Mesh generation

ST5

Bowen ZHU

Date: Oct 2023

Contents

1 Question1

The given function computes the aspect ratio of each element in a mesh.

Code Explanation

```python
def compute_aspect_ratio_of_element(node_coords, p_elem2nodes, elem2nodes):
    nelems = p_elem2nodes.shape[0]-1
    elem_quality = np.zeros((nelems,1), dtype=np.float64)

    for i in range(0,nelems):
        node_indices = elem2nodes[p_elem2nodes[i]:p_elem2nodes[i+1]]
        node_coord = node_coords[node_indices,:]

        # For triangular elements
        if len(node_coord) == 3:
            diffs = node_coord[:, np.newaxis] - node_coord
            distances = np.linalg.norm(diffs, axis=2)
            distances = [distances[0,1], distances[0,2], distances[1,2]]

            d_max = np.max(distances)
            d_min = np.min(distances)
            s = np.sum(distances) / 2
            K = np.sqrt(s*(s-distances[0])*(s-distances[1])*(s-distances[2]))
            r = K/s

            elem_quality[i,:] = d_max / r *(np.sqrt(3)/6)

        # For quadrilateral elements
        if len(node_coord) == 4:
            total = 0
            for j in range(4):
                v1 = node_coord[(j+1)%4,:] - node_coord[j,:]
                v2 = node_coord[(j+2)%4,:] - node_coord[(j+1)%4,:]
```

```
                nip = np.dot(v1, v2) / (np.linalg.norm(v1) * np.
                    linalg.norm(v2))
                total += abs(nip)

            elem_quality[i,:] = 1 - total/4

    return elem_quality
```

Explanation

For **triangular elements**, the code computes the aspect ratio by:

1. Calculating pairwise distances between the nodes of a triangle.
2. Using the semiperimeter, s, of the triangle, which is half of the sum of its sides.
3. Using Heron's formula to calculate the area, K, of the triangle.
4. Calculating the circumradius, r, using the formula:
$$r = \frac{K}{s}$$
5. Computing the aspect ratio using the formula:
$$\text{Aspect Ratio} = \frac{d_{\max}}{r} \times \left(\frac{\sqrt{3}}{6}\right)$$
where $d_{\max}$ is the largest pairwise distance.

For **quadrilateral elements**, the code computes the aspect ratio by:

1. Calculating the normalized inner product of consecutive edges of the quadrilateral.
2. Using the sum of the absolute values of the normalized inner products to calculate the aspect ratio.

2 Question2

The provided function calculates the edge length factor for each element in the mesh.

Code Explanation

```python
def compute_edge_length_factor_of_element(node_coords, p_elem2nodes, elem2nodes):
    nelems = p_elem2nodes.shape[0]-1
    elem_quality = np.zeros((nelems,1), dtype=np.float64)

    for i in range(0,nelems):
        node_indices = elem2nodes[p_elem2nodes[i]:p_elem2nodes[i+1]]
        node_coord = node_coords[node_indices,:]

        # For triangular elements
        if len(node_coord) == 3:
            diffs = node_coord[:, np.newaxis] - node_coord
            distances = np.linalg.norm(diffs, axis=2)
            distances = [distances[0,1], distances[0,2], distances[1,2]]
            distances.sort()

            d_max = np.max(distances)
            d_min = np.min(distances)
            d_mid = distances[1]

            elem_quality[i,:] = d_min/d_mid

    return elem_quality
```

Explanation

For **triangular elements**, the code computes the edge length factor by:

1. Calculating pairwise distances between the nodes of a triangle.
2. Sorting these distances to determine the smallest (min), middle (mid), and largest (max) edges.
3. Using the sorted distances to compute the edge length factor as:

$$\text{Edge Length Factor} = \frac{d_{\text{min}}}{d_{\text{mid}}}$$

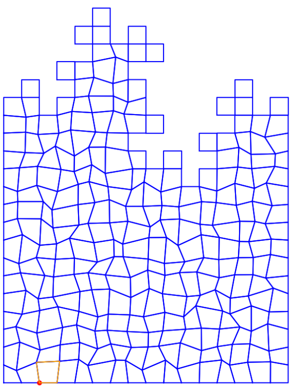
Quadrilateral elements are not considered in this implementation.

3 Question3

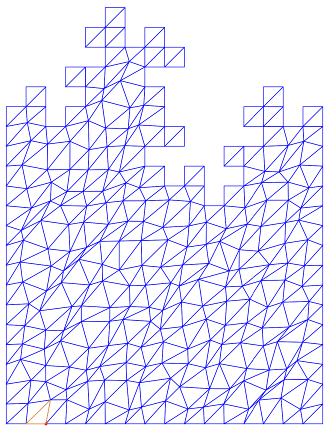
Code Explanation

```python
def random_shift_node(boundary_idx,node_coords,amp):
    # Input: boundary_idx: boundary nodes indices
    # node_coords: node_coords
    # amp: absolute amplitude of uniform sampling
    mask = np.ones(node_coords.shape[0], dtype=bool)
    mask[boundary_idx] = False

    # Generate random shifts
    random_shifts = np.random.uniform(-amp, amp, size=node_coords
        .shape)

    # Apply shifts only to rows specified by the mask
    node_coords[mask] += random_shifts[mask]

    return node_coords
```

Demo



(a) Shift quad mesh



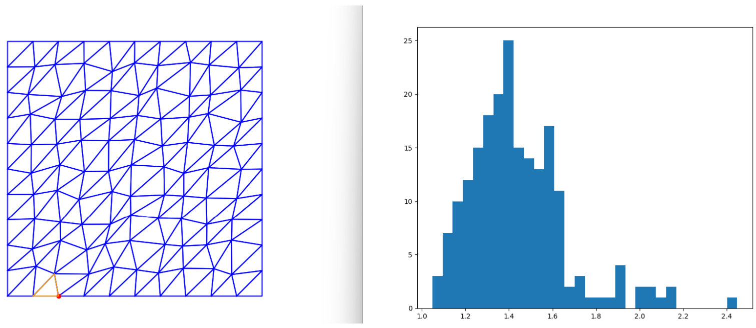
(b) Shift tri mesh

4 Question4

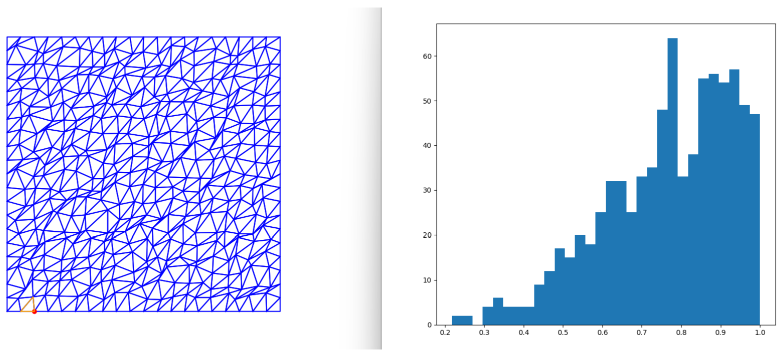
```
def histogram_mesh_quality(nelemsx,percent,quality_type="aspect",mesh_type="tri"):
    # INPUT:
    # nelemsx: X direction element number
    # percent: the relative uniform sampling amplitude wrt element size
    # quality_type: "aspect" or "edge"
    # mesh_type: "tri" or "quad". note "quad" type element do not have "edge" quality type

    if mesh_type=="tri":
        node_coords, p_elem2nodes, elem2nodes,boundary_idx=fractal_mesh_tri(nelemsx,0,1,0)
    elif mesh_type=="quad":
        node_coords, p_elem2nodes, elem2nodes,boundary_idx=fractal_mesh_tri(nelemsx,0,1,0)

    dh=1/nelemsx
    amp=dh*percent
    node_coords=random_shift_node(boundary_idx,node_coords,amp)

    if quality_type=="aspect":
        quality=compute_aspect_ratio_of_element(node_coords, p_elem2nodes, elem2nodes)
    elif quality_type=="edge":
        quality=compute_edge_length_factor_of_element(node_coords, p_elem2nodes, elem2nodes)


    plt.figure(1)
    ax = plt.gca()
    ax.set_aspect('equal')
    ax.axis('off')
    solutions._plot_mesh(p_elem2nodes, elem2nodes, node_coords, color='blue')
    node = 2
    solutions._plot_node(p_elem2nodes, elem2nodes, node_coords,
```

```
        node, color='red', marker='o')
    elem = 2
    solutions._plot_elem(p_elem2nodes, elem2nodes, node_coords,
        elem, color='orange')
    plt.show()


    plt.figure(2)
    plt.hist(quality, bins=30)
    plt.show()
```

demo



(a) Aspect ratio with 10*10 elements and a relative amplitude of 20%



(b) Edge length factor with 20*20 elements and a relative amplitude of 40%

Figure 2: Histogram

5 Question5

The given function computes the barycenter (or centroid) of each element in the mesh.

Code Explanation

```
def compute_barycenter_of_element(node_coords, p_elem2nodes,
    elem2nodes):
    spacedim = node_coords.shape[1]
    nelems = p_elem2nodes.shape[0]-1
    elem_coords = np.zeros((nelems, spacedim), dtype=np.float64)

    for i in range(0, nelems):
        node_indices = elem2nodes[p_elem2nodes[i]:p_elem2nodes[i
            +1]]
        elem_coords[i,:] = np.mean(node_coords[node_indices,:],
            axis=0)

    return elem_coords
```

Explanation

The method used for barycenter computation is based on the principle of averaging the nodal coordinates. For each mesh element:

1. Extract the nodal indices of the current element using the provided indexing structures.

2. Compute the average x and y coordinates of these nodes.

3. This average position represents the barycenter of the element.

Thus, for an element with nodes (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), the barycenter $(\bar{x}, \bar{y})$ is given by:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

6 Question 6

```
def quad2tri(node_coords, p_elem2nodes, elem2nodes):
    nelems=p_elem2nodes.shape[0]-1
    p_elem2nodes_new=[0]
    elem2nodes_new=[]
    for i in range(0,nelems):
```

```
        node_indices=elem2nodes[p_elem2nodes[i]:p_elem2nodes[i+1]]
        idx1=p_elem2nodes[i]
        idx2=p_elem2nodes[i+1]
        if len(node_indices)==4:
            node_diag1=node_indices[1]
            node_diag2=node_indices[3]
            node_indices1=[node_indices[0],node_diag1,node_diag2]
            node_indices2=[node_diag1,node_indices[2],node_diag2]
            elem2nodes_new=elem2nodes_new+node_indices1+node_indices2
            p_elem2nodes_new=p_elem2nodes_new+[(2*i+1)*3]+[(2*i+2)*3]


    return node_coords, p_elem2nodes_new, elem2nodes_new
```

Explanation

The function starts by determining the number of elements from the shape of `p_elem2nodes`. Two lists, `p_elem2nodes_new` and `elem2nodes_new`, are initialized. For each element, if it's a quadrilateral, it's divided diagonally to produce two triangles. The function finally returns the original node coordinates and the new data for the triangular elements.

7 Question 7

This section presents functions for adding and removing nodes and elements in a mesh.

Adding a Node

```
def add_node_to_mesh(node_coords, p_elem2nodes, elem2nodes, nodeid_coords):
    node_coords=np.vstack([node_coords,nodeid_coords])
    return node_coords, p_elem2nodes, elem2nodes
```

Explanation:

A new node, specified by its coordinates ‘*nodeid_coords*‘, is added to the end of the ‘*node_coords*‘ list.

Adding an Element

```python
def add_elem_to_mesh(node_coords, p_elem2nodes, elem2nodes, elemid2nodes):
    elem2nodes = np.vstack([elem2nodes, elemid2nodes])
    new_p_elem = p_elem2nodes[-1] + len(elemid2nodes)
    p_elem2nodes = np.append(p_elem2nodes, new_p_elem)
    return node_coords, p_elem2nodes, elem2nodes
```

Explanation:

A new element, specified by its nodal connectivity ‘elemid2nodes‘, is added to the mesh. The ‘*p_elem2nodes*‘ array is updated to account for the new element’s connectivity.

Removing an Element

```python
def remove_elem_to_mesh(node_coords, p_elem2nodes, elem2nodes, elemid):
    # .. todo:: Modify the lines below to remove one element to the mesh
    start_id=p_elem2nodes[elemid]
    end_id=p_elem2nodes[elemid+1]
    indices_to_delete = range(start_id, end_id)
    size_node=end_id-start_id
    p_elem2nodes = np.delete(p_elem2nodes,[elemid])
    p_elem2nodes[elemid:]=p_elem2nodes[elemid:]-size_node
    elem2nodes=np.delete(elem2nodes,indices_to_delete)

    return node_coords, p_elem2nodes, elem2nodes
```

Explanation:

1. **Determine Indices of Nodes to be Removed**:

- Given an element ID (`elemid`), the function determines the start and end indices in the `elem2nodes` array which defines the nodes associated with the said element. This is achieved using the *p_elem2nodes* array.
- *start_id* gives the starting index, and *end_id* gives the ending index in `elem2nodes` for the given element.
- The indices of nodes associated with this element, which are to be removed from `elem2nodes`, are then defined in the range from *start_id* to *end_id*.

2. **Update Connectivity Lists**:
   - The range of indices (*indices_to_delete*) that correspond to the element in question are removed from the `elem2nodes` array.
   - Similarly, the element's index (`elemid`) is removed from the *p_elem2nodes* array, which adjusts the reference indices for the subsequent elements.
   - Since the element is removed, the indices in the *p_elem2nodes* array after the removed element's index are decremented by the size of the removed element to reflect the new indexing.

3. **Return Updated Lists**:
   - The function then returns the updated node coordinates (*node_coords*, which remains unchanged in this function), *p_elem2nodes*, and `elem2nodes` lists, reflecting the removal of the specified element.

Removing a Node

```python
def build_node2elems(p_elem2nodes,elem2nodes):

    e2n_coef=  np.ones(len(elem2nodes))
    e2n_mtx = scipy.sparse.csr_matrix((e2n_coef, elem2nodes,
        p_elem2nodes))
    n2e_mtx = e2n_mtx.transpose()
    n2e_mtx = n2e_mtx.tocsr()
    p_node2elems = n2e_mtx.indptr
    node2elems=n2e_mtx.indices
    return p_node2elems, node2elems
```

```python
def remove_node_to_mesh(node_coords, p_elem2nodes, elem2nodes,
    nodeid):
```

```python
    # .. todo:: Modify the lines below to remove one node to the mesh
    p_node2elems, node2elems=build_node2elems(p_elem2nodes, elem2nodes)
    p_elem_id_st=p_node2elems[nodeid]
    p_elem_id_ed=p_node2elems[nodeid+1]

    elem_ids=node2elems[p_elem_id_st:p_elem_id_ed]
    e2n_idx = []
    for eid in elem_ids:
        start = p_elem2nodes[eid]
        end = p_elem2nodes[eid+1]
        e2n_idx.extend(list(range(start, end)))
    e2n_idx = sorted(list(set(e2n_idx)))

    # Delete the elements from elem2nodes that use the node
    elem2nodes = np.delete(elem2nodes, e2n_idx)

    sort_elem_ids=sorted(elem_ids)

    for i in sort_elem_ids:
        size=p_elem2nodes[i+1]-p_elem2nodes[i]
        p_elem2nodes[i+1:]=p_elem2nodes[i+1:]-size
    p_elem2nodes = np.delete(p_elem2nodes, elem_ids+1)
    node_idx_chg=np.where(elem2nodes>nodeid)
    elem2nodes[node_idx_chg]=elem2nodes[node_idx_chg]-1
    # Remove the node's coordinates
    node_coords = np.delete(node_coords, nodeid, axis=0)


    return node_coords, p_elem2nodes, elem2nodes
```

Explanation:

Purpose: The function '*build_node2elems*' creates a mapping of each node to the elements it's a part of, which is then used in '*remove_node_to_mesh*' to efficiently remove a node and all the elements attached to it from the mesh.

Functionality:

- **remove_node_to_mesh:**

1. Use build_node2elems to get mapping of each node to the elements it's part of.
2. Identify all elements associated with the node to be removed.
3. Remove these elements and update the p_elem2nodes and elem2nodes lists.
4. Adjust node indices in elem2nodes to account for the removed node.
5. Remove the node's coordinates from node_coords.

8 Question 8

Generating Square Koch Boundary

```python
def fractal_koch_idx(start_x, start_y, size, degree, edge_idx):
    """
    Generate coordinates for a segment of the Koch fractal based
        on the specified parameters.

    Parameters:
    - start_x, start_y: Starting point coordinates.
    - size: Length of the segment.
    - degree: Recursive depth of the fractal.
    - edge_idx: Determines the direction of the segment (0 for
        horizontal, 1 for vertical).

    Returns:
    - List of tuple coordinates forming the segment.
    """

    # Check for minimum size requirement based on the desired
        fractal degree
    if size < 4 * (degree - 1):
        print("can't generate. change size or degree")
        return

    # Base case: first degree of the fractal for horizontal
        direction
    if degree == 1 and edge_idx == 0:
        coord = [(start_x, start_y), (start_x + size, start_y),
```

```
                  (start_x + size, start_y + size), (start_x + 2 * size, start_y),
                  (start_x + 2 * size, start_y - size), (start_x + 3 * size, start_y - size),
                  (start_x + 3 * size, start_y), (start_x + 4 * size, start_y)]
        return coord

    # Base case: first degree of the fractal for vertical direction
    if degree == 1 and edge_idx == 1:
        coord = [(start_x, start_y), (start_x, start_y + size),
                  (start_x - size, start_y + size), (start_x - size, start_y + 2 * size),
                  (start_x + size, start_y + 2 * size), (start_x + size, start_y + 3 * size),
                  (start_x, start_y + 3 * size), (start_x, start_y + 4 * size)]
        return coord
    # Recursive generation for horizontal direction
    if edge_idx==0:
        coord_1=fractal_koch_idx(start_x,start_y,int(size/4),degree-1,0)
        coord_2=fractal_koch_idx(start_x+size,start_y,int(size/4),degree-1,1)
        coord_3=fractal_koch_idx(start_x+size,start_y+size,int(size/4),degree-1,0)
        coord_4=fractal_koch_idx(start_x+2*size,start_y,int(size/4),degree-1,1)
        coord_5=fractal_koch_idx(start_x+2*size,start_y-size,int(size/4),degree-1,1)
        coord_6=fractal_koch_idx(start_x+2*size,start_y-size,int(size/4),degree-1,0)
        coord_7=fractal_koch_idx(start_x+3*size,start_y-size,int(size/4),degree-1,1)
        coord_8=fractal_koch_idx(start_x+3*size,start_y,int(size/4),degree-1,0)
    # Recursive generation for vertical direction
    if edge_idx==1:
```

```
        coord_1=fractal_koch_idx(start_x,     start_y,int(size/4),degree-1,1)
        coord_2=fractal_koch_idx(start_x-size,     start_y+size,int(size/4),degree-1,0)
        coord_3=fractal_koch_idx(start_x-size,start_y+size,int(size/4),degree-1,1)
        coord_4=fractal_koch_idx(start_x-size,start_y+2*size,int(size/4),degree-1,0)
        coord_5=fractal_koch_idx(start_x,     start_y+2*size,int(size/4),degree-1,0)
        coord_6=fractal_koch_idx(start_x+size,start_y+2*size,int(size/4),degree-1,1)
        coord_7=fractal_koch_idx(start_x,start_y+3*size,int(size/4),degree-1,0)
        coord_8=fractal_koch_idx(start_x,start_y+3*size,int(size/4),degree-1,1)
     # Combine all the coordinates
    coord_assemble=coord_1+coord_2+coord_3+coord_4+coord_5+coord_6+coord_7+coord_8


    return coord_assemble
```

Determine Inside or Outside of Boundary

```
def intersection(ray_origin, ray_direction, segment):
    """
    Determine if a ray intersects with a line segment.

    Parameters:
    - ray_origin: The starting point of the ray.
    - ray_direction: The direction of the ray represented as a vector.
    - segment: A tuple representing the endpoints of the line segment.

    Returns:
    - True if the ray intersects the segment.
    - "on boundary" if the ray lies on the segment.
```

```
    - False if there is no intersection.
    """

    # Ray in parametric form: P = R_o + tR_d
    R_o = ray_origin
    vec_x, vec_y = ray_direction
    px, py = ray_origin

    # Extract segment endpoints
    x1, y1 = segment[0]
    x2, y2 = segment[1]

    # Check if ray lies on a horizontal segment
    if y1 == y2 and px >= min(x1, x2) and px <= max(x1, x2) and
       y1 == py:
        return "on boundary"

    # Check if ray lies on a vertical segment
    if x1 == x2 and py >= min(y1, y2) and py <= max(y1, y2) and
       x1 == px:
        return "on boundary"

    # Calculate slope of the ray
    k = vec_y / vec_x

    # Check intersection with vertical segment
    if x1 == x2:
        y_int = k * (x1 - px) + py
        if y_int >= min(y1, y2) and y_int <= max(y1, y2) and
           y_int > py:
            return True
        else:
            return False

    # Check intersection with horizontal segment
    if y1 == y2:
        x_int = (y1 - py) / k + px
        if x_int >= min(x1, x2) and x_int <= max(x1, x2) and
           x_int > px:
```

```
            return True
        else:
            return False
```

```
def is_inside(px, py, nodes):
    """
    Check if a point is inside, outside, or on the boundary of a polygon defined by nodes.

    Parameters:
    - px, py: The coordinates of the point.
    - nodes: A list of tuples representing the vertices of the polygon in sequence.

    Returns:
    - "inside" if the point is inside the polygon.
    - "outside" if the point is outside the polygon.
    - "on boundary" if the point lies on the boundary of the polygon.
    """

    # Define a ray direction (arbitrarily chosen as 80 degrees here)
    ray_direction = (math.cos(math.radians(80)), math.sin(math.radians(80)))

    count = 0
    for i in range(len(nodes)):
        segment = (nodes[i], nodes[(i+1) % len(nodes)])

        # If point lies on the boundary of the polygon
        if intersection((px, py), ray_direction, segment) == "on boundary":
            return "on boundary"

        # If ray intersects a segment of the polygon
        if intersection((px, py), ray_direction, segment) == True:
            count += 1

```

```python
    # Using the ray casting algorithm: if count is odd, point is
        inside; else, outside.
    if count % 2 == 1:
        return "inside"
    else:
        return "outside"
```

Generating Mesh with Fractal Boundary

```python
def remove_adjacent_duplicates(input_list):
    if not input_list:
        return []

    result = [input_list[0]]
    for i in range(1, len(input_list)):
        if input_list[i] != input_list[i-1]:
            result.append(input_list[i])

    return result
```

```python
def fractal_mesh_tri(nelemsx,start,size,degree):
    """
    Generates a triangular mesh based on the fractal Koch curve.
    Removes nodes and elements falling outside the fractal curve.

    Parameters:
    - nelemsx: Number of elements in the x-direction.
    - start: Starting position for the fractal.
    - size: Size of the fractal.
    - degree: Degree of recursion for the fractal.

    Returns:
    - node_coords: Node coordinates of the resulting mesh.
    - p_elem2nodes: Parent elements to nodes mapping.
    - elem2nodes: Element to nodes mapping.
    - indices: Indices of the boundary nodes.
    """
    if size<4*(degree-1):
        print("can't generate. change size or degree")
```

```
        return
    if start+size*4>nelemsx:
        print("can't generate. change size or start")
        return
    xmin, xmax, ymin, ymax = 0.0, 1.0, 0.0, 2.0

    nelemsy=2*nelemsx
    nelems = nelemsx * nelemsy
    nnodes = (nelemsx + 1) * (nelemsy + 1)
    node_coords, node_l2g, p_elem2nodes, elem2nodes = solutions.
        _set_trimesh(xmin, xmax, ymin, ymax, nelemsx, nelemsy)
    # Getting fractal coordinates based on degree.
    if degree!=0:
        fract_coords_idx=fractal_koch_idx(start,nelemsx,size,
            degree,0)
        fract_coords_idx=remove_adjacent_duplicates(
            fract_coords_idx)
    else:
        fract_coords_idx=[]
    # Add bounding box coordinates to fractal coordinates.
    fract_coords_idx.insert(0, (0,nelemsx))
    fract_coords_idx.insert(0, (0,0))
    fract_coords_idx.append((nelemsx,nelemsx))
    fract_coords_idx.append((nelemsx,0))
    fract_coords_idx=remove_adjacent_duplicates(fract_coords_idx)
    # Convert continuous node coordinates to grid indices.
    dh=1/nelemsx
    node_coords_idx=np.round(node_coords/ dh)
    # Remove nodes that are outside of the fractal.
    j=0
    for i in range(len(node_coords_idx)):
        print(str(i)+" over "+ str(len(node_coords_idx)))

        if is_inside(int(node_coords_idx[i,0]),int(
            node_coords_idx[i,1]), fract_coords_idx)=="outside":

             real_idx=i-j

             try:
```

```python

                node_coords, p_elem2nodes, elem2nodes=remove_node_to_mesh(node_coords, p_elem2nodes, elem2nodes, real_idx)
                print(node_coords_idx[i])
                j=j+1
            except:
                continue

        elif is_inside(int(node_coords_idx[i,0]),int(node_coords_idx[i,1]), fract_coords_idx)=="on boundary":
            try:
                boundary_node_coords=np.vstack([boundary_node_coords,node_coords_idx[i]])
            except:
                boundary_node_coords=node_coords_idx[i]

    elem_coords=compute_barycenter_of_element(node_coords, p_elem2nodes, elem2nodes)
    elem_coords_idx=elem_coords/dh
    # Remove elements that are outside of the fractal.
    j=0
    for i in range(len(elem_coords_idx)):
        print(str(i)+" over "+ str(len(elem_coords_idx)))

        if is_inside(elem_coords_idx[i,0],elem_coords_idx[i,1], fract_coords_idx)=="outside":

            real_idx=i-j
            try:
                node_coords, p_elem2nodes, elem2nodes=remove_elem_to_mesh(node_coords, p_elem2nodes, elem2nodes, real_idx)
            except:
                break
            j=j+1
    # Remove nodes not associated with any element.

```

```
    node_idx_with_elem_max = np.max(list(set(elem2nodes)))

    node_coords=node_coords[:node_idx_with_elem_max+1 ]
    node_coords_idx=np.round(node_coords/ dh)
    mask = np.all(boundary_node_coords[:, np.newaxis] ==
       node_coords_idx, axis=2)
    indices = np.argmax(mask, axis=1)

    return node_coords, p_elem2nodes, elem2nodes,indices
```

Explanation:

The function `fractal_mesh_tri` aims to produce a triangular mesh based on the Koch fractal. The process can be broadly categorized into two main steps: the mesh generation and the ray-casting to trim the mesh. The mesh is generated in the region defined by:

$$x \in [0, 1]$$
$$y \in [0, 2]$$

Given the number of elements desired in the x-direction as `nelemsx`, the function computes the number of elements in the y-direction, which is twice that of x-direction, i.e., `nelemsy` $= 2 \times$ `nelemsx`.

The primary mesh generation is performed by the function `solutions._set_trimesh`, which returns the node coordinates, element-to-node mapping, and other essential mesh properties.

Ray-Casting for Trimming the Mesh

To determine which parts of the mesh fall within the fractal's boundary, the algorithm uses a ray-casting approach. This method involves emitting a ray from a point and determining how many times it intersects with the fractal boundary.

For a given point P:

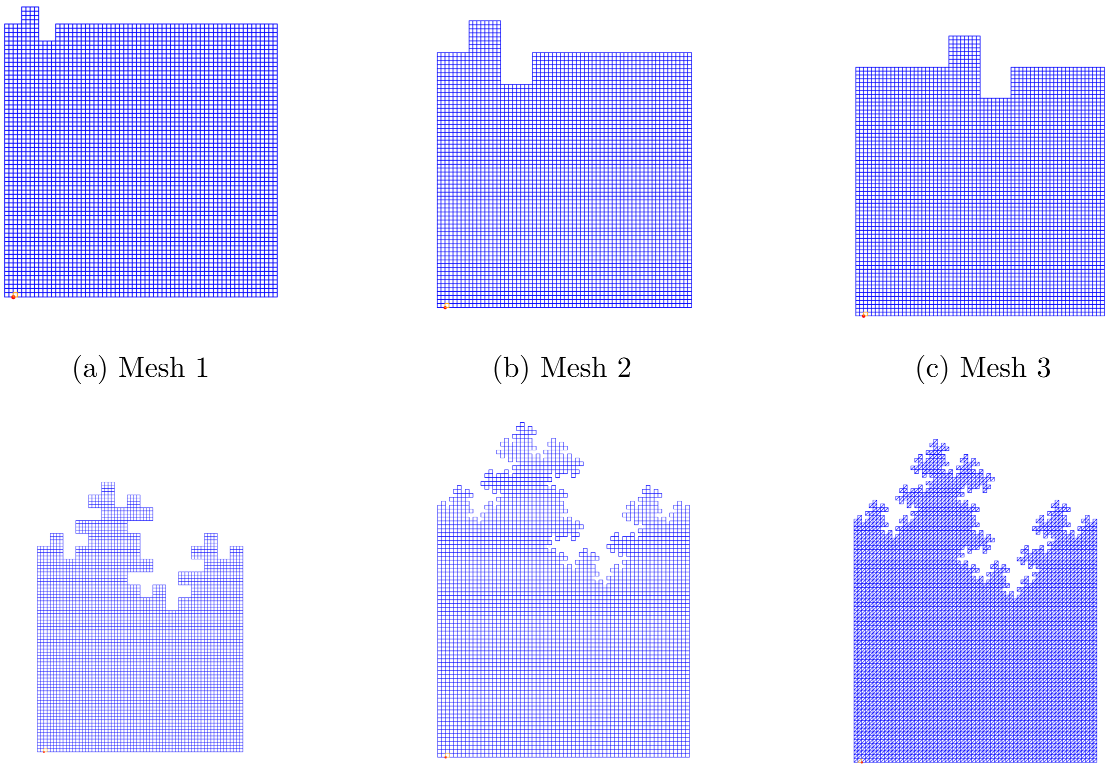
- If the number of intersections is odd, P is inside the fractal.
- If the number of intersections is even, P is outside the fractal.
- If the ray lies on a segment of the fractal, P is on the fractal boundary.

In the code, the ray direction is arbitrarily chosen to be at 80 degrees. For every node in the mesh, a ray is cast, and its relationship with the fractal is determined using the `is_inside` function. Nodes that are found outside the fractal are removed.

The same ray-casting approach is then applied to the barycenter (center of mass) of each element in the mesh to determine if an element lies outside the fractal. Elements outside are subsequently removed.

Note: The ray-casting method's efficiency can significantly impact the performance, especially for a dense mesh or a high-degree fractal.

Demonstration



(a) Mesh 1 (b) Mesh 2 (c) Mesh 3

(d) Mesh 4 (e) Mesh 5 (f) Mesh 6

Figure 3: Fractal mesh of different level, size and position.