Problem Chosen	2024 MCM/ICM Summary Sheet	Team Control Number
ABCDEF		1111111

2024 Mathematical Contest in Modeling (MCM) Summary Sheet

Summary

This study investigates the data of the geomagnetic monitoring sensors through **matrix** approach, **KD Tree(KDT)** model and **Heuristic Filtering Based on Nearest Neighbor Algorithm(HFBNNA)**. By analyzing data from all sensors in specific range of time, we address three key questions:

Most Representative Data In 4 hours(Problem 1):
We introduce a matrix approach to accurately calculate the difference between the measured data and the initial data of each sensor. We further utilize the calculations in the difference matrix to find the most typical value for each time point in the given 4 hours.

Most Representative Data In 24 hours(Problem 2):
We use the identical approach concerning matrix in problem 1. However, the high requirement of the computer's VRAM caused by the huge amount of data prevent us using the method continuously. As some experiments are taken into practice, KDT are eventually chosen to seek out the consequences of the typical values in 24 hours' time point.

Signal Variation From Specific Object. (Problem 3):

Our findings highlight that xxxx

Keywords: Earth's Magnetic Field; Sensor; Time Point; Matrix; KDT; HFBNNA.

Contents

1 Introduction

1.1 Problem background

The Earth's magnetic field is not static; it is constantly changing due to various natural and external influences. One of the major factors that affect the magnetic field is solar activity, such as solar flares and coronal mass ejections, which interact with the Earth's magnetic field through the solar wind. These interactions can result in geomagnetic storms, which can disrupt power grids, communication systems, and navigation technologies, with potentially serious consequences for modern society.

The Earth's magnetic field is also influenced by processes occurring within the Earth itself. The movement of the liquid iron in the outer core, as well as the complex interactions between the outer core and the mantle, play a significant role in shaping the geomagnetic field. Over time, these internal dynamics lead to changes in the strength, direction, and structure of the magnetic field, which can be observed at the Earth's surface.

To understand and monitor these variations, a network of geomagnetic observatories is required. These observatories would continuously measure changes in the Earth's magnetic field and provide crucial data for scientific research, earthquake prediction, national defense, and various technological applications. Accurate measurements of the magnetic field require sensitive sensors that can detect small variations in magnetic field strength and direction. Some data processing are needed in order to find apparent characteristic and accurate results concerning to the intensity and direction of the Earth's magnetic field.

1.2 Restatement of the Problem

Based on the background information and the constraints outlined in the problem statement, we need to address the following issues:

- How to select the best sensors to represent the magnetic field of a specific time.
- How to perform the selection algorithm efficiently on a large amount of data.
- How to extract the original signal from the data of many sensors and eliminate noises.

1.3 Our work

Our work is illustrated in Figure 1.

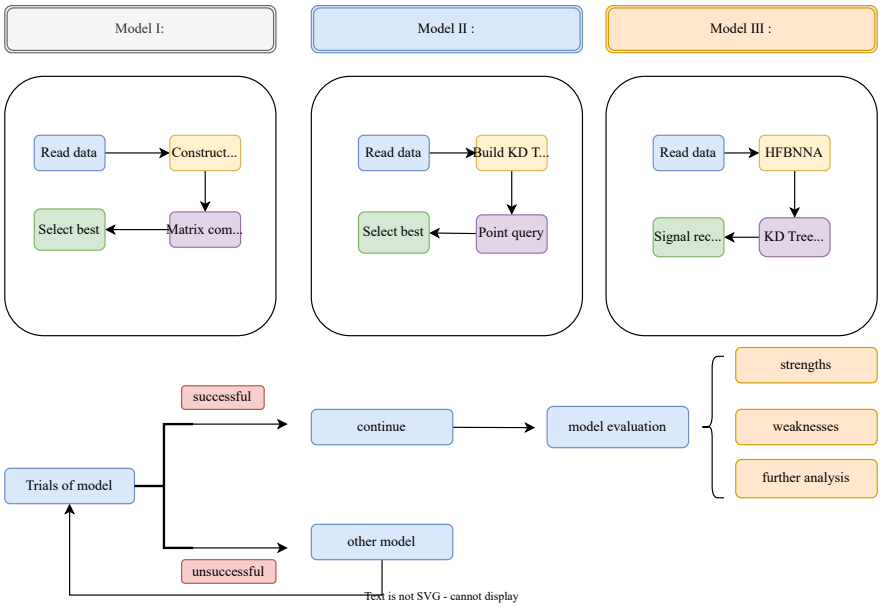

Figure 1: Our work

2 Assumption and Justifications

Given that practical problems often involve numerous complex factors, the first step is to make reasonable assumptions in order to simplify the model. Each assumption is then followed by a corresponding explanation.

- **Assumption**: The geomagnetic field and magnetic field from a specific objects are uniform in space.
- **Justification**: To make the variation of data be consistent with data from sensors from different unknown locations only uniform magnetic field's model can be established with relative accurate data.
- **Assumption**: The data from the sensors is accurate enough to model the magnetic field.
- **Justification**: The amount of the sensors is large and for the geomagnetic data is nearly consistent.

3 Model Establishment

3.1 Data processing

To speed up data I/O, we first combine all the sensor data into a dictionary format stored in JSON file. Then, the data is rearranged as a matrix, its dimension is the time stamps, the second dimension represents every sensors, the final dimension contains data from all channels.

3.1.1 Model for problem 1 & 2

1) Differential matrix

2) KD Tree

Considering problem 2 that the sensors data from 24 hours is requied to be processed, the size of matrix constructed for differential calculation is extremely large (4311×4311). It requires approximately 25GB of GPU memory if the data is stored in 64-bit floting point form which exceed the limit of household PC. Although we rent a A800 GPU to obtain the accurate results, this is not pratical in because of high cost.

References

Appendix A

```
#ifndef KD_TREE_H
#define KD_TREE_H
#include <cmath>
#include <limits>
#include <vector>
#include <algorithm>
using LD=long double;
struct Point {
    LD x{},y{},z{};
    int index_in_sensor{};
};
using T_Point_VEC=std::vector<Point>;
class KdTree {
public:
    struct KDNode {
        Point point;
        KDNode* left;
        KDNode* right;
          explicit KDNode(const Point& point) : point(point), left(nullptr),
right(nullptr) {}
    };

private:
    static LD distance(const Point &p1, const Point &p2) {
        return sqrt(pow((p1.x - p2.x),2) +
               pow(abs(p1.y - p2.y),2)  +
               pow(abs(p1.z - p2.z),2));
    }
    static bool compare(const Point &p1, const Point &p2, int depth) {
        if(depth%3==0) {
            return p1.x<p2.x;
        }
        if(depth%3==1) {
            return p1.y<p2.y;
        }
        return p1.z<p2.z;
    }
    KDNode *root{nullptr};
    static KDNode* r_build_tree(std::vector<Point> &points, int depth=0) {
        if(points.empty()) {
            return nullptr;
        }
        int axis=depth%3;
        std::ranges::sort(points, [depth](const Point &p1, const Point &p2) {
```

```
            return compare(p1,p2,depth);
        });
        int median=points.size()/2;
        auto *new_node=new KDNode(points[median]);
        std::vector left(points.begin(), points.begin()+median);
        std::vector right(points.begin()+median+1, points.end());
        new_node->left=r_build_tree(left, depth+1);
        new_node->right=r_build_tree(right, depth+1);
        return new_node;
    }
    static void r_find_nearest(KDNode *root, const Point &query, KDNode*& best,
LD &bestDist, int depth=0) {
        if (root == nullptr) {
            return;
        }

        // Use squared distance for more precision
        LD dist = distance(query, root->point);
        if (dist < bestDist) {
            bestDist = dist;
            best = root;
        }

        int axis = depth % 3;
        KDNode *next_node = nullptr;
        KDNode *other_node = nullptr;

        // Decide which side to search first
        if ((axis == 0 && query.x < root->point.x) || (axis == 1 && query.y <
root->point.y) || (axis == 2 && query.z < root->point.z)) {
            next_node = root->left;
            other_node = root->right;
        } else {
            next_node = root->right;
            other_node = root->left;
        }

        // Explore the next node (recursive search)
        r_find_nearest(next_node, query, best, bestDist, depth + 1);

        // Check if we need to explore the other side
        if ((axis == 0 && abs(query.x - root->point.x) < bestDist) ||
            (axis == 1 && abs(query.y - root->point.y) < bestDist) ||
            (axis == 2 && abs(query.z - root->point.z) < bestDist)) {
            r_find_nearest(other_node, query, best, bestDist, depth + 1);
            }
```

```
    }


public:
    void build(std::vector<Point> &points) {
        root = r_build_tree(points);
    }
    struct NearResult {
        Point best_point;
        LD best_dist{};
    };
    [[nodiscard]] NearResult find_nearest(const Point &query) const {
        LD best_dist{std::numeric_limits<LD>::infinity()};
        KDNode *best_node=nullptr;
        r_find_nearest(root, query, best_node, best_dist, 0);
        if(best_node!=nullptr) {
            return {best_node->point, best_dist};
        }
        return {Point(0,0,0),-1};
    }
};
#endif //KD_TREE_H
```