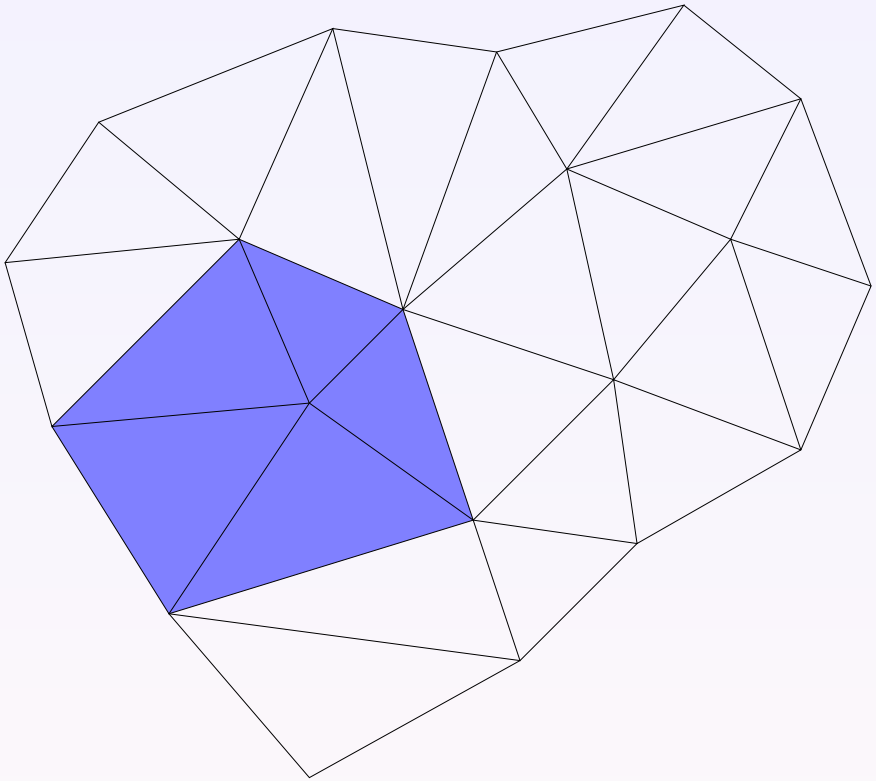
Goal: Construct a good cluster tree for multi-dimensional meshes.



Consider an unstructured triangular mesh.

General construction

Goal: Construct a good cluster tree for multi-dimensional meshes.

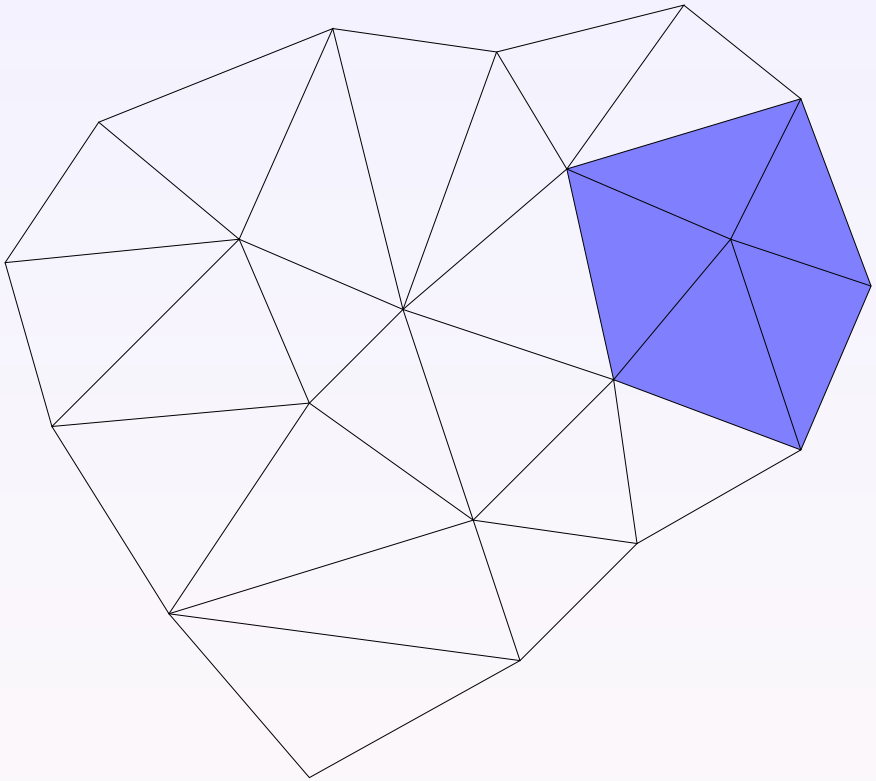


Nodal basis functions may have irregular polygonal supports.

Even overlapping supports are not uncommon.

General construction

Goal: Construct a good cluster tree for multi-dimensional meshes.

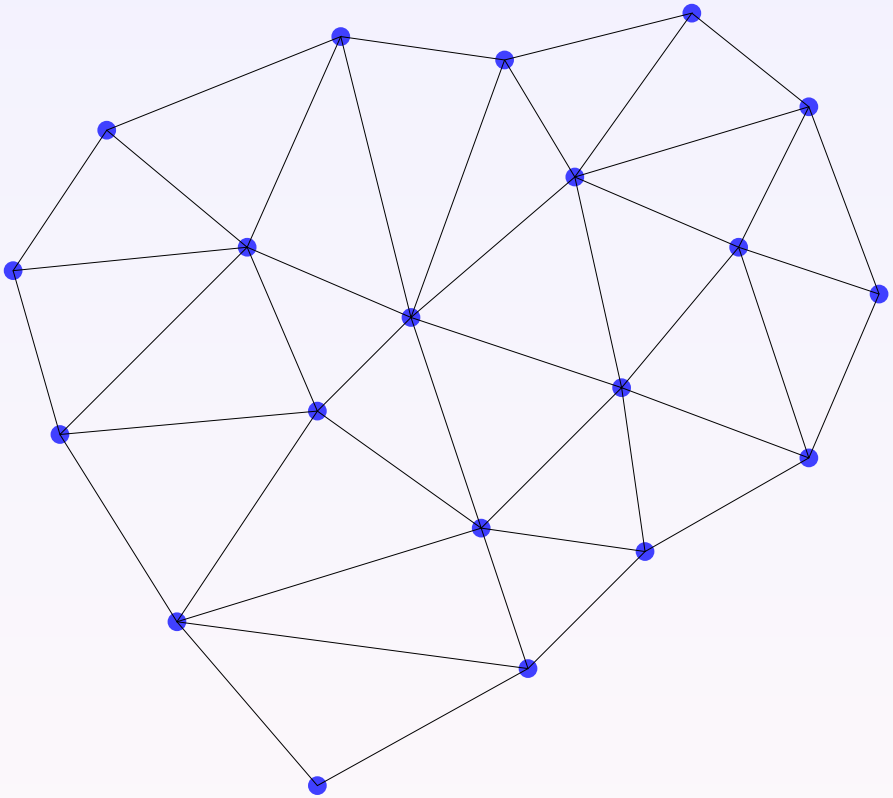


Nodal basis functions may have irregular polygonal supports.

Even overlapping supports are not uncommon.

General construction

Goal: Construct a good cluster tree for multi-dimensional meshes.

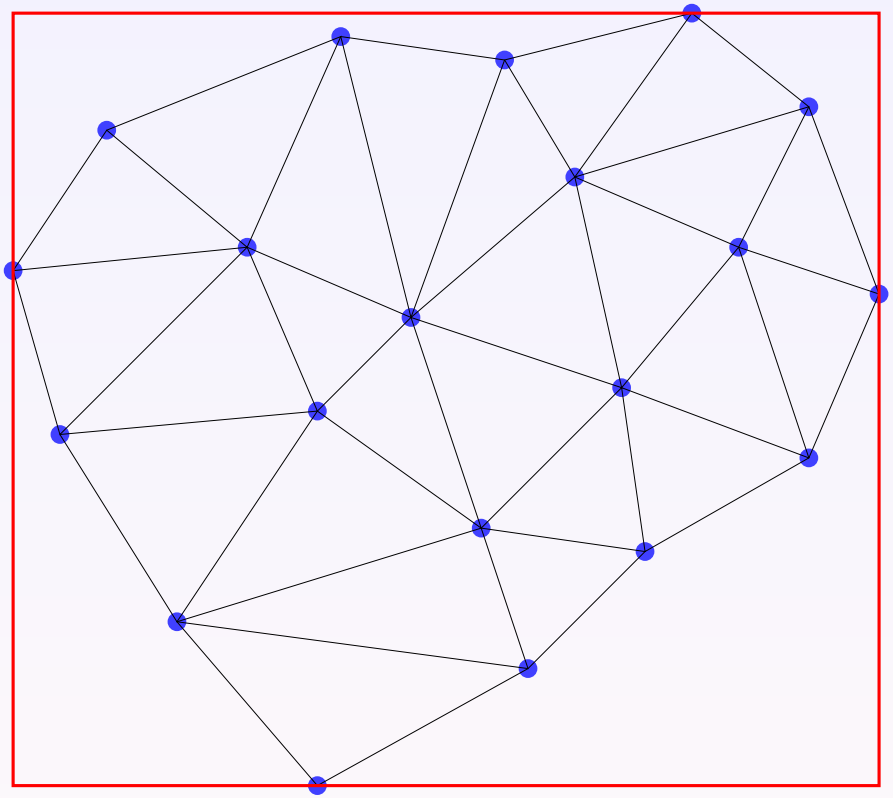


Idea: Replace each support by just one characteristic point.

No overlap, very simple geometric objects.

General construction

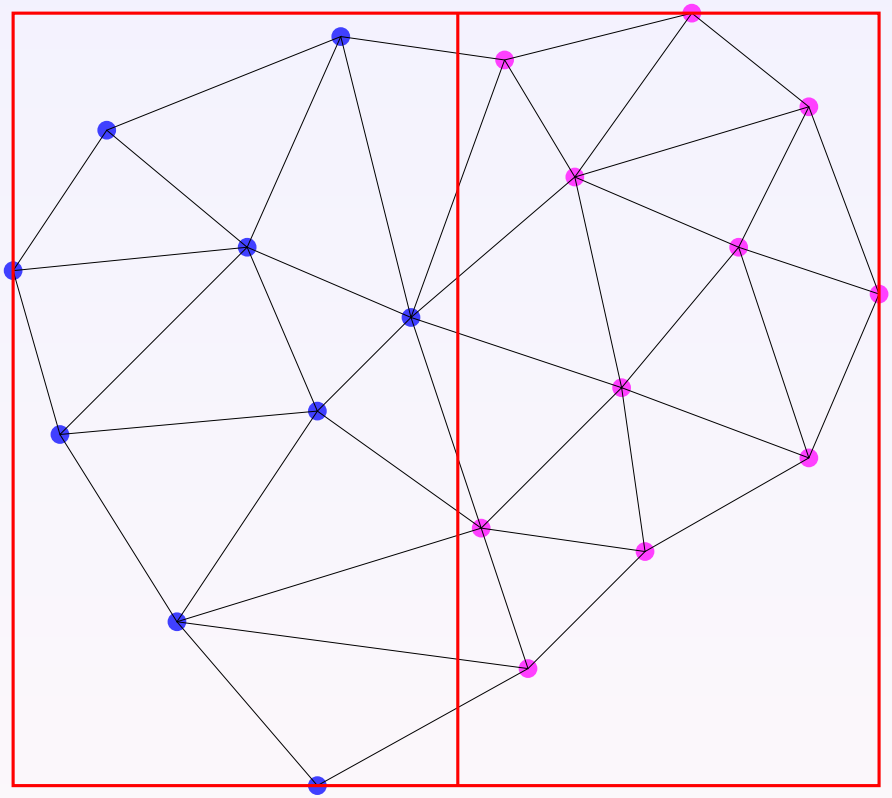
Goal: Construct a good cluster tree for multi-dimensional meshes.



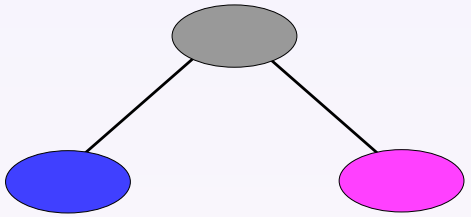
Find an axis-parallel box containing all points.

General construction

Goal: Construct a good cluster tree for multi-dimensional meshes.

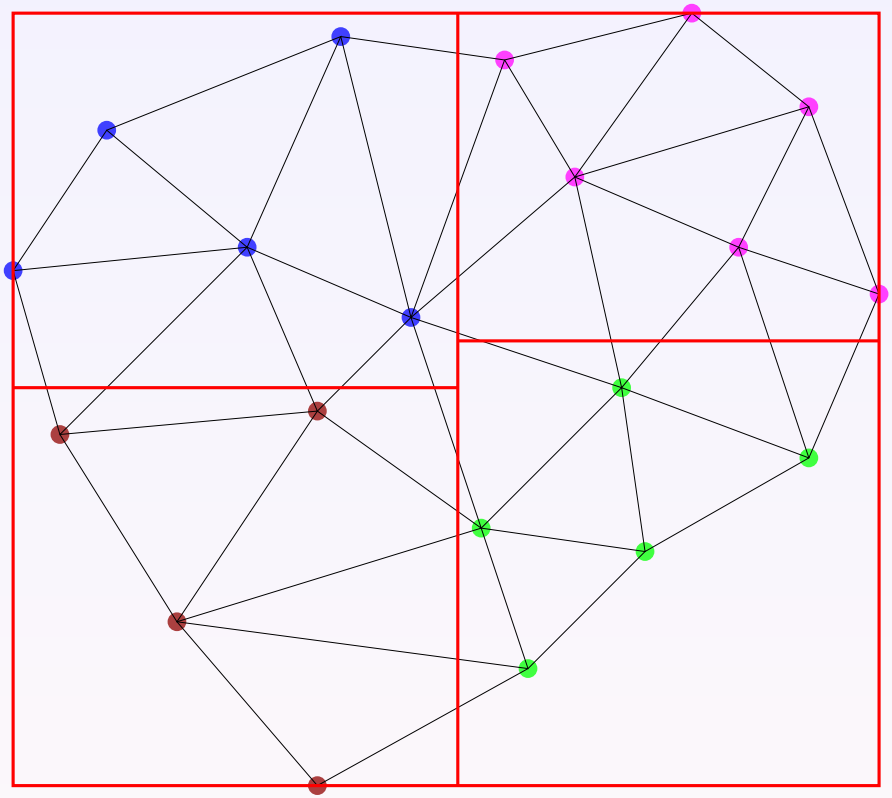


Splitting the box suggests a splitting of the point set.

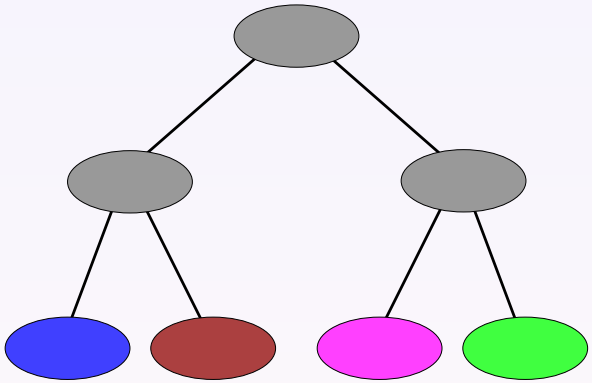


General construction

Goal: Construct a good cluster tree for multi-dimensional meshes.

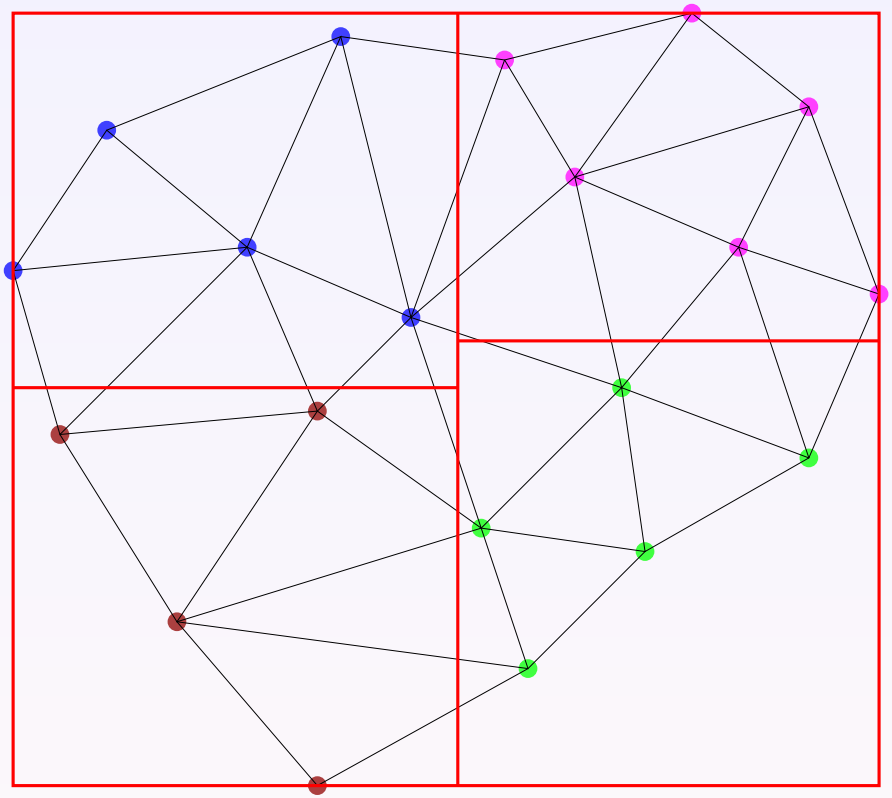


Repeatedly splitting yields a cluster tree.

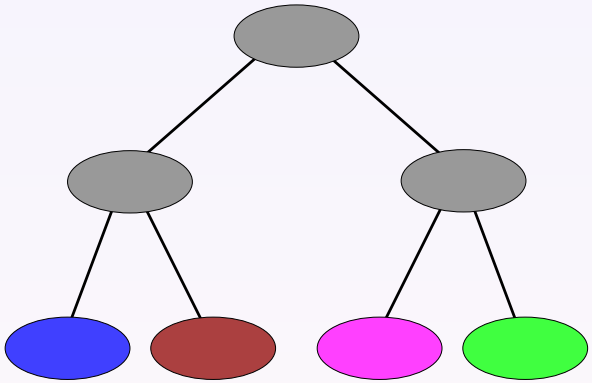


General construction

Goal: Construct a good cluster tree for multi-dimensional meshes.



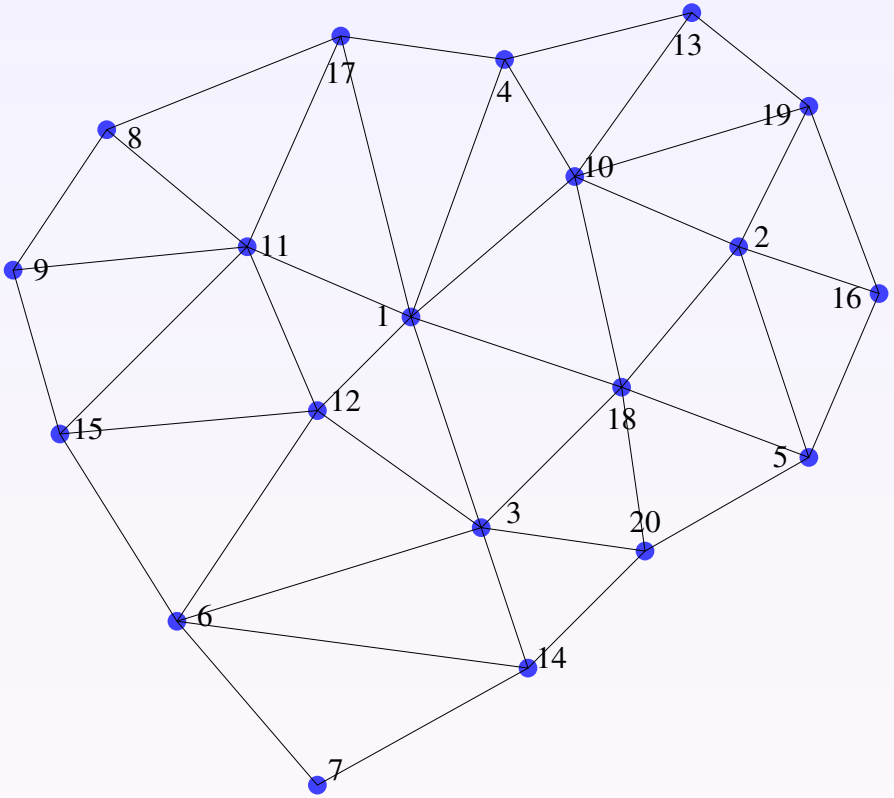
Repeatedly splitting yields a cluster tree.



Result: Simple general algorithm for constructing cluster trees.

Splitting step

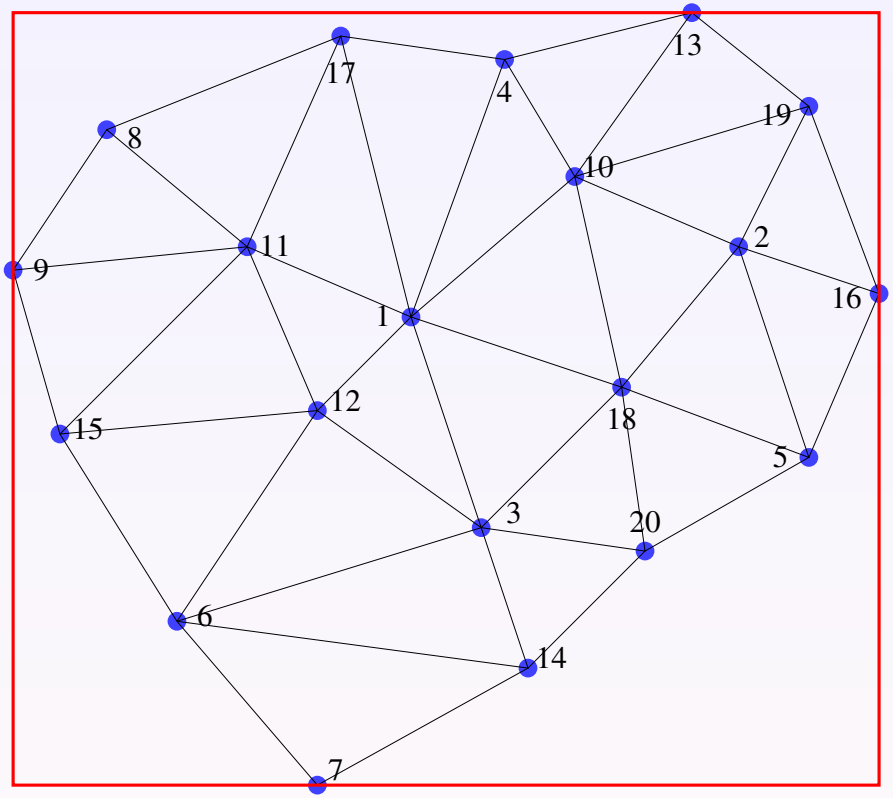
Basic operation: Split one cluster into two sons with consecutively numbered indices.



Consider characteristic points.

Splitting step

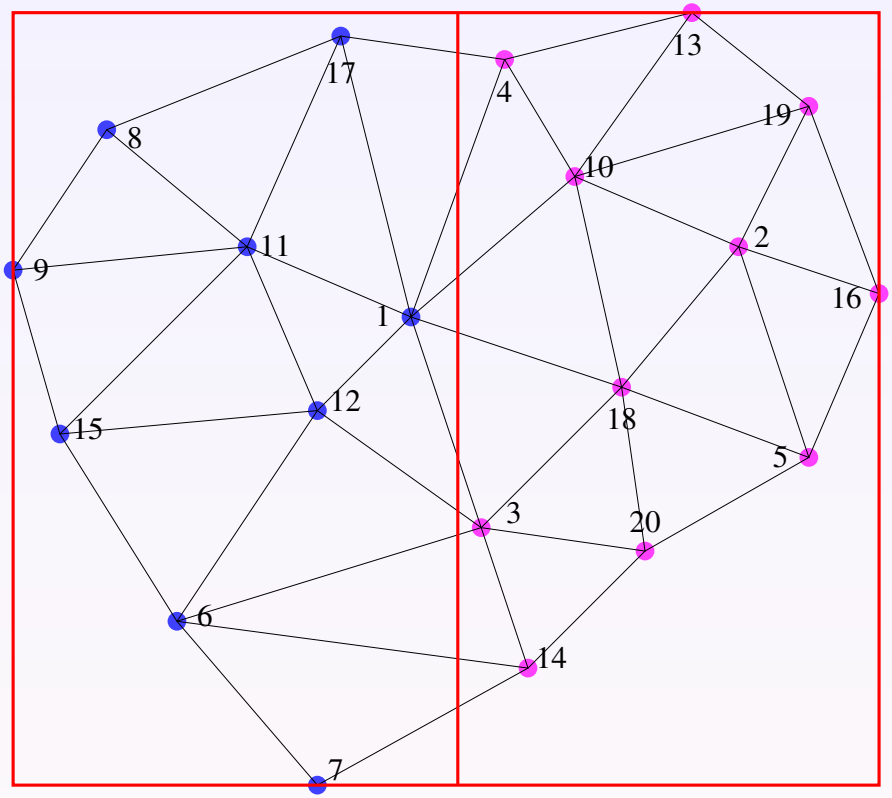
Basic operation: Split one cluster into two sons with consecutively numbered indices.



Find a bounding box.

Splitting step

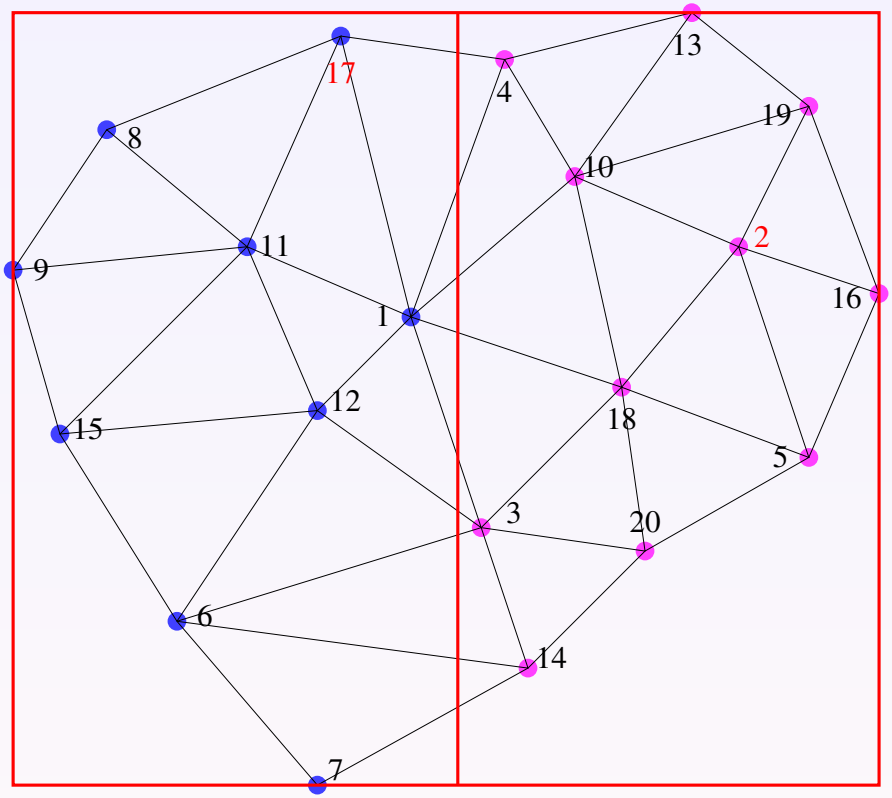
Basic operation: Split one cluster into two sons with consecutively numbered indices.



Split the bounding box.

Splitting step

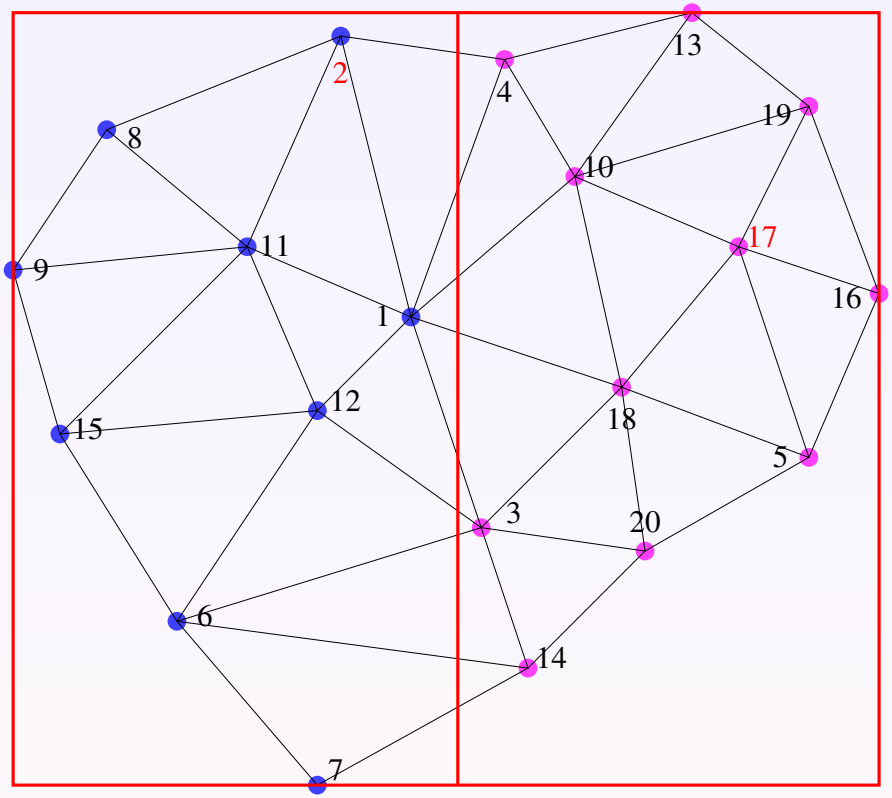
Basic operation: Split one cluster into two sons with consecutively numbered indices.



No consecutive ordering.

Splitting step

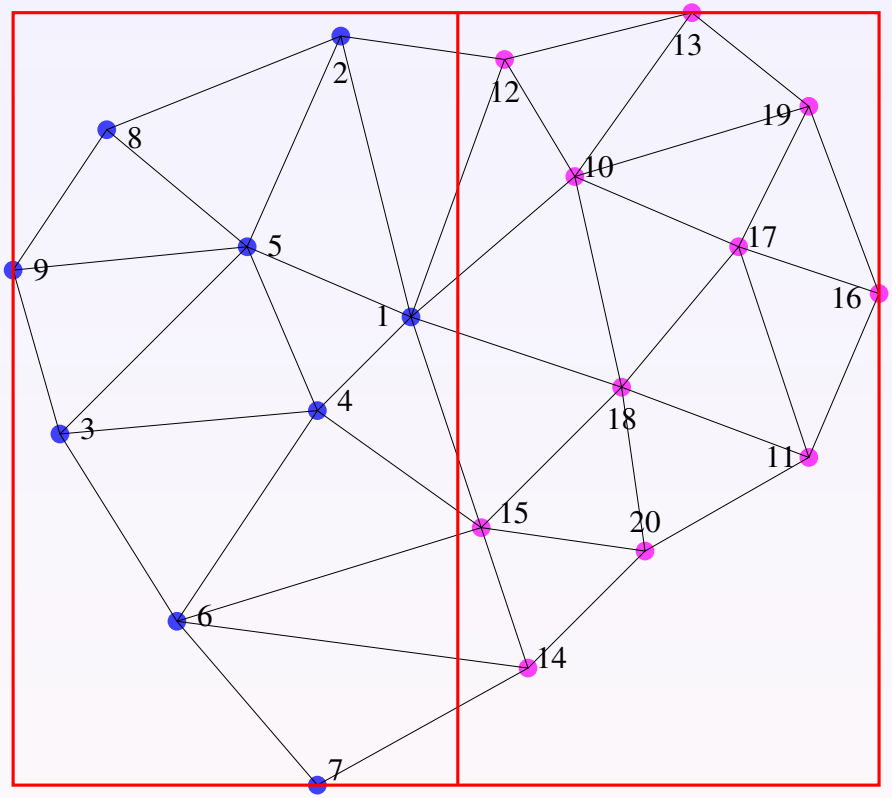
Basic operation: Split one cluster into two sons with consecutively numbered indices.



Fix ordering by exchanging numbers.

Splitting step

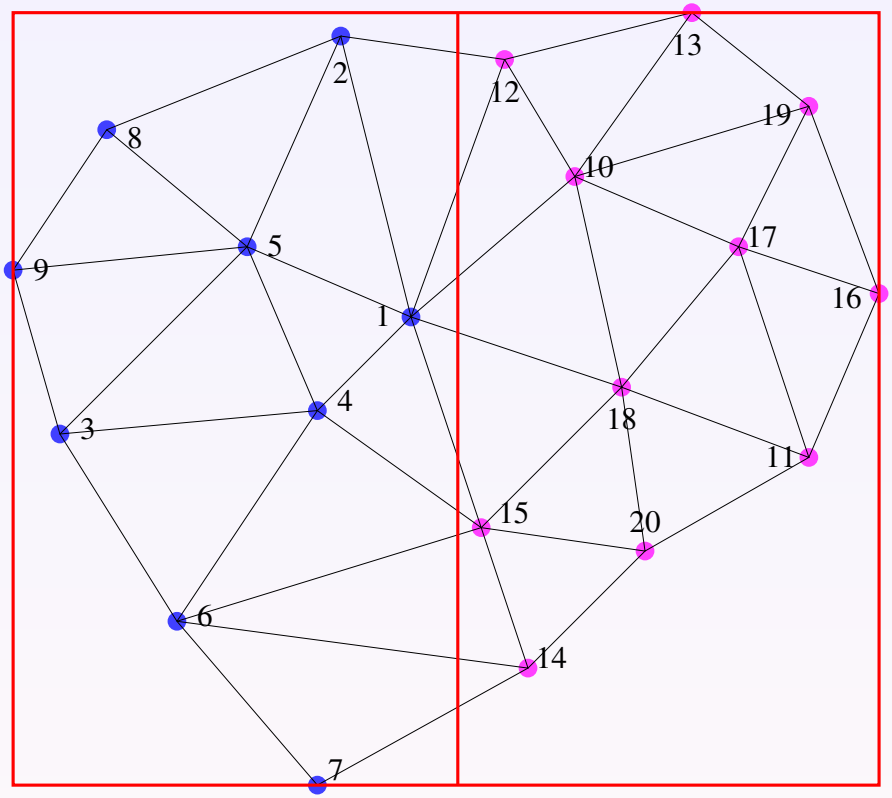
Basic operation: Split one cluster into two sons with consecutively numbered indices.



Keep exchanging until consecutive.

Splitting step

Basic operation: Split one cluster into two sons with consecutively numbered indices.



Keep exchanging until consecutive.

Result: Simple recursive algorithm for finding both a suitable cluster tree and a matching permutation of the indices.

Alternatives

Regular clustering:

- Cycle through coordinate directions instead of choosing direction of maximal extend.
- Do not shrink box adaptively.
- Guarantees geometrically similar boxes every d steps.

Cardinality-balanced clustering:

- Split into two sons of approximately equal cardinality.
- Guarantees balanced cluster tree.

Domain decomposition clustering:

- Split into two sons and their interface.
- Useful for handling sparse matrices, e.g., FEM.

General block cluster tree

Goal: Block cluster tree for general cluster trees $\mathcal{T}_\mathcal{I}$ and $\mathcal{T}_\mathcal{J}$.

Definition: A labeled tree $\mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ is a block cluster tree for $\mathcal{T}_\mathcal{I}$ and $\mathcal{T}_\mathcal{J}$ if

- each node $b \in \mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ is a pair $b = (t, s)$ of clusters $t \in \mathcal{T}_\mathcal{I}$, $s \in \mathcal{T}_\mathcal{J}$,
- the root of $\mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ is the pair of the roots of $\mathcal{T}_\mathcal{I}$ and $\mathcal{T}_\mathcal{J}$,
- the label of $b = (t, s) \in \mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ is the Cartesian product of the labels of t and s, i.e., $\hat{b} = \hat{t} \times \hat{s}$,
- if $b = (t, s) \in \mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ has sons, we have

$$\operatorname{sons}(b) = \begin{cases} \{(t', s) \;:\; t' \in \operatorname{sons}(t)\} & \text{if } \operatorname{sons}(s) = \emptyset, \\ \{(t, s') \;:\; s' \in \operatorname{sons}(s)\} & \text{if } \operatorname{sons}(t) = \emptyset, \\ \{(t', s') \;:\; t' \in \operatorname{sons}(t), s' \in \operatorname{sons}(s)\} & \text{otherwise.} \end{cases}$$

Observation: A block cluster tree $\mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ for $\mathcal{T}_\mathcal{I}$ and $\mathcal{T}_\mathcal{J}$ is a special cluster tree for the index set $\mathcal{I} \times \mathcal{J}$.

How to construct?

Observation: Since $\mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ is a special cluster tree for $\mathcal{I}\times\mathcal{J}$, the labels of its leaves $\mathcal{L}_{\mathcal{I}\times\mathcal{J}}$ correspond to a disjoint partition of matrix entries

$$\{\hat{t}\times\hat{s} \;:\; b=(t,s)\in\mathcal{L}_{\mathcal{I}\times\mathcal{J}}\}.$$

This partition is used to define the hierarchical matrix.

Goals: Since we have to approximate each submatrix, we require

- that the number of leaves is small and
- that inadmissible leaves correspond to small index sets.

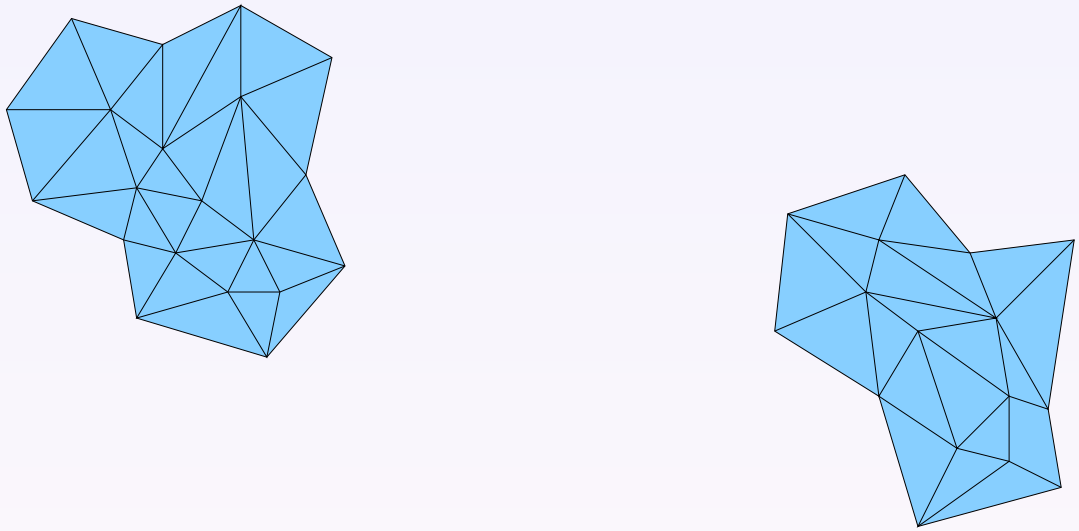
Algorithm: If $(t,s)\in\mathcal{T}_{\mathcal{I}\times\mathcal{J}}$ is admissible, make it a leaf.
Otherwise construct its sons and proceed by recursion.

Admissibility condition

Goal: Construct a good block partition for the admissibility condition

$$\operatorname{diam}(\tau) \leq 2 \operatorname{dist}(\tau, \sigma).$$

Problem: How to check the condition efficiently?

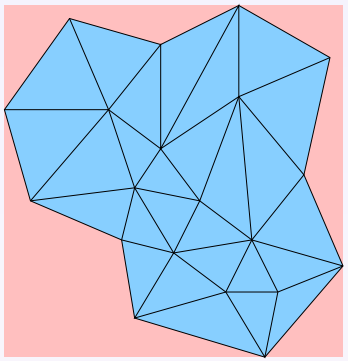


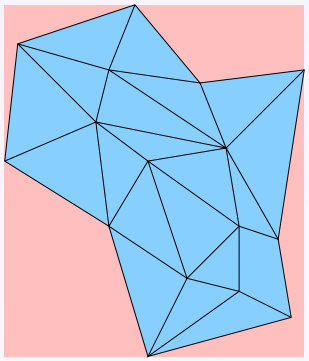
Admissibility condition

Goal: Construct a good block partition for the admissibility condition

$$\operatorname{diam}(\tau) \leq 2 \operatorname{dist}(\tau, \sigma).$$

Problem: How to check the condition efficiently?





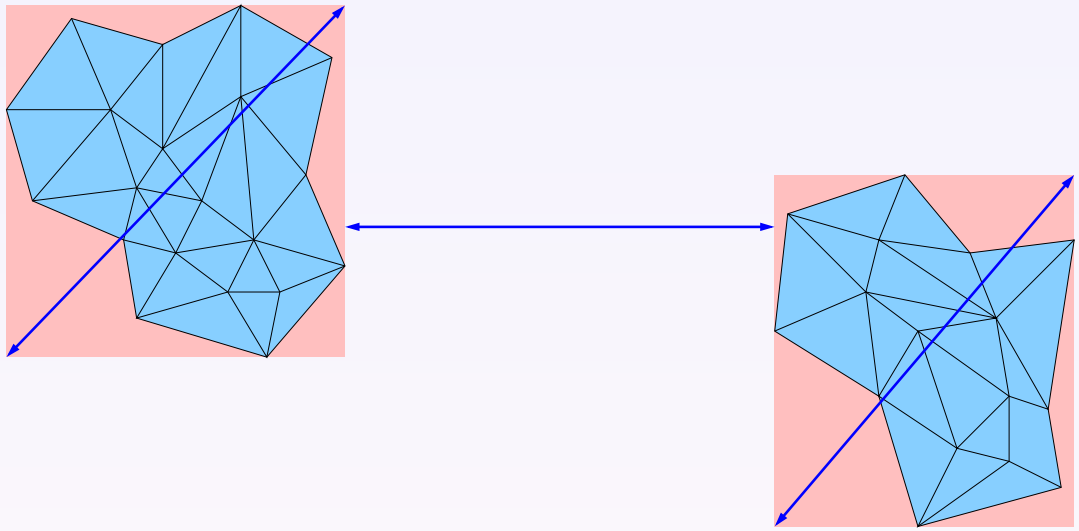
Idea: Replace subdomains τ, σ by larger domains B_t, B_s that can be handled efficiently, e.g., axis-parallel bounding boxes.

Admissibility condition

Goal: Construct a good block partition for the admissibility condition

$$\operatorname{diam}(\tau) \leq 2 \operatorname{dist}(\tau, \sigma).$$

Problem: How to check the condition efficiently?



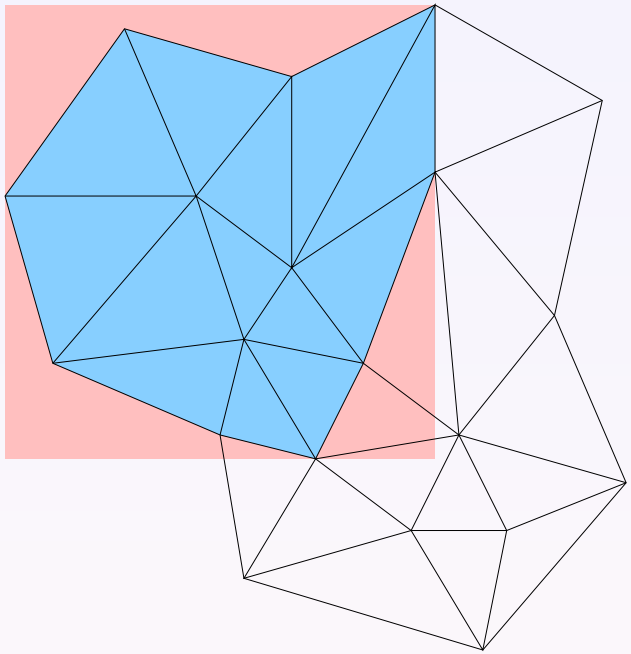
Idea: Replace subdomains τ, σ by larger domains B_t, B_s that can be handled efficiently, e.g., axis-parallel bounding boxes.

$$\operatorname{diam}(B_t) \leq 2 \operatorname{dist}(B_t, B_s) \qquad \Longrightarrow \qquad \operatorname{diam}(\tau) \leq 2 \operatorname{dist}(\tau, \sigma)$$

Construction of bounding boxes

Goal: Find minimal box B_t with $\tau \subseteq B_t$.

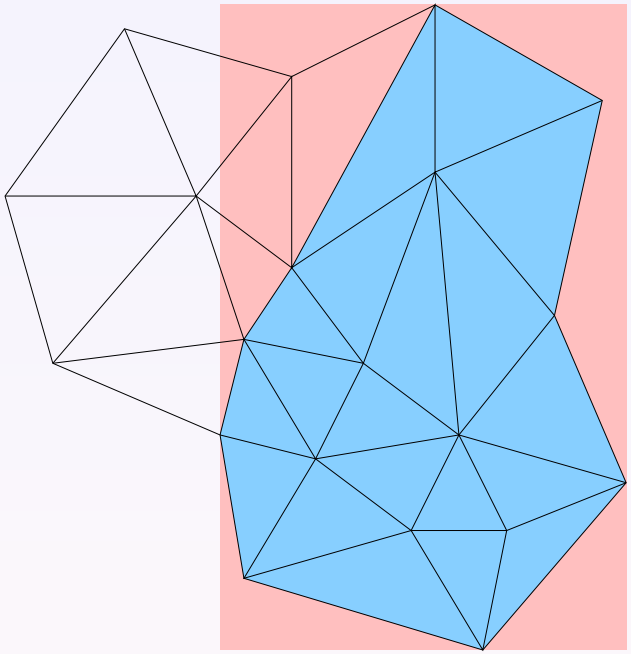
Recursion: Construct boxes for sons $t' \in \operatorname{sons}(t)$ of t and merge them.



Construction of bounding boxes

Goal: Find minimal box B_t with $\tau \subseteq B_t$.

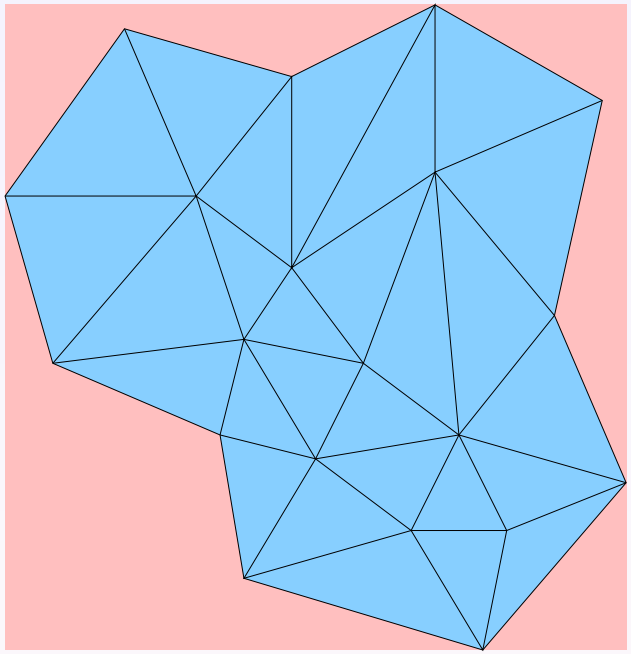
Recursion: Construct boxes for sons $t' \in \operatorname{sons}(t)$ of t and merge them.



Construction of bounding boxes

Goal: Find minimal box B_t with $\tau \subseteq B_t$.

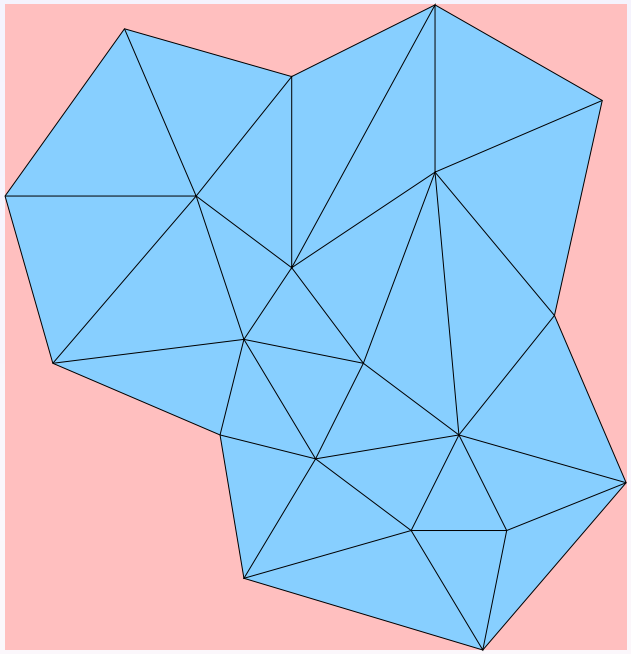
Recursion: Construct boxes for sons $t' \in \operatorname{sons}(t)$ of t and merge them.



Construction of bounding boxes

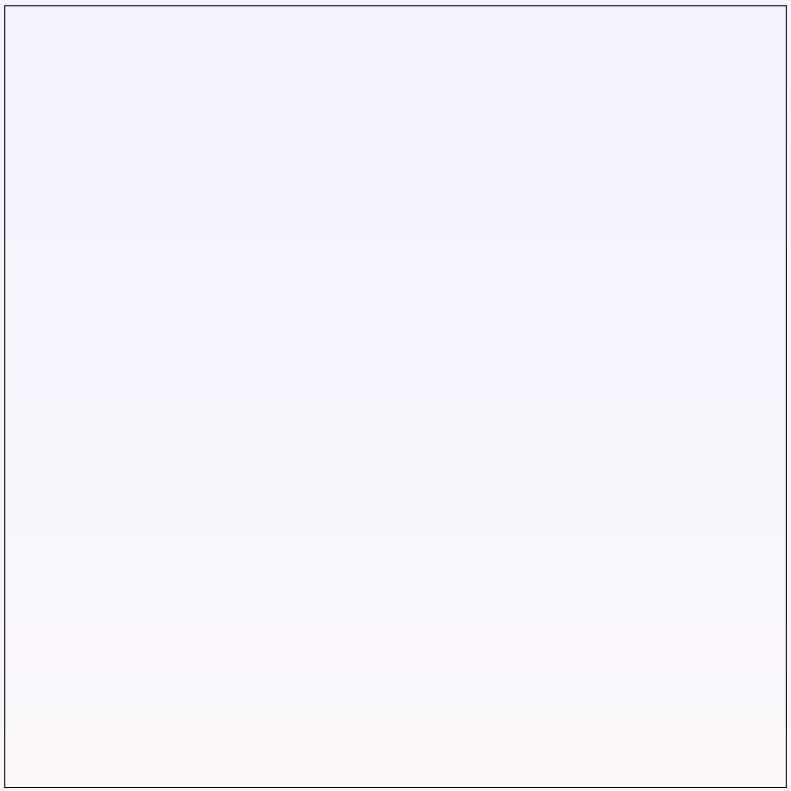
Goal: Find minimal box B_t with $\tau \subseteq B_t$.

Recursion: Construct boxes for sons $t' \in \operatorname{sons}(t)$ of t and merge them.



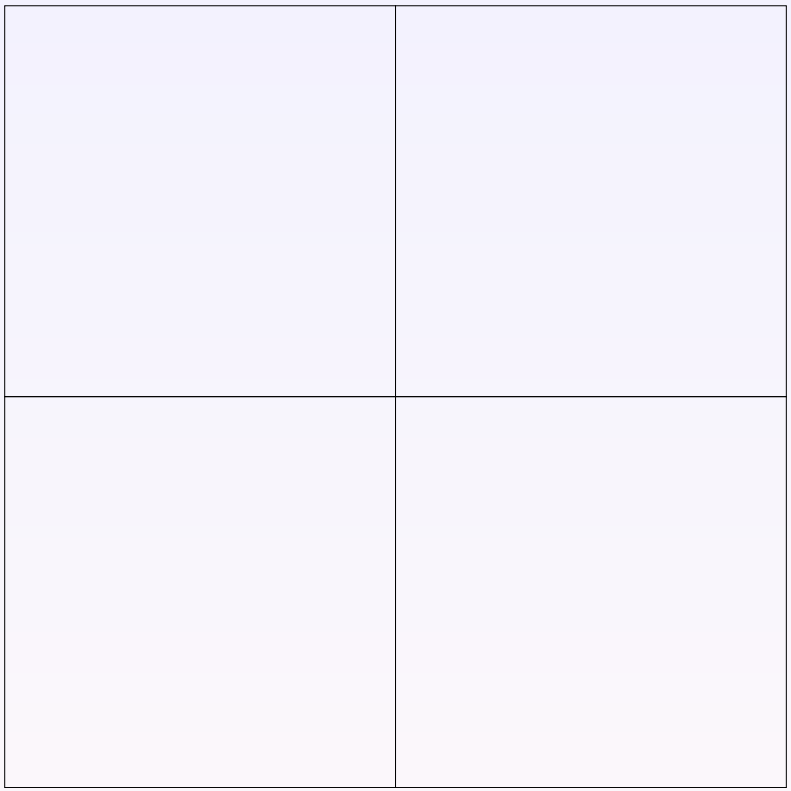
Result: Only $\mathcal{O}(n)$ operations for all bounding boxes.

Matrix structure



Start with the entire matrix.

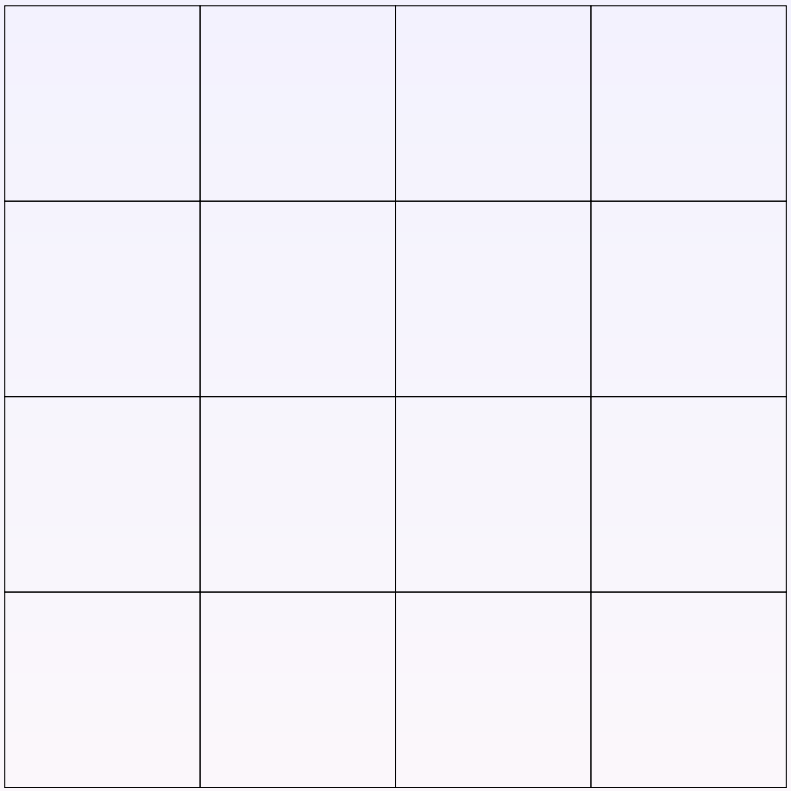
Matrix structure



Start with the entire matrix.

Subdivide according to the cluster tree.

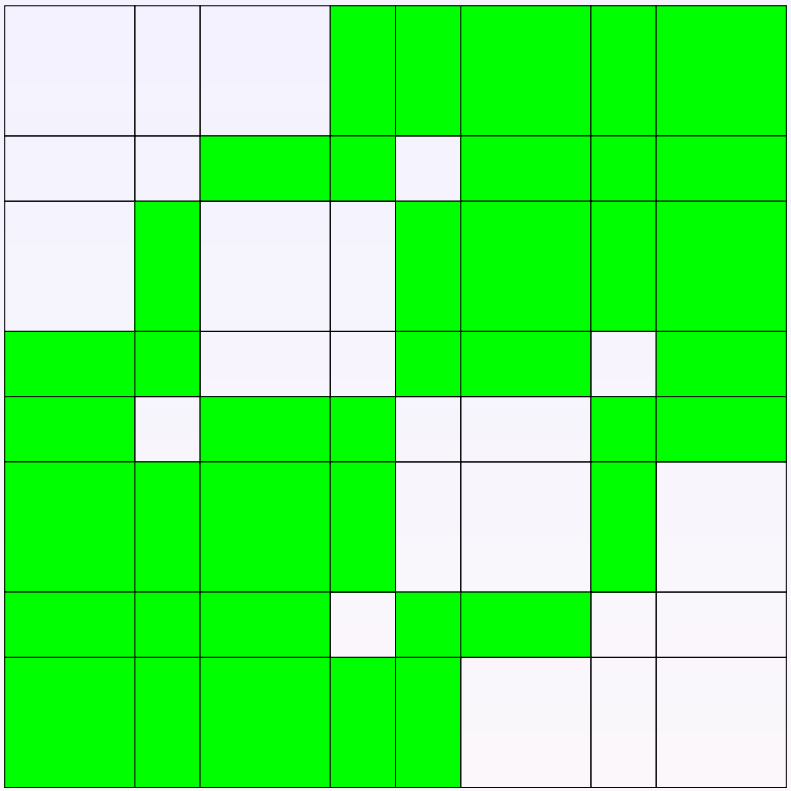
Matrix structure



Start with the entire matrix.

Subdivide according to the cluster tree.

Matrix structure

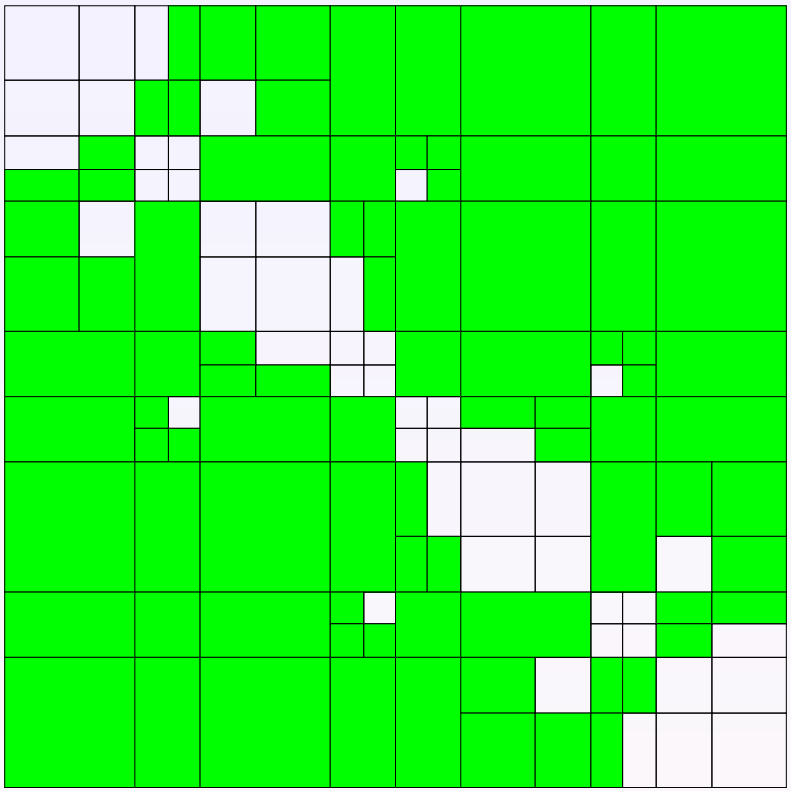


Start with the entire matrix.

Subdivide according to the cluster tree.

Mark admissible blocks and subdivide the others.

Matrix structure

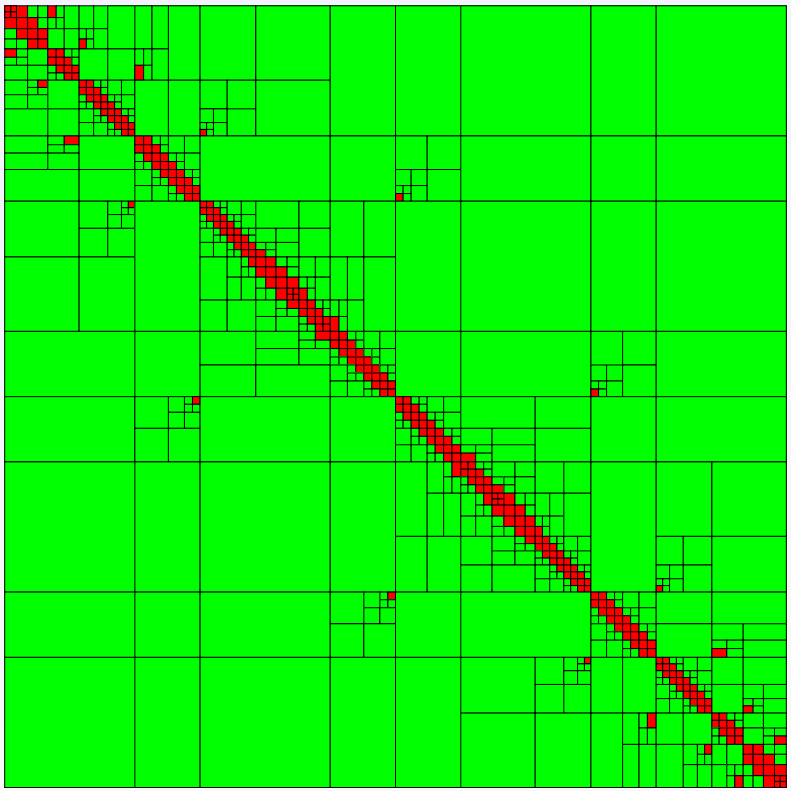


Start with the entire matrix.

Subdivide according to the cluster tree.

Mark admissible blocks and subdivide the others.

Matrix structure



Start with the entire matrix.

Subdivide according to the cluster tree.

Mark admissible blocks and subdivide the others.

Result: $\mathcal{H}$-matrix structure.

Data structures for a cluster tree

Cluster: Consecutive sequence of indices, organized in a tree.

```
struct _cluster {
  int start;
  int size;
  double *bmin, *bmax;
  int sons;
  pcluster *son;
};
```

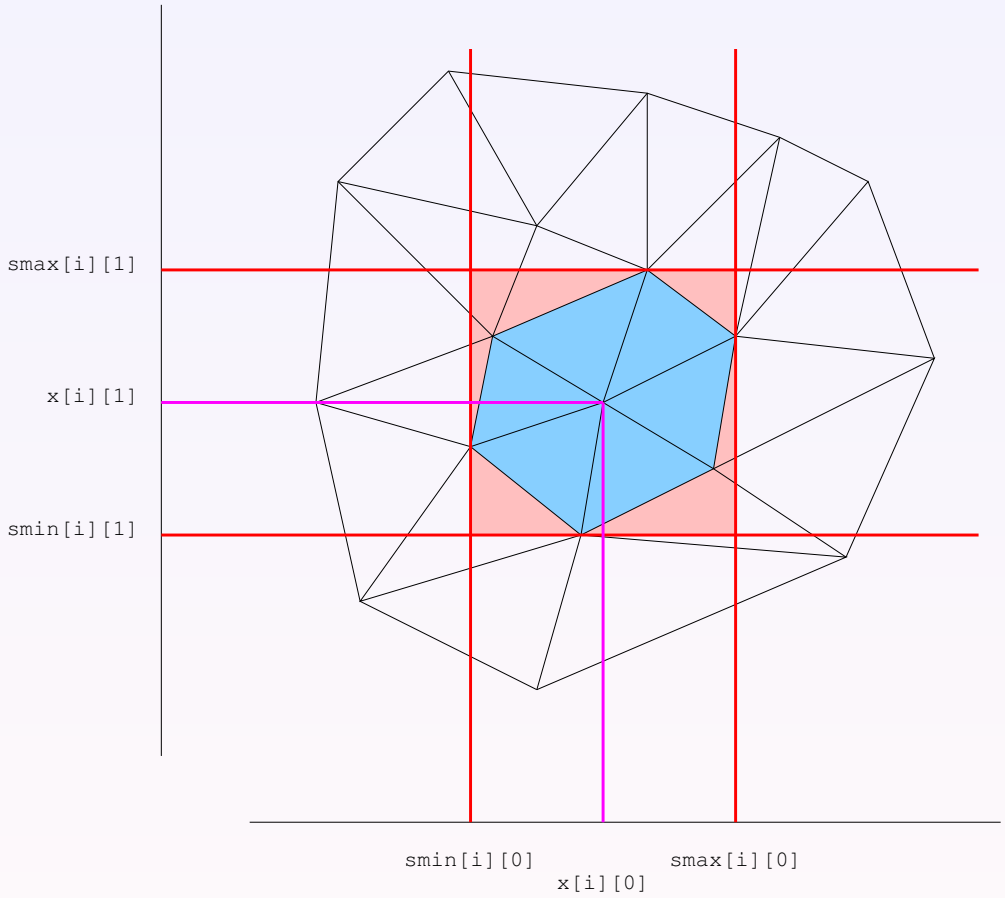
Cluster tree: Index permutation and pointer to root cluster.

```
struct _clustertree {
  int ndof;
  int nidx;
  int *dof2idx;
  int *idx2dof;
  pcluster root;
};
```

Data structure representing the geometry

Cluster factory: Contains coordinates of characteristic points and bounding boxes for supports of single basis functions.

```
struct _clusterfactory {
  double **x;
  double **smin;
  double **smax;
  int n;
  int d;
};
```



Splitting in HLib

```
static pcluster
split_geometrically(pclusterfactory factory, int *index,
                    int start, int sz, int leafsize)
{
  /* ... some initialization ... */
  if(sz <= leafsize)                        /* Stop if small enough */
    this = new_cluster(start, sz, 0);
  else {
    for(j=0; j<d; j++) {                    /* Determine bounding box */
      vmin[j] = vmax[j] = x[index[0]][j];
      for(i=1; i<sz; i++)
        if(x[index[i]][j] < vmin[j])       vmin[j] = x[index[i]][j];
        else if(vmax[j] < x[index[i]][j]) vmax[j] = x[index[i]][j];
    }
    jmax = 0; vdiff = vmax[0] - vmin[0]; /* Find maximal extent */
    for(j=1; j<d; j++)
      if(vmax[j] - vmin[j] > vdiff) { jmax = j; vdiff = vmax[j] - vmin[j]; }
    l = 0; r = sz-1;                        /* Rearrange array */
    vmid = 0.5 * (vmax[jmax] + vmin[jmax]);
    while(l < r) {
      while(l < sz && x[index[l]][jmax] <= vmid) l++;
      while(r >= 0 && x[index[r]][jmax] >  vmid) r--;
      if(l < r) { h = index[l]; index[l] = index[r]; index[r] = h; }
    }
    this = new_cluster(start, sz, 2);    /* Recursion */
    this->son[0] = split_geometrically(factory, index,   start,   l,    leafsize);
    this->son[1] = split_geometrically(factory, index+l, start+l, sz-l, leafsize);
  }

  return this;
}
```

Bounding boxes in HLib

```
static void
find_boundingbox(pcluster tau, pcclusterfactory factory)
{
  /* ... some initialization ... */

  if(sons > 0) {
    for(i=0; i<sons; i++)
     find_boundingbox(tau->son[i], factory);

    for(j=0; j<d; j++) {
      bmin[j] = son[0]->bmin; bmax[j] = son[0]->bmax;
    }
    for(i=1; i<sons; i++)
      for(j=0; j<d; j++) {
        bmin[j] = dmin(bmin[j], son[i]->bmin[j]);
        bmax[j] = dmax(bmax[j], son[i]->bmax[j]);
      }
  }
  else {
    for(j=0; j<d; j++) {
      bmin[j] = smin[index[0]]; bmax[j] = smax[index[0]];
    }
    for(i=1; i<size; i++)
      for(j=0; j<d; j++) {
        bmin[j] = dmin(bmin[j], smin[index[i]][j]);
        bmax[j] = dmax(bmax[j], smax[index[i]][j]);
      }
  }
}
```

Data structure for a block cluster tree

Block cluster: Contains row and column cluster, admissibility flags and pointers to sons.

```
typedef struct {
  unsigned weakadm : 1;
  unsigned minadm : 1;
  unsigned maxadm : 1;
} BlockType;

struct _blockcluster {
  pccluster row;
  pccluster col;
  BlockType type;
  pblockcluster *son;
  int block_rows;
  int block_cols;
};
```

Block tree in HLib

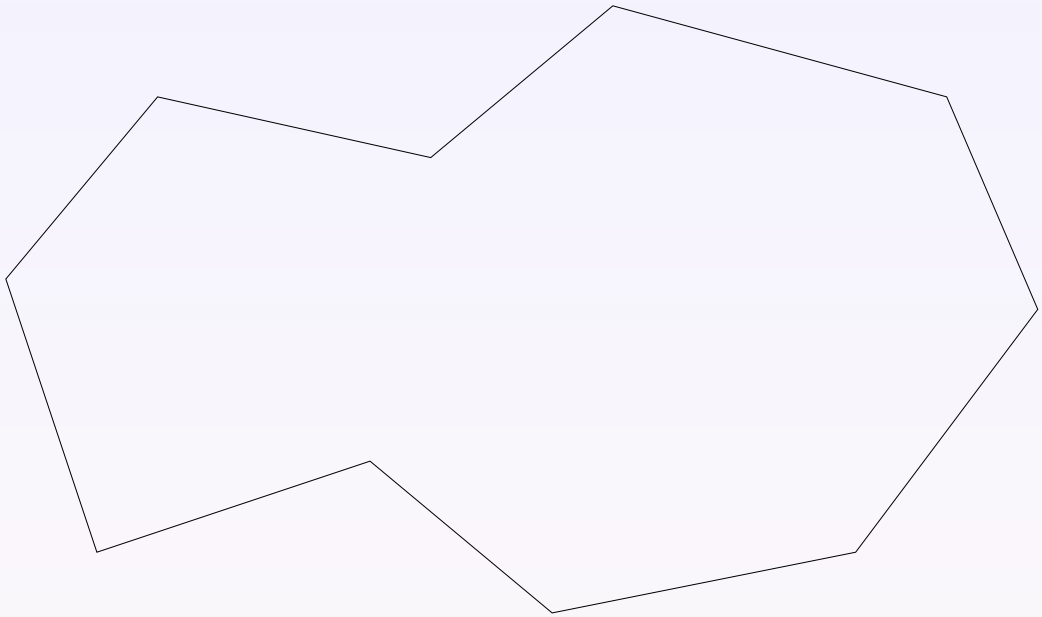
```
pblockcluster
build_blockcluster(pccluster row, pccluster col, double eta)
{
  dist = distance_cluster(row, col);
  diam_row = diameter_cluster(row);
  diam_col = diameter_cluster(col);
  type.maxadm = type.minadm = type.weakadm = 0;
  if(diam_row < eta*dist || diam_col < eta*dist) {
    type.minadm = type.weakadm = 1;
    bc = new_blockcluster(row, col, 0, 0, type);
  }
  else if(row->sons > 0 && col->sons > 0) {
    bc = new_blockcluster(row, col, row->sons, col->sons, type);
    for(j=0; j<col->sons; j++)
      for(i=0; i<row->sons; i++)
        bc->son[i+j*bc->block_rows] =
          build_blockcluster(row->son[i], col->son[j], eta);
  }
  else bc = new_blockcluster(row, col, 0, 0, type);
  return bc;
}
```

$\mathcal{H}$-matrix in HLib

```
psupermatrix
build_supermatrix_from_blockcluster(pcblockcluster bc, int k)
{
  if(bc->son) {
    s = new_supermatrix(block_rows, block_cols, rows, cols,
                        NULL, NULL, NULL);
    for(j=0; j<block_cols; j++) for(i=0; i<block_rows; i++)
      s->s[i+j*block_rows] =
        build_supermatrix_from_blockcluster(bc->son[i+j*block_rows], k, eps);
  }
  else if(bc->type.weakadm) {
    r = new_rkmatrix(k, rows, cols);
    s = new_supermatrix(1, 1, rows, cols, NULL, r, NULL);
  }
  else {
    f = new_fullmatrix(rows, cols);
    s = new_supermatrix(1, 1, rows, cols, NULL, NULL, f);
  }
  return s;
}
```

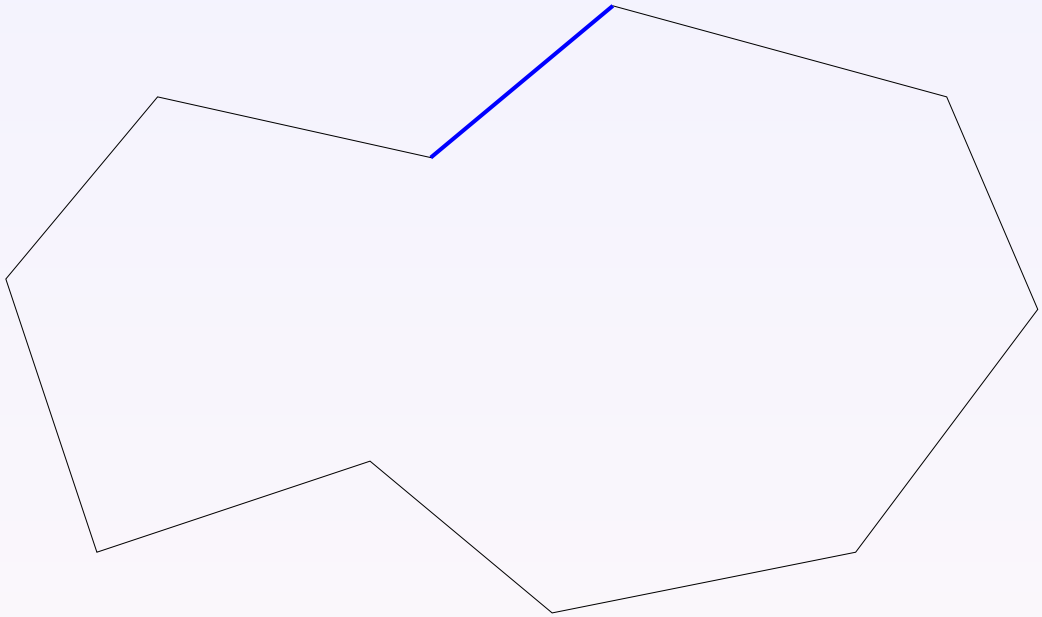
Example: Curve

Example: Polygonal curve in two-dimensional space with piecewise constant basis functions.



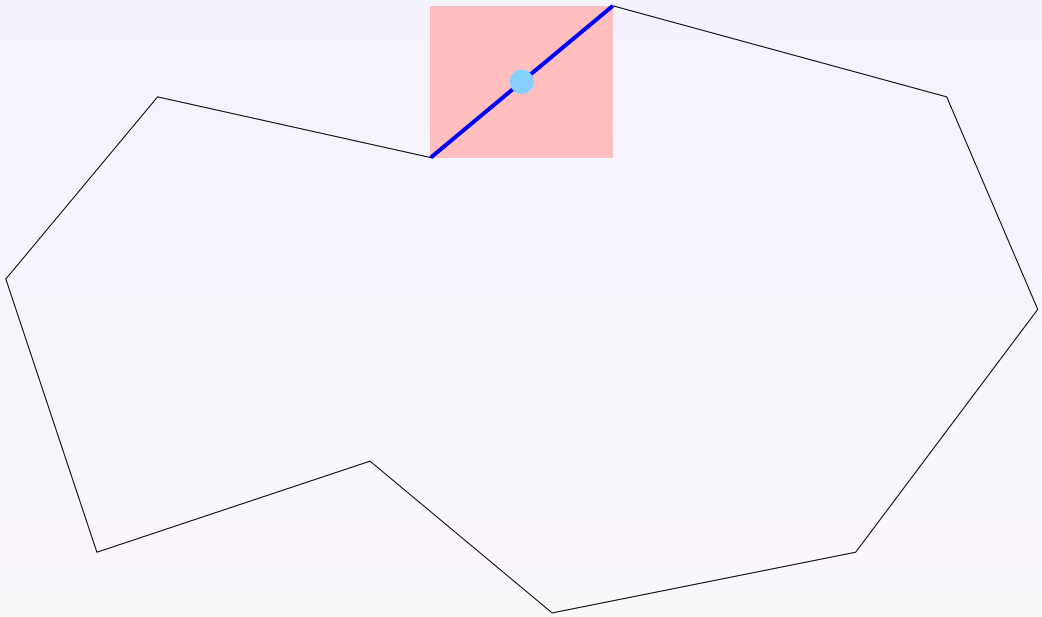
Example: Curve

Example: Polygonal curve in two-dimensional space with piecewise constant basis functions.



Example: Curve

Example: Polygonal curve in two-dimensional space with piecewise constant basis functions.



Observation: Midpoints are good characteristic points.
Optimal bounding boxes are easily found.

Setup of cluster factory

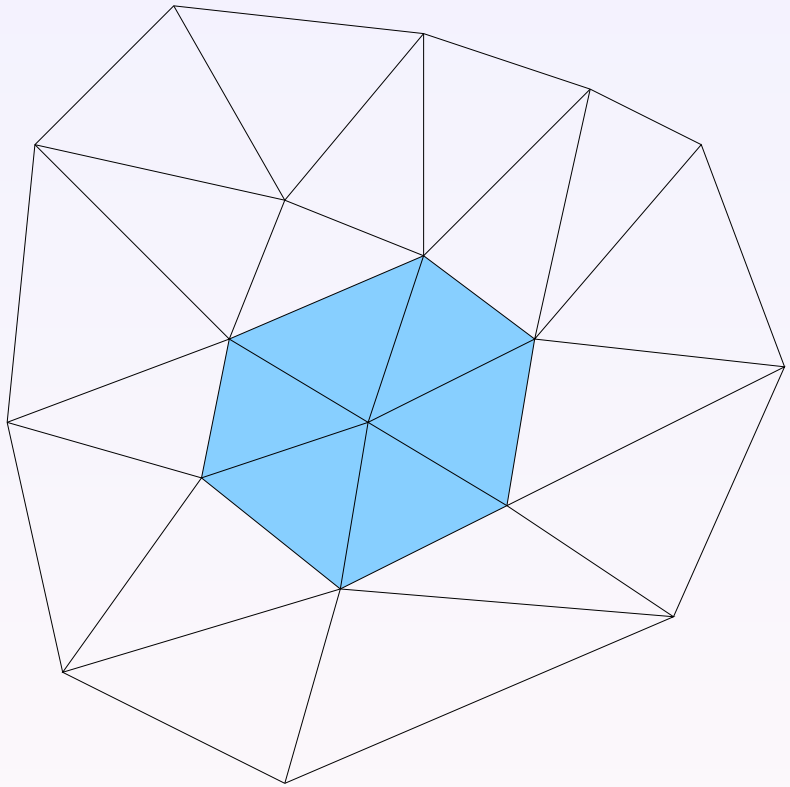
Approach: Represent curve by cyclic array of vertices, compute midpoints and bounding boxes, use general algorithm in HLib.

```
factory = new_clusterfactory(n, 2);

for(i=0; i<n; i++) {
  factory->x[i][0] = 0.5 * (vertex[i][0] + vertex[(i+1)%n][0]);
  factory->x[i][1] = 0.5 * (vertex[i][1] + vertex[(i+1)%n][1]);
  factory->smin[i][0] = dmin(vertex[i][0], vertex[(i+1)%n][0]);
  factory->smax[i][0] = dmax(vertex[i][0], vertex[(i+1)%n][0]);
  factory->smin[i][1] = dmin(vertex[i][1], vertex[(i+1)%n][1]);
  factory->smax[i][1] = dmax(vertex[i][1], vertex[(i+1)%n][1]);
}

cluster = create_clustertree(factory, HLIB_GEOMETRIC, 1, 0);
```

Result: Construction of a cluster tree requires only small amount of problem-specific code.

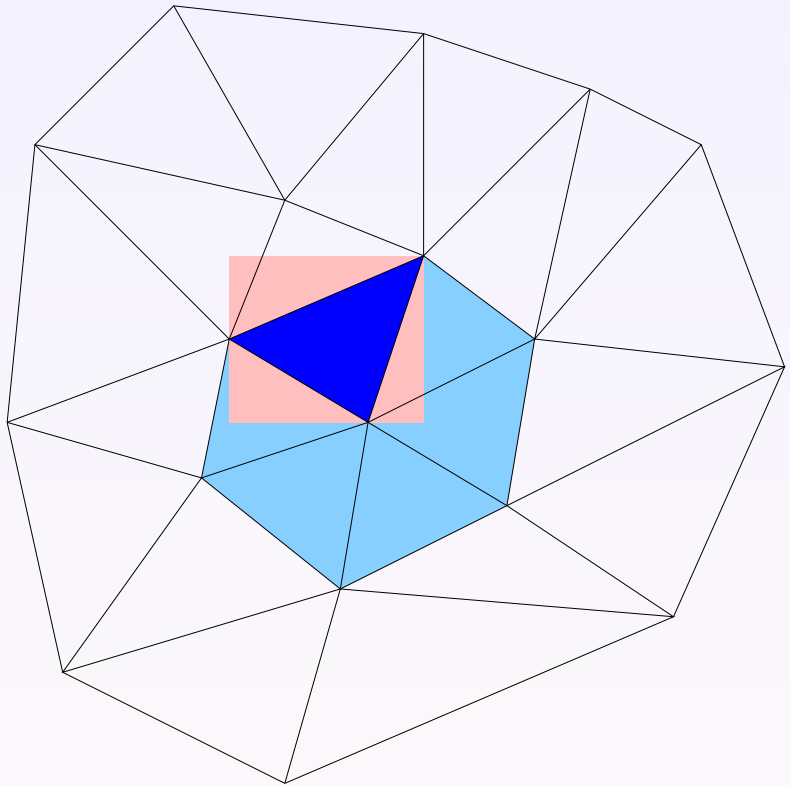
Example: FEM

Example: Polygonal mesh in two-dimensional space with piecewise linear nodal basis functions.



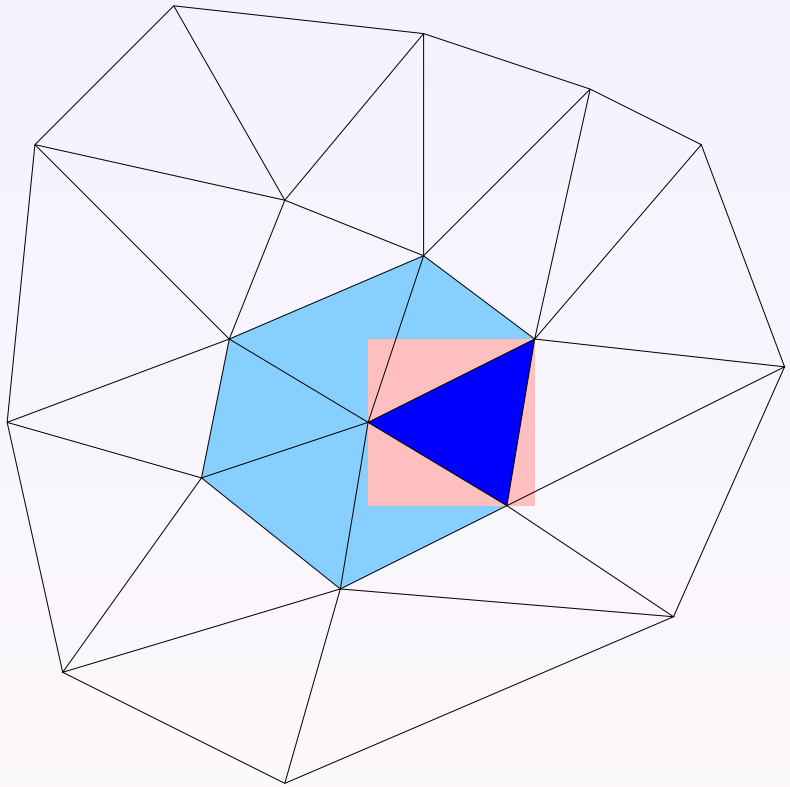
Example: FEM

Example: Polygonal mesh in two-dimensional space with piecewise linear nodal basis functions.



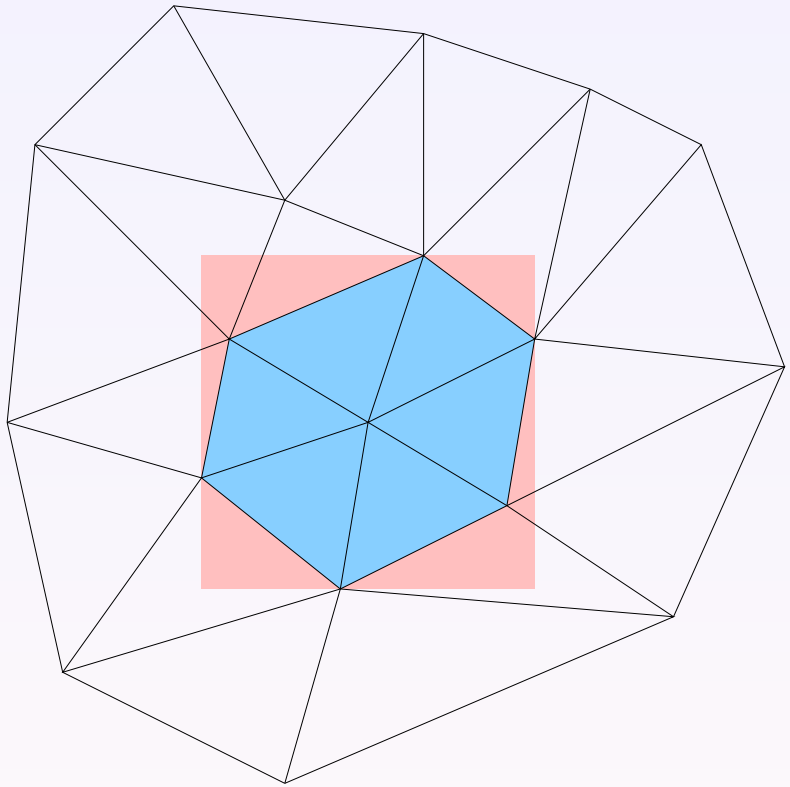
Example: FEM

Example: Polygonal mesh in two-dimensional space with piecewise linear nodal basis functions.



Example: FEM

Example: Polygonal mesh in two-dimensional space with piecewise linear nodal basis functions.



Observation: Nodal points are good characteristic points.
Optimal bounding boxes constructed from single triangles.

Setup of cluster factory

Approach: Represent polygon by array of vertices and array of triangles referencing the vertices, compute points and bounding boxes.

```
for(i=0; i<nv; i++) {
  factory->x[i][0] = vertex[i][0];
  factory->x[i][1] = vertex[i][1];
  factory->smin[i][0] = factory->smax[i][0] = vertex[i][0];
  factory->smin[i][1] = factory->smax[i][1] = vertex[i][1];
}

for(i=0; i<nt; i++) {
  tmin[0] = dmin(vertex[triangle[i][0]][0], dmin(vertex[triangle[i][1]][0],
                 vertex[triangle[i][2]][0]));
  tmax[0] = dmax(vertex[triangle[i][0]][0], dmax(vertex[triangle[i][1]][0],
                 vertex[triangle[i][2]][0]));
  tmin[1] = dmin(vertex[triangle[i][0]][1], dmin(vertex[triangle[i][1]][1],
                 vertex[triangle[i][2]][1]));
  tmax[1] = dmax(vertex[triangle[i][0]][1], dmax(vertex[triangle[i][1]][1],
                 vertex[triangle[i][2]][1]));

  for(j=0; j<3; j++) {
    k = triangle[i][j];
    factory->smin[k][0] = dmin(factory->smin[k][0], tmin[0]);
    factory->smax[k][0] = dmax(factory->smax[k][0], tmax[0]);
    factory->smin[k][1] = dmin(factory->smin[k][1], tmin[1]);
    factory->smax[k][1] = dmax(factory->smax[k][1], tmax[1]);
  }
}
```

Summary

Cluster tree: Constructed based on characteristic points by recursive bisection.

Block cluster tree: Recursive subdivision based on admissibility condition.

Admissibility condition: Evaluated using bounding boxes.

Implementation: `clusterfactory` structure contains simple geometric data, all trees and matrices can be constructed by standardized routines.