Task 1: Geometry classes

The task is to create two classes representing a point and a rectangle in bidimensional space. The code for these two classes is provided in appendix 4.
The rectangle class has been written with two additional methods:

- A method to check if a point object falls within the rectangle
- A method to check if another rectangle object intersects with it

Task 2: Quadtree class

The task is to implement a quadtree data structure class and test it with some randomly-generated and geographical data. The code for this task is presented in appendix 1 (main script) and appendix 5 (class code).

2.1 Random data set

In this subtask, randomly generated data sets are passed to the quadtree generator as per the following table

Number of points	**Type of distribution**	**Bucket size**	**Figure**
30	Normal, (0, 0) mean	5	2.1.1
1000	Uniform	12	2.1.2

and the results visualised as below.

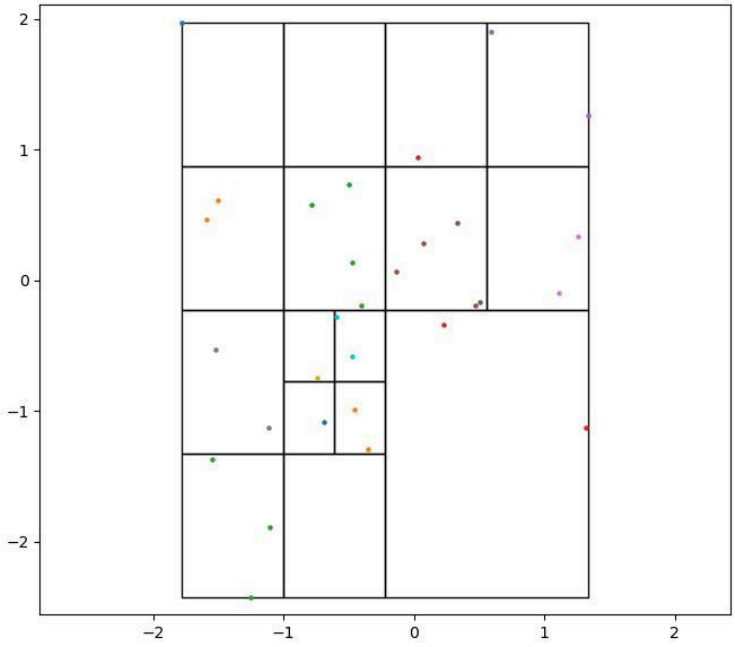

Figure 2.1.1

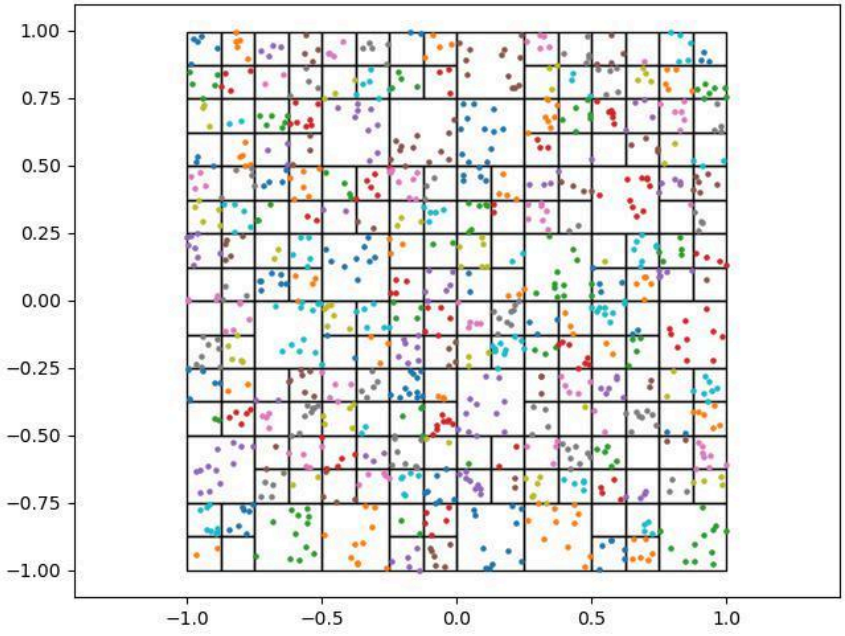

Figure 2.1.2

2.2 Swedish districts

In this subtask, a quadtree is generated with bucket capacity 64 and points supplied in swe.csv. The result is presented in Figure 2.2.1.

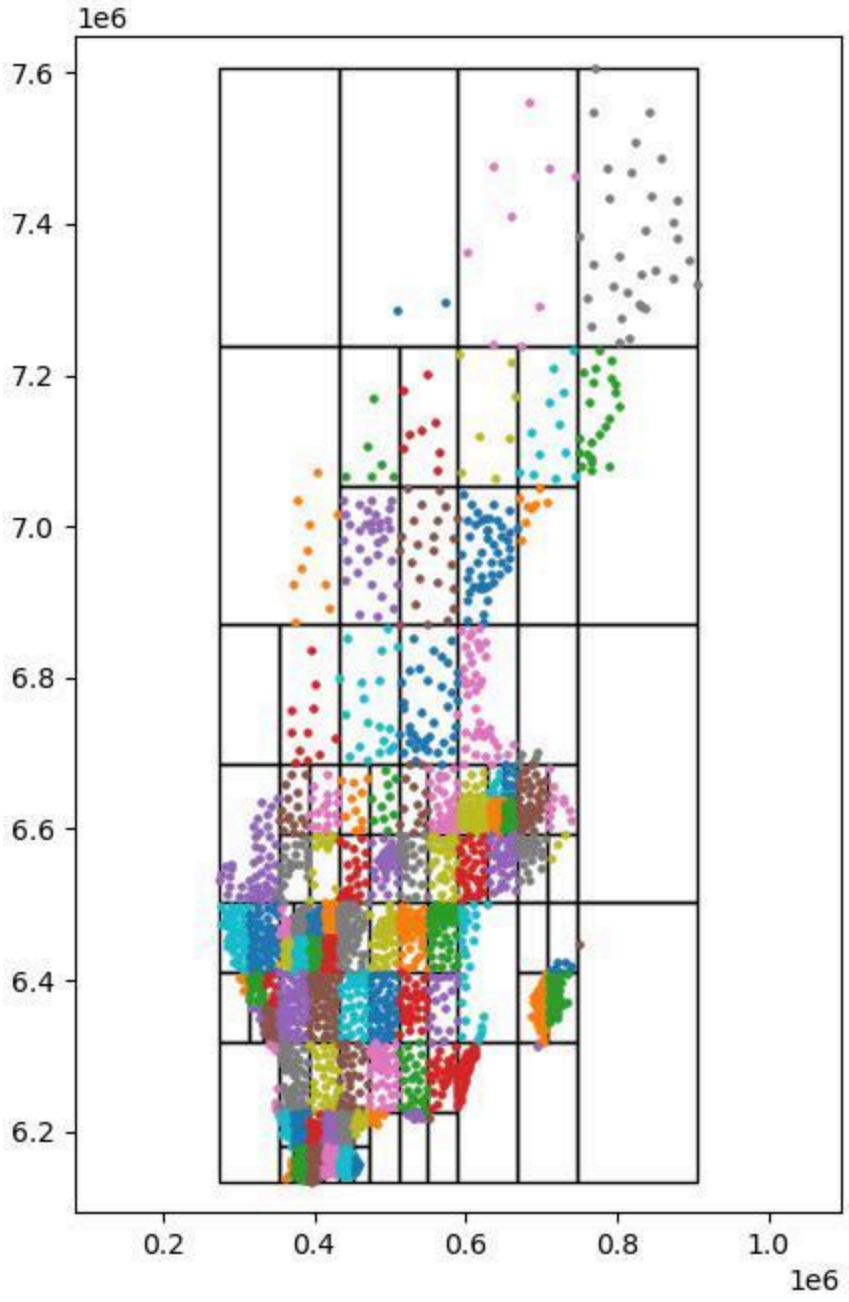

Figure 2.2.1

2.3 Swedish lakes outlines

In this subtask, a quadtree is generated with bucket capacity 640 and points supplied in swelakes.csv. The result is presented in Figure 2.3.1.

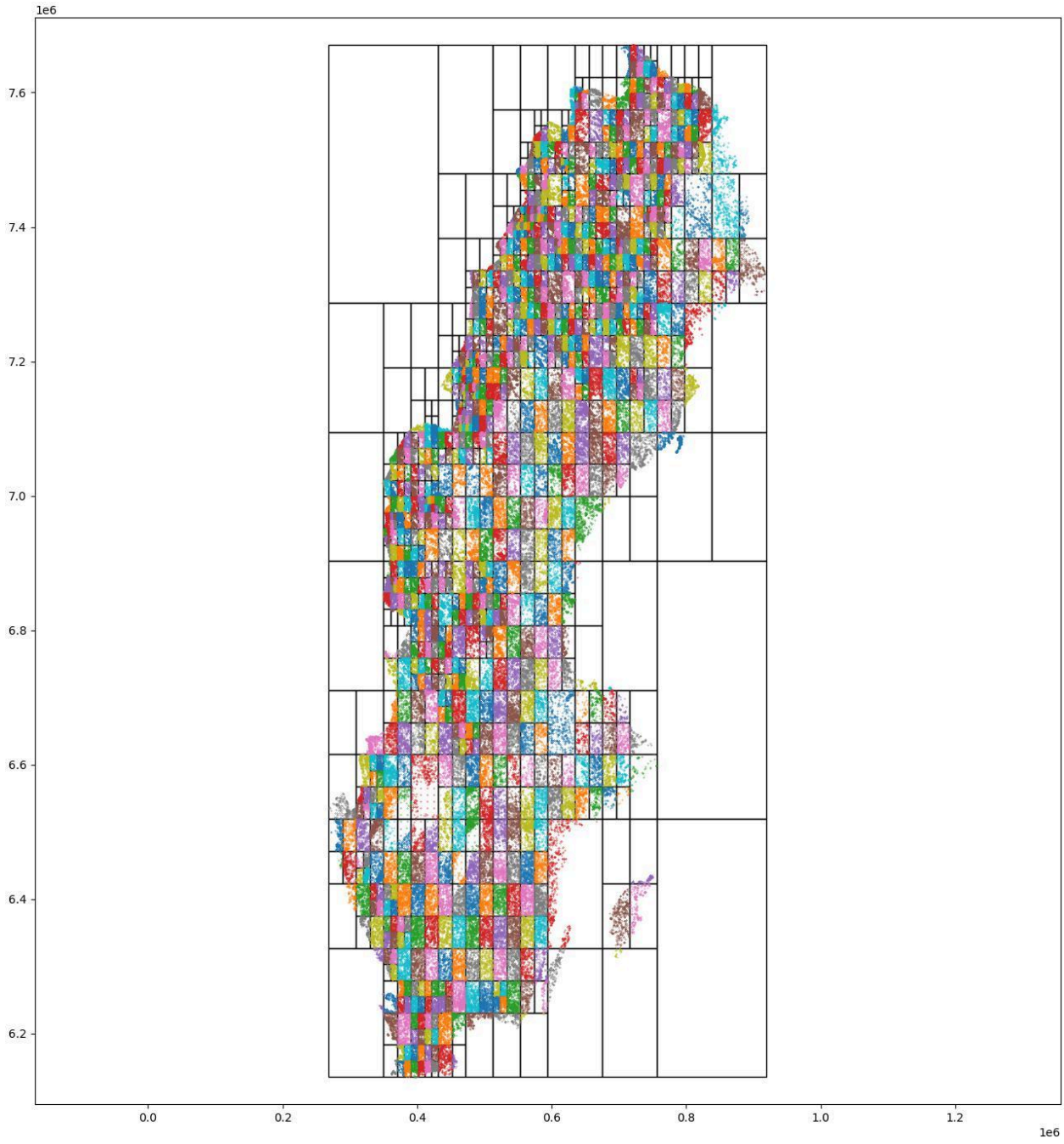

Figure 2.3.1

Task 3: Notes on ownership

Parent Node:

1. Owned by the node that created it
2. Owns the immediate child nodes generated on subdivision

    a. Implies that all subsequent nodes in quadtree are cleaned up when parent node goes out of scope
    b. Access to child node can only happen through parent node
3. First parent node created by the calling function is called the root node

Child Node:

1. Owned by parent node that created it
    a. Implies it is destroyed when parent node is out of scope
2. Owns all further child nodes generated on subdivision
3. Leaf node (Child nodes that do not have their own child nodes) store the data points within their boundaries. They are the owners of the datapoints within the boundaries

Task 4: Query method

In this task, a query method is added to the quadtree class which returns the points lying within a specified rectangle. The data sets are the same as in task 2.1. The code for this method is included in appendix 5. Results are presented in the figures below.

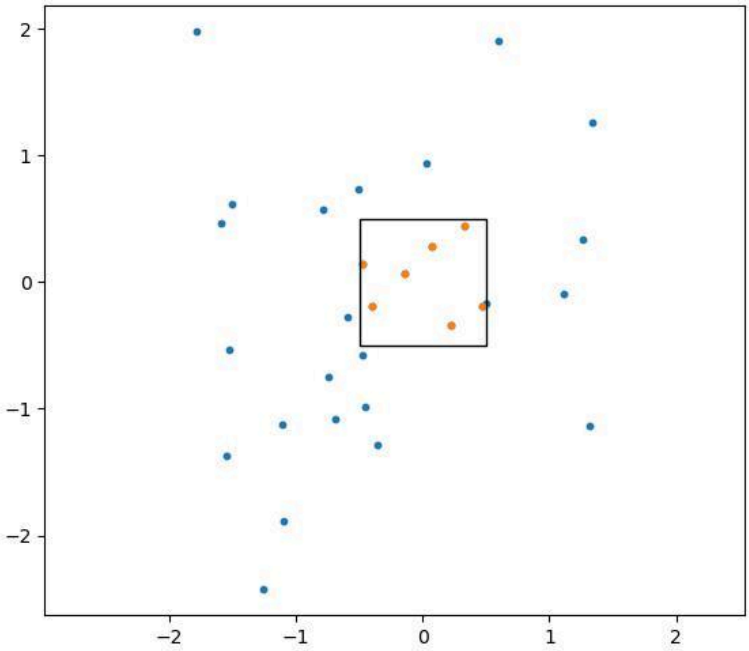

Figure 4.1 - normally-distributed points

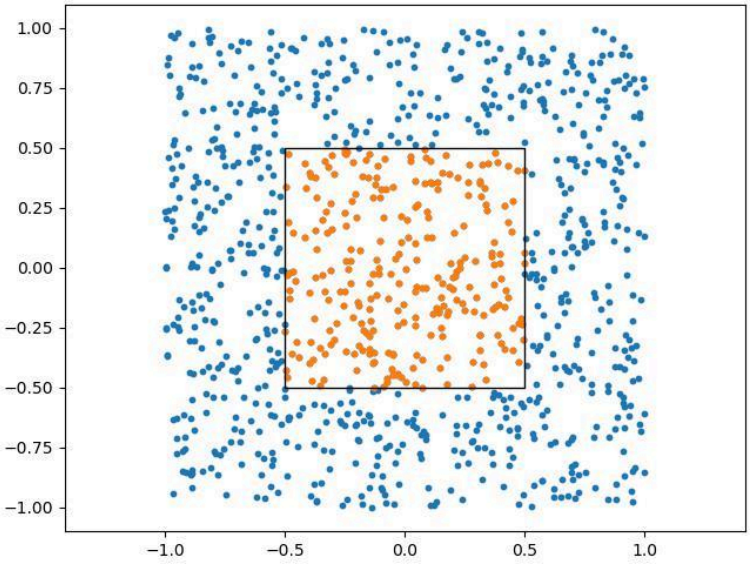

Figure 4.2 - uniformly-distributed points

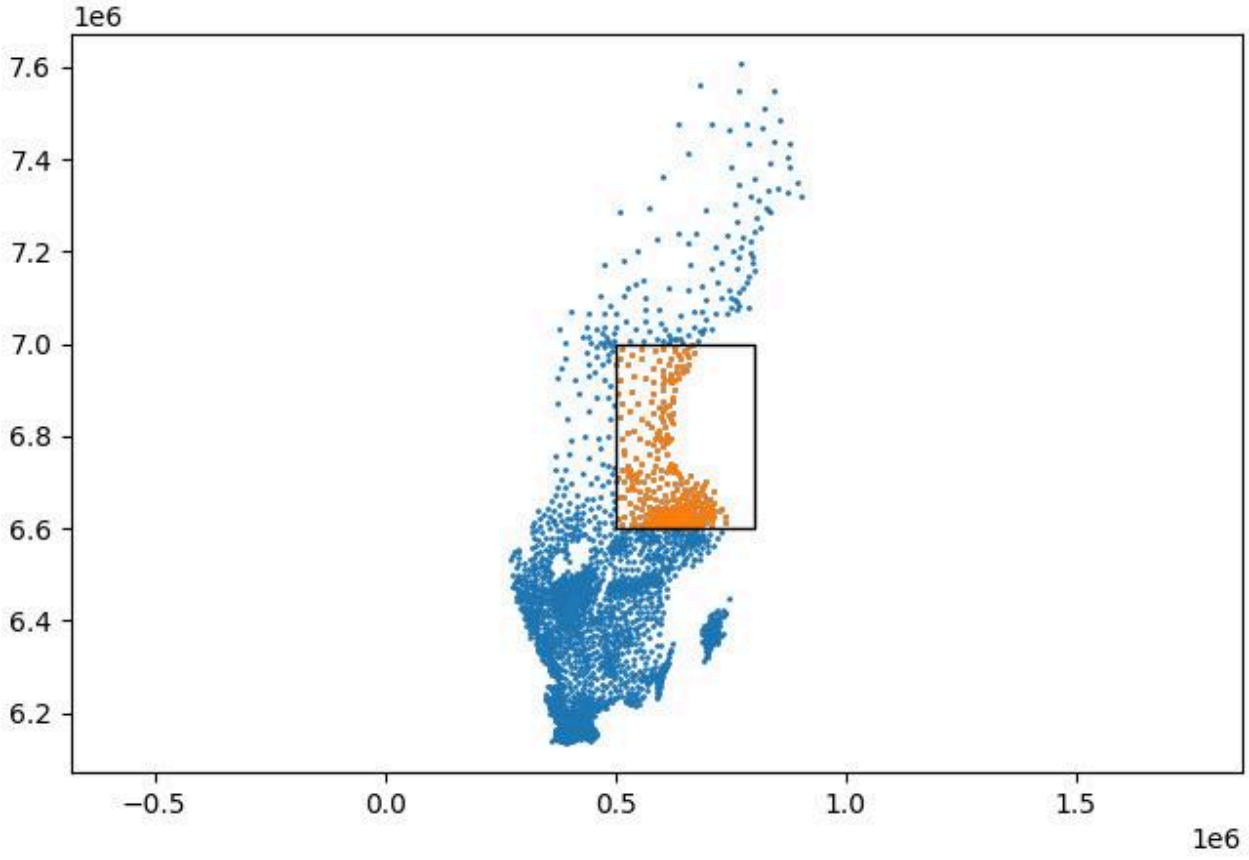

Figure 4.3 - Sweden Districts

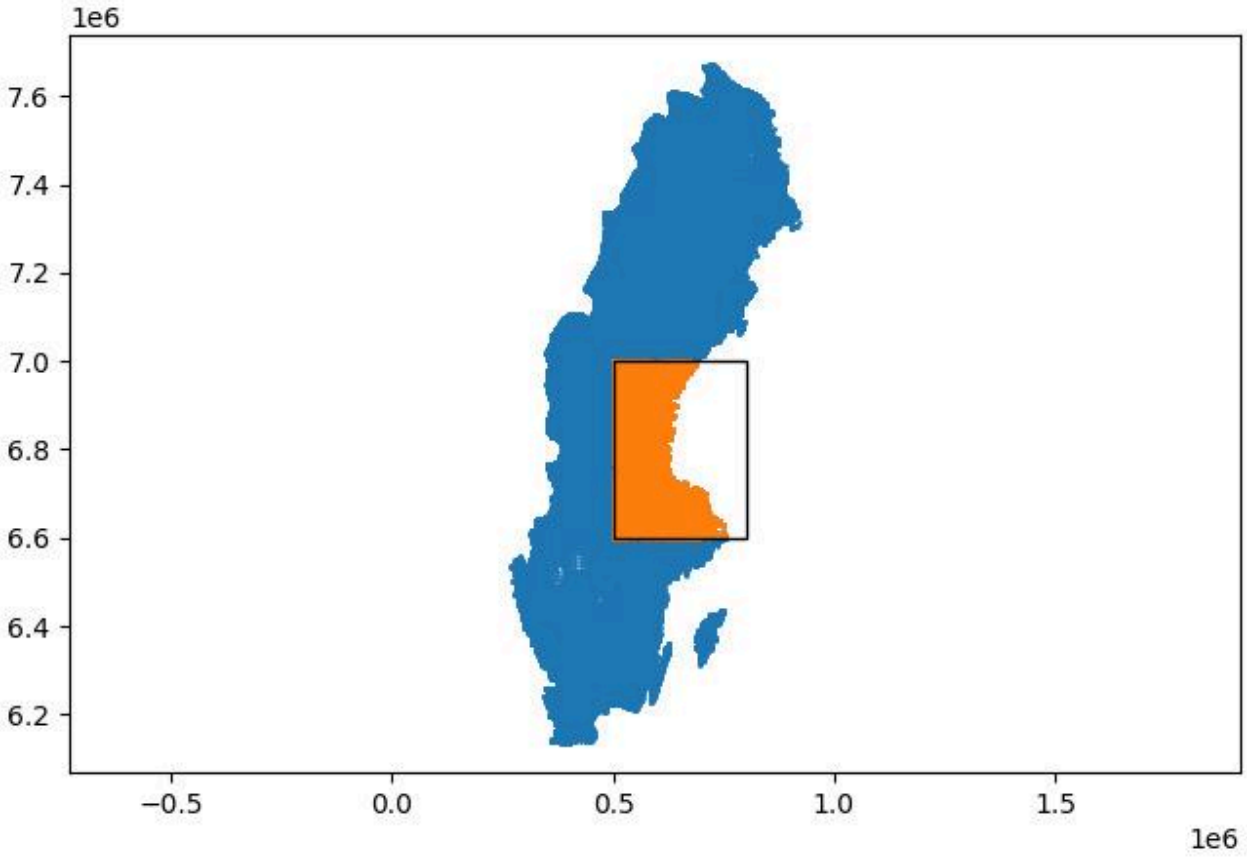

Figure 4.4 - Sweden Lakes

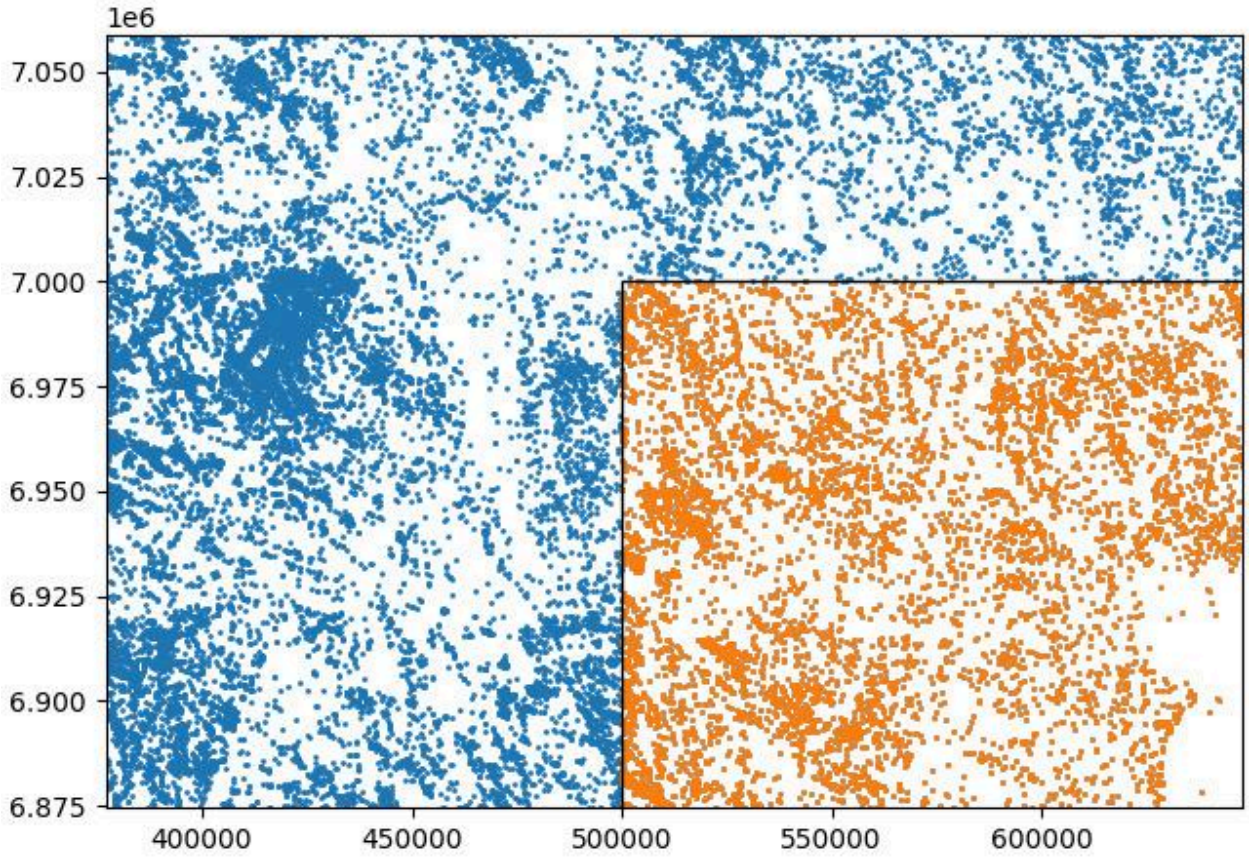

Figure 4.4.1 - Sweden Lakes (Zoomed)

Task 5: Performance measurement

Note:

1. For all datasets, a Quadtree bucket size of 64 was used
2. Randomly Generated dataset uses a uniform point generator with seed 2
    i. RandomGen0 has 10000 data points
    ii. RandomGen1 has 1000000 data points
    iii. RandomGen2 has 10000000 data points
3. Method:Query - Queries points inside provided boundary
4. Method:Quadtree - Builds Quadtree

Quadtree Performance

Comparison of time taken for building and querying data points with multiple datasets

Dataset	Performance	
	Method:Query	**Method:Quadtree**
RandomGen0	797.27 usec	309.98 msec
RandomGen1	1283.02 usec	2208.03 msec
RandomGen2	3878.26 usec	44.25 sec

Performance Comparison Measurements

Comparison of time taken between using query method of quadtree and just iterating through all data points for multiple data sets

Dataset	Performance		Improvement [(1)/(2)%]
	Quadtree Query(1)	**Iterating through data points(2)**	
RandomGen0	797.27 usec	3611.24 usec	22.07%
RandomGen1	1283.02 usec	63.87 msec	2.01%
RandomGen2	3878.26 usec	230.53 msec	1.68%

Conclusion: Quadtrees seem to be significantly more efficient than simple iteration through data points. This becomes more apparent the larger the data set.

Bucket Size Performance Comparisons

Comparison of time taken for building and querying data points with multiple datasets and bucket sizes

Dataset	Method:Query Performance		
	Bucket Size 64	**Bucket Size 10240**	**Bucket Size 102400**
RandomGen0	198.446 usec	674.17 usec	1696.61 usec
RandomGen1	1498.05 usec	4935.77 usec	4956.79 usec
RandomGen2	21.37 msec	23.97 msec	26.01 msec

Dataset	Method:Quadtree Performance		
	Bucket Size 64	**Bucket Size 10240**	**Bucket Size 102400**
RandomGen0	200.357 msec	21.83 msec	5.59 msec
RandomGen1	1739.48 msec	506.82 msec	226.59 msec
RandomGen2	35.82 sec	21.87 sec	7.27 sec

Conclusion: The larger the bucket size, the better the speed performance when building the tree. However, since there are fewer child nodes, more points are placed into child nodes and as a result querying points takes longer.

Appendix

Attached below are the code file descriptions:

1. Main code for tasks
2. Rectangle and Point classes
3. Quadtree class
4. Class definitions

```cpp
/*
 * main.c
 *
 *  Created on: Oct 14, 2024
 *      Author: Shawn, Alessio
 */

#include <chrono>
#include "class_definitions.hpp"
#include "utilities_v3/point_reader.hpp"
#include "utilities_v3/random_points.hpp"
#include "utilities_v3/mpl_writer.hpp"
#include "utilities_v3/timer.hpp"

using namespace std::chrono;
sf::Timer timer;

#define BUCKETSIZE 64

void plot_sort(std::vector<Point>&total_points, Rectangle& rect){

    // Create an MplWriter instance and specify the output filename
    sf::MplWriter<Point, Rectangle> writer("plot_sort.py", 10.0);

    timer.start("Elapsed time to query all points: ");
    // Get query points using rectangle
    std::vector<Point> query_points;

    // Sort through all points and keep those within rectangle
    for (auto const& p : total_points) {
        if(rect.check_point_within_rect(p))
            query_points.push_back(p);
    }

    timer.stop();

    writer << total_points;
    writer << query_points;
    writer << rect;

}

void plot_query(Quadtree& quad, std::vector<Point>&total_points, Rectangle& rect){

    // Create an MplWriter instance and specify the output filename
    sf::MplWriter<Point, Rectangle> writer("plot_query.py", 1);

    timer.start("Elapsed time to query graph: ");
    // Get query points using rectangle
    std::vector<Point> query_points;
    quad.query(rect, query_points);

    timer.stop();

    writer << total_points;
    writer << query_points;
    writer << rect;

    //  // Collect all node boundaries and points
    //  std::vector<Rectangle> boundaries;
    //  std::vector<std::vector<Point>> points;
    //  quad.collectNodes(boundaries, points);
```

```
    //
    //  for (const auto& boundary : boundaries) {
    //      writer << boundary;
    //  }
}

void plot_quadtree(Quadtree& quad){

    // Collect all node boundaries and points
    std::vector<Rectangle> boundaries;
    std::vector<std::vector<Point>> points;
    quad.collectNodes(boundaries, points);

    // Create an MplWriter instance and specify the output filename
    sf::MplWriter<Point, Rectangle> writer("plot_tree.py", 1);

    // Write boundaries and points to the writer for visualization
    for (const auto& boundary : boundaries) {
        writer << boundary;
    }

    for (const auto& pointSet : points) {
        writer << pointSet;
    }
}

void test_random(uint32_t size){

    // Instantiate a random point generator
    sf::RandomPointGenerator<Point> generator{ 2 };

    // Generate random points
    generator.addUniformPoints(size);
    // Get the generated points
    std::vector<Point> pointCollection = generator.takePoints();

    timer.start("Elapsed time to build graph: ");
    // Instantiate a quadtree based on points
    Quadtree quad(pointCollection, BUCKETSIZE);
    timer.stop();

    //Point topRight (0.5,0.5), bottomLeft(-0.5,-0.5);
    Point topRight (0.01,0.01), bottomLeft(-0.01,-0.01);
    Rectangle query_rect(bottomLeft,topRight);

    // Plot quadtree
    plot_quadtree(quad);

    // Plot query points
    plot_query(quad, pointCollection, query_rect);

    // Plot query points
    plot_sort(pointCollection, query_rect);

}

void test_swe(void){

    // Load vector of points from swe.csv file
    std::vector<Point> pointCollection = sf::readCsvPoints<Point>("test_data/swe.csv");

    timer.start("Elapsed time to build graph: ");
```

```
    // Instantiate a quadtree based on points
    Quadtree quad(pointCollection, BUCKETSIZE);
    timer.stop();

    Point topRight (0.8e6,7e6), bottomLeft(0.5e6,6.6e6);
    Rectangle query_rect(bottomLeft,topRight);

    // Plot quadtree
    plot_quadtree(quad);

    // Plot query points
    plot_query(quad, pointCollection, query_rect);

    // Plot query points
    plot_sort(pointCollection, query_rect);

}

void test_swelakes(void){

    // Load vector of points from swe_lakes.csv file
    std::vector<Point> pointCollection = sf::readCsvPoints<Point>("test_data/swelakes.csv");

    timer.start("Elapsed time to build graph: ");
    // Instantiate a quadtree based on points
    Quadtree quad(pointCollection, BUCKETSIZE);
    timer.stop();

    Point topRight (0.8e6,7e6), bottomLeft(0.5e6,6.6e6);
    Rectangle query_rect(bottomLeft,topRight);

    // Plot quadtree
    plot_quadtree(quad);

    // Plot query points
    plot_query(quad, pointCollection, query_rect);

    // Plot query points
    plot_sort(pointCollection, query_rect);

}

int main (){

    test_random(100000);
    //test_random(1000000);
    //test_random(10000000);
    //test_swe();
    //test_swelakes();

    return 1;
}
```

```
/*
 * point_class.cpp
 *
 *  Created on: Oct 23, 2024
 *      Author: user
 */


#include "class_definitions.hpp"

Point::Point(void) : x(0), y(0){}

Point::Point(const double xCoord, const double yCoord) : x(xCoord), y(yCoord){}
```

```
/*
 * classes.cpp
 *
 *  Created on: 18 Oct 2024
 *      Author: shawn, Alessio
 */


#include "class_definitions.hpp"

Rectangle::Rectangle(void){

}

Rectangle::Rectangle(const Point bottomLeft, const Point topRight):
                        bottomLeft(bottomLeft), topRight(topRight){

}


bool Rectangle::check_point_within_rect (Point p){

    if ((p.x >= bottomLeft.x && p.y >= bottomLeft.y) &&
            ((p.x <= topRight.x && p.y <= topRight.y))){
        return 1;
    }
    return 0;
}

bool Rectangle::overlaps (Rectangle rect){

    // CHeck if no intersecting is happening
    bool noHorizontalIntersect = (rect.topRight.x < bottomLeft.x
            || rect.bottomLeft.x > topRight.x);
    bool noVerticalIntersect = (rect.topRight.y < bottomLeft.y
            || rect.bottomLeft.y > topRight.y);
    bool noIntersect = (noHorizontalIntersect && noVerticalIntersect);

    // return true rectangle overlaps with current rectangle
    return (!noIntersect);
}
```

```cpp
/*
 * quadtree_class.cpp
 *
 *  Created on: Oct 18, 2024
 *      Author: Shawn, Alessio Bacchiocchi
 */

#include <iostream>
#include "class_definitions.hpp"


Quadtree::Quadtree(const Rectangle& boundary, unsigned long bucketSize)
: m_bucketSize(bucketSize), m_boundary(boundary), m_node_divided(false),
  m_northWest(nullptr), m_northEast(nullptr),
  m_southWest(nullptr), m_southEast(nullptr){

}

// Constructor automatically calculating boundary
Quadtree::Quadtree(std::vector<Point>& pointCollection, unsigned long bucketSize)
: m_bucketSize(bucketSize), m_node_divided(false),
  m_northWest(nullptr), m_northEast(nullptr),
  m_southWest(nullptr), m_southEast(nullptr) {

  // Initialise the variables with the largest possible values */
  double minX = std::numeric_limits<double>::max();
  double minY = std::numeric_limits<double>::max();
  double maxX = std::numeric_limits<double>::lowest();
  double maxY = std::numeric_limits<double>::lowest();

  // Determine the smallest containing box
  // Parse through vector of points. Assign dimensions of
  // current quadrant based on max value x and y across
  // All points
  for (const Point& p : pointCollection) {
    if (p.x < minX) minX = p.x;
    if (p.y < minY) minY = p.y;
    if (p.x > maxX) maxX = p.x;
    if (p.y > maxY) maxY = p.y;
  }

  // Create point objects with dimensions determined above
  // These will act as boundary of our rectangle
  Point bottomLeft(minX, minY);
  Point topRight(maxX, maxY);

  // Create rectangle object with above dimensions
  m_boundary = Rectangle(bottomLeft, topRight);

  // Insert all points into the quadtree
  for (const Point& p : pointCollection) {
    //insert(p);
    m_points.push_back(p);
  }
```

```
  // Recursively build tree
  build_tree();

}


// Destructor
Quadtree::~Quadtree() {

}

// Recursively build the tree
void Quadtree::build_tree(void){

  // Check if points to be allotted exceed
  // max size of current node
  if (m_points.size() > m_bucketSize) {
    // If node is a leaf, divide it
    if (!m_node_divided)
      subdivide();

    // Allot points in created children
    for (auto const& p : m_points) {

      if (m_northWest->m_boundary.check_point_within_rect(p)){
        m_northWest->m_points.push_back(p);
      } else if (m_northEast->m_boundary.check_point_within_rect(p)){
        m_northEast->m_points.push_back(p);
      } else if (m_southWest->m_boundary.check_point_within_rect(p)){
        m_southWest->m_points.push_back(p);
      } else if (m_southEast->m_boundary.check_point_within_rect(p)){
        m_southEast->m_points.push_back(p);
      }
    }

    // Clear points from current node
    m_points.clear();

    // Continue to build the tree from each node
    m_northWest->build_tree();
    m_northEast->build_tree();
    m_southWest->build_tree();
    m_southEast->build_tree();

  }

}

// Traverse the tree, get the (boundary, points) pair for each node,
// and return a vector of (boundary, points) pairs.
void Quadtree::collectNodes(std::vector<Rectangle>& boundaries,
    std::vector<std::vector<Point>>& points) const {

  // Collect leaf node's boundary and points
```

```
    if (!m_node_divided) {
      boundaries.push_back(m_boundary);
      points.push_back(m_points);
      return;
    }

    // Recursively collect from subdivided quadrants if they exist
    m_northWest->collectNodes(boundaries, points);
    m_northEast->collectNodes(boundaries, points);
    m_southWest->collectNodes(boundaries, points);
    m_southEast->collectNodes(boundaries, points);

  }

  // Subdivide the current node into 4 quadrants
  void Quadtree::subdivide() {

    // Find midpoint of current rectangle boundary
    double midX = (m_boundary.bottomLeft.x
        + m_boundary.topRight.x) / 2.0;
    double midY = (m_boundary.bottomLeft.y
        + m_boundary.topRight.y) / 2.0;

    // Split rectangle into 4 quadrants
    m_northWest = std::make_unique<Quadtree>(Rectangle(
        Point(m_boundary.bottomLeft.x, midY),
        Point(midX, m_boundary.topRight.y)),
        m_bucketSize);

    m_northEast = std::make_unique<Quadtree>(Rectangle(
        Point(midX, midY),
        m_boundary.topRight),
        m_bucketSize);

    m_southWest = std::make_unique<Quadtree>(Rectangle(
        m_boundary.bottomLeft,
        Point(midX, midY)),
        m_bucketSize);

    m_southEast = std::make_unique<Quadtree>(Rectangle(
        Point(midX, m_boundary.bottomLeft.y),
        Point(m_boundary.topRight.x, midY)),
        m_bucketSize);

    // Set flag that quad has been divided
    m_node_divided = true;
  }

  // Used to return all points within a given boundary provided
  void Quadtree::query(Rectangle& rect, std::vector<Point>& pointsInRect) {

    // check if query box intersects with Quadtree boundary
    if (m_boundary.overlaps(rect)) {
      // check if leafnode
      if (!m_node_divided) {
```

```
        // check if points are within boundary and if so, collect them
        for (const auto& point : m_points) {
          if (rect.check_point_within_rect(point)) {
            pointsInRect.push_back(point);
          }
        }
      }
      // if not a leaf node, call query on the children
      else {
        m_northWest->query(rect, pointsInRect);
        m_northEast->query(rect, pointsInRect);
        m_southWest->query(rect, pointsInRect);
        m_southEast->query(rect, pointsInRect);
      }

    }
  }
```

```
/*
 * class_definitions.h
 *
 *  Created on: Oct 18, 2024
 *      Author: Shawn, Alessio
 */

#ifndef CLASS_DEFINITIONS_HPP_
#define CLASS_DEFINITIONS_HPP_

#include <math.h>
#include <vector>
#include <memory>

#include "utilities_v3/mpl_writer.hpp"

class Point {
public:
    // Default Constructor
    Point();

    // Parameter Constructor
    Point(const double xCoord, const double yCoord);

    // Coordinate variables
    double x, y;

private:
protected:

};


class Rectangle {
public:

    //Default constructor
    Rectangle();

    //Constructor that takes bottomleft, topright
    Rectangle(const Point bottomLeft, const Point topRight);

    // Checks if a particular point is within the rectangle
    bool check_point_within_rect (Point p);

    // Check if another rectangle overlaps with this rectangle
    bool overlaps (Rectangle rhs);

    // Rectangle boundaries
    Point bottomLeft, topRight;

private:
protected:
```

```
};

class Quadtree {

public :
    // Constructor for specified boundary
    Quadtree(const Rectangle& boundary, unsigned long bucketSize);

    // Constructor automatically calculating boundary
    Quadtree(std::vector<Point>& pointCollection, unsigned long bucketSize);

    // Destructor
    ~Quadtree();

    // Recursively build the quadtree
    void build_tree(void);

    // For each node, collects the relevant boundary and points
    // Returns a vector of (boundaries, points) tuples
    void collectNodes(std::vector<Rectangle>& boundaries,
                      std::vector<std::vector<Point>>& points) const;

    // Returns all the points that fall within a specified rectangle
    void query(Rectangle& rect, std::vector<Point>& pointsInRect);

private :

    // Subdivide the current node into 4 quadtrants
    void subdivide();

    unsigned long m_bucketSize;      // max. No of points per node allowed
    Rectangle m_boundary;            // boundary of node
    std::vector<Point> m_points;     // points in node
    bool m_node_divided = false;     // Flag to track if rectangle has children

    // Child quadtrees
    std::unique_ptr<Quadtree> m_northWest;
    std::unique_ptr<Quadtree> m_northEast;
    std::unique_ptr<Quadtree> m_southWest;
    std::unique_ptr<Quadtree> m_southEast;
};



#endif /* CLASS_DEFINITIONS */
```