Laurel Purcell

DS210 Final Project Report

15 December 2024

Near-Earth Object Analysis

Introduction

Objects such as asteroids and comets that pass near the Earth have the potential to be hazardous to human life. If one of these objects were to pass too near to Earth or even collide with us directly, it could be devastating to humanity. NASA's Jet Propulsion Laboratory records all of these asteroids and comets, known as "near-Earth objects", or NEOs, to track information about their orbital paths, distance from Earth, and when it will pass the closest. Analyzing this data can reveal how close these objects may pass to Earth and provide insight into identifying characteristics of NEOs that pass closest to Earth. This in turn could help predict if one of these NEOs were on a collision path with Earth and how we could counter that.

This dataset contains information about 2,537 different NEOs that passed near Earth between January 1, 2023, and September 25, 2024. There are 10 columns in the dataset, explained below:

des (Designation): A unique identifier for the asteroid or comet. This can be a numeric designation, a provisional designation, or a name if the object is well-known.

orbit_id (Orbit Identifier): This is the specific identifier for the orbit calculation used for this close approach. Orbit determinations can be updated, so this ID ensures you are referencing the exact calculation used for a particular approach.

jd (Julian Date): The Julian Date representing the exact time of the closest approach. Julian Date is a continuous count of days since the beginning of the Julian Period, often used in astronomical calculations.

cd (Calendar Date): The calendar date and time (in UTC) of the closest approach, formatted in a human-readable form.

dist (Nominal Distance): The nominal (best estimate) distance of the object from Earth at the moment of closest approach, measured in astronomical units (AU).

1 AU: The average distance between Earth and the Sun (~149.6 million km).

dist_min (Minimum Distance): The minimum possible distance from Earth at closest approach, accounting for uncertainties in the orbit. This distance is measured in AU.

dist_max (Maximum Distance): The maximum possible distance from Earth at closest approach, accounting for uncertainties in the orbit. This distance is also measured in AU.

v_rel (Relative Velocity): The relative velocity of the object as it passes Earth, measured in kilometers per second (km/s). This value represents how fast the object is moving relative to Earth at the moment of closest approach.

v_inf (Velocity at Infinity): The object's velocity relative to a massless object (the theoretical velocity it would have if it approached from an infinite distance). This is also measured in kilometers per second (km/s).

t_sigma_f (Time Uncertainty): The uncertainty in the time of the closest approach, provided in days, hours, and minutes. This accounts for the potential errors in orbit calculations, indicating how much the exact time of the approach could vary.

(Descriptions of each column come from Kaggle, where the dataset was sourced from).

Analysis

For this dataset, I wanted to determine the hazard level of an asteroid and rank it in terms of how potentially dangerous it could be to Earth. To do this, I chose to use this with the minimum distance that the asteroid could pass by Earth, at its nearest, and the velocity that it would have relative to Earth. The dataset contains the distance that the asteroid will pass by Earth, as well as the uncertainty associated with this distance. I chose to use the minimum possible distance because that would be the nearest it could possibly pass, which would ensure no error. At this speed, the velocity of the asteroids are generally all fast enough that minor differences would not minimize the potential effects on Earth if it were to make impact. However, a slower-moving asteroid may be easier to divert or engage in other countermeasures if we were to discover that it were on a collision path with Earth. Therefore, I included higher velocities in my analysis of the hazard that an asteroid poses.

I clustered the asteroids in terms of their relative risk, from “Negligible” to “Highest.” “Highest” is still quite far from Earth, but it is the most potentially hazardous relative to the other asteroids. None of these come near enough to pose any real threat, but the analysis on this small sample could be

extrapolated to a larger dataset and used to evaluate the hazard of asteroids that may pass much nearer to Earth.

The hazard ranking is effective in this case because it can directly compare the asteroids using a quantitative scale of danger. It factors in both proximity and velocity. It sorts these efficiently in descending order. Clustering, in this instance, allows us to easily group the asteroids into a group based on their hazard. All of the highest-risk asteroids are contained neatly in one group. We can easily see how the asteroids might be distributed across risk levels, and even adjust the ranges to include a more rigorous hazard definition. Using graph-based comparisons uses the nodes and weights to easily compare the asteroids and uncover relationships between asteroids that might otherwise be impossible to see.

***csv_reader** module; Code Explanation*

I first created a module to hold my function that reads the CSV file. I titled it **csv_reader.rs**. This helps to keep my code organized so that the CSV reading function is isolated, and the main module can contain only functions that are directly related to the analysis of the data.

```
// import
use csv::ReaderBuilder;
use std::error::Error;
```

I begin by importing the two utilities that I will need to build the CSV reader. **use csv::ReaderBuilder** is a utility from the **csv** crate that helps to create functions to read CSV files. This allows us to to import the traits from this module to use in our own function.

use std::error::Error helps us to represent errors in Rust, which we will implement into our CSV reading function. This helps us manage potential errors and handle them gracefully, so the program does not crash if there are missing datapoints in the CSV file. Although the dataset we are working with is already clean, this is very useful in situations where the dataset may contain missing data, as we can use the error module to ignore these gaps.

```rust
// define a struct to represent the rows of the asteroid dataset
#[derive(Debug)]
1 implementation
pub struct AsteroidData {
    // each type is a given data type based on their function in the dataset
    pub des: String,
    pub orbit_id: String,
    pub jd: f64,
    pub cd: String,
    pub dist: f64,
    pub dist_min: f64,
    pub dist_max: f64,
    pub v_rel: f64,
    pub v_inf: f64,
    pub t_sigma_f: String,
}
```

Next in **csv_reader.rs** I define my struct for the dataset. It begins with **#[derive(Debug)]**, which is a trait that allows for the struct to print and format when debugging. Next I define the struct **AsteroidData**. I preface the struct with **pub** because this makes the struct public, allowing me to access it from my **main.rs** module; I do the same for each of the fields in the function. For each column in the dataset, I include a field that corresponds to the name of it. For these fields, I have to decide which type would be most appropriate for storing the data from that column. I annotate the field with these types.

Rust is a statically typed language, so each value must have a data type to allow Rust to work with that data. This must be specified at the time of compilation. It is our responsibility to assign each value with a type annotation so that Rust can function properly. Here, I annotate the columns **des** (the asteroid's designation), **orbit_id** (another identifier for the asteroids)**, cd** (calendar date of the asteroid passing near Earth)**,** and **t_sigma_f** (the uncertainty associated with the time that the asteroid will pass nearby) with the **String** type, as these are labeled with names in the data set and it is not necessary to store them in a numerical dataset. The columns **jd** (Julian Date), **dist** (average estimated distance that the asteroid will pass by Earth, at its nearest), **dist_min** (the closest distance that the asteroid might pass by Earth, at its nearest), **dist_max** (the furthest distance that the asteroid might pass by Earth, at its nearest), **v_rel** (the relative velocity of the asteroid as it passes near Earth), and **v_inf** (the relative velocity of the asteroid as it passes through infinity) all take the data type **f64** because they all deal with numerical data.

f64 is for floating-point numbers, which specifically deal with decimal points in the calculations and analysis. This allows each of these columns to be stored as **AsteroidData** struct in a vector in the **read_csv** function later in my code.

In my analysis, not each of the columns was incorporated. Some, like the Julian Date that the

```rust
// creates a function to parse the CSV file into a vector of AsteroidData
pub fn read_csv(file_path: String) -> Result<Vec<AsteroidData>, Box<dyn Error>> {
    let mut reader: Reader<File> = ReaderBuilder::new() ReaderBuilder
        .has_headers(yes: true) &mut ReaderBuilder
        .from_path(&file_path)?; // fixed variable name that calls the local path

    let headers: StringRecord = reader.headers()?.clone();
    let des_idx: usize = headers.iter().position(|h: &str| h == "des").ok_or(err: "Missing header 'des'")?;
    let orbit_id_idx: usize = headers.iter().position(|h: &str| h == "orbit_id").ok_or(err: "Missing header 'orbit_id'")?;
    let jd_idx: usize = headers.iter().position(|h: &str| h == "jd").ok_or(err: "Missing header 'jd'")?;
    let cd_idx: usize = headers.iter().position(|h: &str| h == "cd").ok_or(err: "Missing header 'cd'")?;
    let dist_idx: usize = headers.iter().position(|h: &str| h == "dist").ok_or(err: "Missing header 'dist'")?;
    let dist_min_idx: usize = headers.iter().position(|h: &str| h == "dist_min").ok_or(err: "Missing header 'dist_min'")?;
    let dist_max_idx: usize = headers.iter().position(|h: &str| h == "dist_max").ok_or(err: "Missing header 'dist_max'")?;
    let v_rel_idx: usize = headers.iter().position(|h: &str| h == "v_rel").ok_or(err: "Missing header 'v_rel'")?;
    let v_inf_idx: usize = headers.iter().position(|h: &str| h == "v_inf").ok_or(err: "Missing header 'v_inf'")?;
    let t_sigma_f_idx: usize = headers.iter().position(|h: &str| h == "t_sigma_f").ok_or(err: "Missing header 't_sigma_f'")?;
```

asteroid will pass the closest to Earth, were not relevant to this particular analysis. However, I elected to keep each of them in to allow me to properly build and represent my complete dataset within the code, as well as allowing me the freedom to perform additional analysis in the future if I need those columns.

After I have established my struct, I create the function **read_csv**. The header function will extract the header row and ensure that all of the column headers that we need are present in the dataset. It also locates the column index of each of these headers.

Like the struct, I make this public with **pub** so that it can be accessed from my main module. I used **file_path: String** to denote it as a string argument, because the file path to my CSV file that I will eventually input into my main function will be a string. This represents the location of the CSV file for my function. Then I store the result of this in **Result<Vec<AsteroidData>, Box<dyn Error>>.** This means that it will parse the dataset and return it as a vector of AsteroidData structs if the dataset is okay, or it will return an error message if something goes wrong. This error message is a dynamic error type, which means that regardless of the error type, the program will be able to handle it.

Next, I use **let mut reader = ReaderBuilder::new()** to initialize my CSV reader. The reader will be stored as the variable name **reader**, which is mutable because the vector will need to be changed as the function iterates over the rows of the dataset and keeps track of the part of the file where it is. Additionally, several methods within the reader function internally modify the state of the reader function, so it must be mutable in order for this to work. **.has_headers(true)** specifies to the function that the CSV file we will be accessing contains a row of headers. **.from_path(&file_path)?** allows us to open the CSV file on the path defined in **file_path**. **from_path** takes an argument reference to a string slice. I used **&** before **file_path** in this case to allow me to pass a reference to it. .**from_path** expects this to be passed as a reference because it only needs to look at the data of the string in order to access the desired CSV file, and we do not need to copy the dataset. If we passed **file_path** directly, it would move the ownership of the string into this function, which would mean that we could not use **file_path** after this occasion. We can reuse the variable. Using the **?** operator at the end of the function propagates any errors that might occur.

let headers = reader.headers()?.clone(); parses the headers of the CSV file. **reader.headers()** reads the header rows of the dataset and returns a reference to the header values. This also ties the reference to the lifetime of the **reader** function, so after the reader object goes out of scope, it will not be usable. **.clone()** copies the headers of the dataset for later use. This creates an owned copy of the headers, which is independent of the reader. With this, it allows the headers to be stored and used later regardless of the **reader**, so even when the **reader** mutates (such as when iterating over rows), the headers may still be used without conflict.

let des_idx = headers.iter().position(|h| h == "des").ok_or("Missing header 'des'")?; is the first in the list of fields for the column names. Each column follows the same format for finding the column indexes. It begins with **headers.iter()**, which calls on the headers for the dataset to iterate over them. **headers** represents a string of the header row from the CSV file. **.iter()** creates an iterator to go over the fields; each is a string slice that represents one of the names of the columns. **.position(|h| h == "des")**, in this case, locates the index of the header for the column “des”; this is different depending on

the name of the column. It uses a closure as its argument, which checks if the current element within the iterator is equal to the column name string. **.position()** runs through each of the items within the iterator until a match is found, which it then returns as the index of the first element that matches. This will either be **Some(index)** if valid or **None** if it does not find the column name. It uses the optional enum **Option** to represent the value that may be present or absent. This allows Rust to handle an absence of a value safely. It will return either **Some(result)** if it is successful or **None** if there is no value present. The result is wrapped in an **Option**, which helps to handle errors in the data at compile-time, not a runtime check.

Finally **.ok_or("Missing header 'des'")?** will return an error for the function if the column (in this case, "des" once again) without crashing the entire program. **.ok_or()** converts the result of the **.position()** into an **Ok(index)** if the header is found, and an error if the header is not found. It also contains a **?** as an error propagator, which will unwrap the column name and assign it to the **des_idx** if the column is found and returns the error if not. This ensures that each of the necessary headers exist and stores their index position for use later on in the code.

```
let mut records: Vec<AsteroidData> = Vec::new();
```

let mut records = Vec::new() creates an empty vector to store the parsed rows in as **AsteroidData** structs.

```
// returns the results after the function is parsed; includes error case if there is missing data
for result: Result<StringRecord, Error> in reader.records() {
    let record: StringRecord = result?;
    records.push(AsteroidData {
        des: record.get(des_idx).ok_or(err: "Missing value for 'des'")?.to_string(),
        orbit_id: record.get(orbit_id_idx).ok_or(err: "Missing value for 'orbit_id'")?.to_string(),
        jd: record.get(jd_idx).ok_or(err: "Missing value for 'jd'")?.parse()?,
        cd: record.get(cd_idx).ok_or(err: "Missing value for 'cd'")?.to_string(),
        dist: record.get(dist_idx).ok_or(err: "Missing value for 'dist'")?.parse()?,
        dist_min: record.get(dist_min_idx).ok_or(err: "Missing value for 'dist_min'")?.parse()?,
        dist_max: record.get(dist_max_idx).ok_or(err: "Missing value for 'dist_max'")?.parse()?,
        v_rel: record.get(v_rel_idx).ok_or(err: "Missing value for 'v_rel'")?.parse()?,
        v_inf: record.get(v_inf_idx).ok_or(err: "Missing value for 'v_inf'")?.parse()?,
        t_sigma_f: record.get(t_sigma_f_idx).ok_or(err: "Missing value for 't_sigma_f'")?.to_string(),
    });
}
Ok(records)
} fn read_csv
```

This will initialize the records vector. It iterates through the rows of the CSV function and extracts each field into its assigned data style to convert it into an instance of the **AsteroidData** stuct. It also can manage instances where the data are missing.

for result in reader.records() will iterate over the rows of the CSV file with **reader.records()**, which will return each row as a **Result<StringRecord>**; then **result?** will unwrap this result. If the result succeeds as an **Ok(record)**, the data from that row will be assigned to **record**. However, if the result is an error, the function will return an error without crashing the code.

records.push(AsteroidData creates a new **AsteroidData** struct for each of the row, and then adds that row into the **records** vector. Each field within **AsteroidData** reflects a column in the dataset, and it calls upon the values from **record.**

Like the previous method, each row follows an identical format, varying only with the column name that it searches for. **des: record.get(des_idx).ok_or("Missing value for 'des'")?.to_string(),** will extract the value for that particular field. **record.get(des_idx)** retrieves the value that we just established in our previous function. This is the location of the column's index for that row. It uses an **Option** like the previous function. **.ok_or("Missing value for 'des'")** will convert this into a result or return an error if the value is missing. Then the line includes **?**, which is once again an error propagator which will unwrap the value if it is **Ok(value). .**

Some of the lines for the columns of data stored as strings include **to_string()** which will convert this into an owned string so that the AsteroidData struct can use it. It must be owned and not borrowed because it needs to be able to exist independently of other data. It copies the content of the row into string, which the **AsteroidData** struct needs it. The row is transformed into an **AsteroidData** instance, and then stored in the **records** vector.

For the lines that need numerical data, however, they have **.parse()?** called on them. This converts the string into the numerical type that was assigned to the data. For this dataset, it was previously defined as **f64**. It functions similarly to the **to_string()** function because it will either return **Ok(value)** if it is successful or an error if not. For example, if there were an invalid or missing number, the function will return an error. Then **?** will propagate these errors, if any exist.

Finally, **Ok(records)** returns the **records** vector if every row succeeds. This vector contains all of the parsed rows and is ready for further analysis in the **main.rs**.

main.rs *module; Code Explanation*

```rust
use petgraph::graph::Graph;
use std::collections::HashMap;
use std::error::Error;
mod csv_reader;
use csv_reader::read_csv;
```

I begin by importing the necessary crates for my project.

petgraph::graph::Graph will import the **Graph** data structure from the **petgraph** crate, which I use later to help model the relationships between asteroids in the dataset.

std::collections::HashMap imports the **Hashmap** storage structure, so that I can more efficiently store information about the asteroids in key-values. Each value will be associated with its corresponding key, which makes it easier to later access these.

std::error::Error helps me with handling errors gracefully, so my whole program does not crash in the event of errors.

mod csv_reader declares the module known as **csv_reader** to parse the CSV file. This allows me to access the functions and information stored in the **csv_reader** module.

use csv_reader::read_csv imports the **read_csv** function from the **csv_reader** module so I can call it in this module when I need to read my dataset.

```rust
// build a graph where nodes represent asteroids and edges indicate hazard comparisons
pub fn build_hazard_graph(data: &Vec<csv_reader::AsteroidData>, dist_threshold: f64, velocity_threshold: f64) -> Graph<(String, f64, f64), f64> {
    let mut graph: Graph<(String, f64, f64), …> = Graph::<(String, f64, f64), f64>::new();
    let mut node_map: HashMap<String, NodeIndex> = HashMap::new();

    // add nodes to the graph
    for record: &AsteroidData in data {
        if record.dist_min <= dist_threshold && record.v_rel >= velocity_threshold {
            let node: NodeIndex = graph.add_node(weight: (record.des.clone(), record.dist_min, record.v_rel));
            node_map.insert(k: record.des.clone(), v: node);
        }
    }

    // add edges to compare hazard levels between asteroids
    for node_a: NodeIndex in graph.node_indices() {
        for node_b: NodeIndex in graph.node_indices() {
            if node_a == node_b {
                continue; // do not compare to self
            }

            let asteroid_a: &(String, f64, f64) = &graph[node_a];
            let asteroid_b: &(String, f64, f64) = &graph[node_b];

            // hazard score comparison
            let hazard_a: f64 = asteroid_a.1 / asteroid_a.2; // a higher velocity & closer distance increases hazard
            let hazard_b: f64 = asteroid_b.1 / asteroid_b.2;

            // add edges weighted by difference in hazard score
            let weight: f64 = (hazard_a - hazard_b).abs();
            graph.add_edge(node_a, node_b, weight);
        }
    }
    // nodes are connected if both asteroids meet a certain "hazard threshold"
    graph
} fn build_hazard_graph
```

Next I construct my function to build a graph of the dataset. Each node of this graph will represent an asteroid, and the edges of these nodes connect the asteroids which have similar hazard comparisons. This is measured by the number of asteroids with a distance (in AU) of less than some threshold that I define. Each node will use the **des** field as its label. The weight of the edges are proportional to the absolute difference in hazard scores. This is efficient for creating a network of the relationships between asteroids.

pub fn build_hazard_graph(data: &Vec<csv_reader::AsteroidData>, dist_threshold: f64, velocity_threshold: f64) -> Graph<(String, f64, f64), f64> { begins the first function. This builds the hazard graph of the asteroids. It takes a reference to the **AsteroidData** struct from the **csv_reader** module without copying it, so it is memory efficient. Then I use **dist_threshold** and **velocity_threshold** to establish numerical parameters **f64** that I will use in relation to determine which asteroids I will include in the graph. This is then stored in a graph **-> Graph<(String, f64, f64), f64>** from the **petgraph** library to create a tuple with the asteroid's name, minimum distance, and velocity. The edges store an **f64** weight that represents the difference of hazard scores of the asteroids.

let mut graph = Graph::<(String, f64, f64), f64>::new(); establishes an empty graph where the nodes can store the tuples of the string and edge weights. I specify the data types of all of the items that will be stored in there. Both this and the HashMap must be mutable because it will be appending new nodes and edges to the graph, so we must be able to modify it after its creation.

let mut node_map = HashMap::new(); initializes an empty **HashMap** that will track which of the asteroids corresponds to each node in the graph. In this case, the keys are the name of the asteroid and the values are the indices of the node. This allows for efficient search to locate the nodes associated with each asteroid.

I add the nodes to the graph with **for record in data**, which is the beginning of a for loop that iterates over each of the records in the **AsteroidData** struct. Each of these records corresponds to the data for a single asteroid. **if record.dist_min <= dist_threshold && record.v_rel >= velocity_threshold {:** will filter the asteroids based on the thresholds that I establish by using the conditional statement **if** to

compare the current reference to the value of the threshold. **record.dist_min <= dist_threshold** ensures that the minimum distance is within the threshold. **record.v_rel >= velocity_threshold:** ensures that the asteroid's relative velocity will be equal to or greater than the threshold. If these conditions are satisfied, the asteroid will be added to the graph. Both conditions around **&&** must be true in order for this to evaluate as true. Then **let node = graph.add_node((record.des.clone(), record.dist_min, record.v_rel));** will add the asteroid as a node to the graph. **record.des.clone()** clones the asteroid's name (stored as **des**) independently of the original data to avoid memory issues, then adds the distance and velocity with **record.dist_min** and **record.v_rel.** Then **graph.add_node** will return a unique node index into the **node** variable. After this, **node_map.insert(record.des.clone(), node);** will map the name of the asteroid (once again calling **des**) to the corresponding index of the node in the **HashMap**. Inserting into a **HashMap** is very efficient and avoids any duplicate insertions. This is the most effective way to record the relationship between an asteroid and its graph nodes.

Next I must add the edges into the graph to compare the hazard levels. I do this with another **for** loop, **for node_a in graph.node_indices()**, which is the outer for loop. It contains **for node_b in graph.node_indices()**, which is the nested inner for loop. They iterate through all of the node indices in the graph, each of which represents an asteroid. The nested loop allows us to compare pairwise between the nodes. **if node_a == node_b { continue; }** makes sure that the code does not compare the node to itself to avoid repetitions that could muddle the analysis. **let asteroid_a = &graph[node_a]** and **let asteroid_b = &graph[node_b];:** retrieves the data of an asteroid that is associated with the nodes and accesses them with **graph[node_a]**. With this data accessed, we may then compare the two nodes to then conduct our hazard score.

For the hazard score, I decided to calculate the distance / velocity to provide a consistent metric ratio of how each of those values compared to one another. This weighs the distance more highly, as it will ultimately be more indicative of how dangerous an asteroid might be. A fast-moving asteroid that does not approach Earth, for example, is not as dangerous as a slower-moving asteroid that may narrowly miss us. This ensures that asteroids that are both closer to Earth and have a higher velocity will have a

greater hazard score. I compute this with **let hazard_a = asteroid_a.1 / asteroid_a.2;** and **let hazard_b = asteroid_b.1 / asteroid_b.2;**. Then I used **let weight = (hazard_a - hazard_b).abs();** to compute the absolute value of the difference in hazard scores between each asteroid, which I used as the weight of the edge. This is then added as an edge between nodes a and b with the weight, via **graph.add_edge(node_a, node_b, weight).** This is representative of how the hazard scores of the two asteroids return.

This constructs a graph that shows how each asteroid is comparatively hazardous relative to all of the other asteroids, which allows us to compare all of them and later rank how they compare to each other. After the function iterates over all of the asteroids, it will return the completed graph.

```rust
// rank asteroids based on hazard score and include details in the result
pub fn rank_hazardous_asteroids_with_details(data: &Vec<csv_reader::AsteroidData>) -> Vec<(String, f64, f64, String)> {
    let mut ranked_asteroids: Vec<(String, f64, f64, String)> = data.&Vec<AsteroidData>
        .iter() Iter<'_, AsteroidData>
        .map(|a: &AsteroidData| {
            let hazard_score: f64 = if a.dist_min > 0.0 {
                (a.v_rel / a.dist_min) / 1_000_000.0 // scale down hazard scores
            } else {
                f64::INFINITY // assign very high hazard for zero distances
            };
            (a.des.clone(), hazard_score, a.dist_min, a.cd.clone())
        }) impl Iterator<Item = (String, …)>
        .collect::<Vec<_>>();

    // sort by hazard score in descending order
    ranked_asteroids.sort_by(compare: |a: &(String, f64, f64, String), b: &(String, f64, f64, String)| {
        b.1.partial_cmp(&a.1).unwrap_or(default: std::cmp::Ordering::Equal)
    });

    ranked_asteroids
}
```

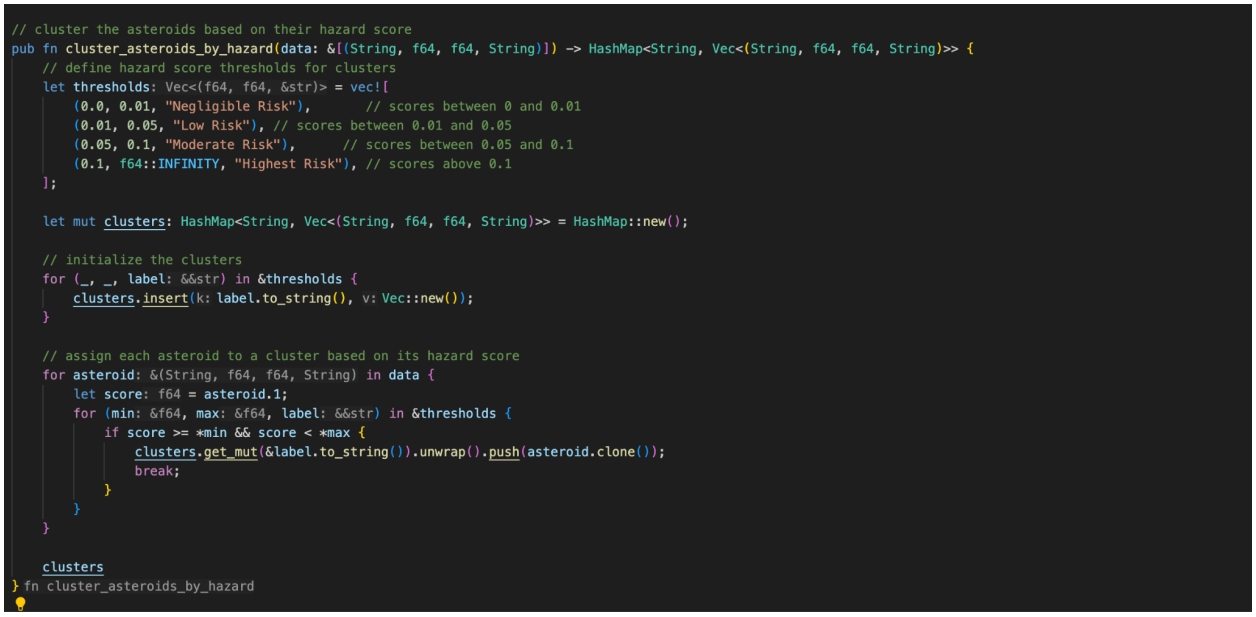
The next function is **pub fn rank_hazardous_asteroids(data: &Vec<csv_reader::AsteroidData>) -> Vec<(String, f64, f64, String)>** which defines the function that will rank the asteroids based on their hazard score. **data: &Vec<csv_reader::AsteroidData> b** takes a reference to a vector of **AsteroidData**. The **&** ensures that the function will only borrow the data, not take ownership. Then it returns a vector of tuples with **-> Vec<(String, f64, f64, String)**, which are two strings and two numerical values.

let mut ranked_asteroids = data declares the mutable variable **ranked_asteroids**, which will store the list of asteroids and their associated hazard scores. This needs to be mutable because we will need to sort it later on, which modifies the list. **data.iter()** creates an iterator over the vector, which runs

through the list of vectors one by one. Here, we can run through the list and compare the items against each other. **.map(|a| { ... })** acts as a method that applies the function to each item in the iterator. In this case, for each asteroid **a**, it checks if the asteroid's minimum distance is greater than zero to prevent errors. It does this with **a.dist_min > 0.0**, with 0 represented as 0.0 to match the **f64** floating point form. If the minimum distance is valid, it calculates the hazard score as **(a.v_rel / a.dist_min) / 1_000_000.0**, where it repeats the velocity and distance division, and then scales it by dividing it by 1,000,000. This makes the scale more readable for visualization when we print it at the end. **else f64::INFINITY** handles cases where the distance minimum is less than or equal to zero; this would create a very high hazard, as this implies that the asteroid is on a collision path. None of the asteroids in this dataset needed this, but if we were extrapolated the data, this would work.

(a.des.clone(), hazard_score, a.dist_min, a.cd.clone()) creates a tuple for the asteroid. **a.des.clone()** takes a copy of the asteroid's name as a string; **hazard_score** is the calculated hazard score; **a.dist_min** is the minimum distance of the asteroid; **a.cd.clone()** clones the approach date. All of these are collected and stored in a new vector with **.collect::<Vec<_>>()**. I use the _ to tell Rust to infer the type of elements in the vector.

ranked_asteroids.sort_by(|a, b| { sorts the **ranked_asteroids** vector by modifying its order. It uses a closure to define the logic used for sorting.
b.1.partial_cmp(&a.1).unwrap_or(std::cmp::Ordering::Equal) takes the second element of both tuples, which corresponds to the hazard score of the asteroid. The closure function uses **partial_cmp** because the hazard scores are floating point numbers. This compares the hazard scores by comparing **b.1** to **a.1**, so the list is sorted in descending order. By reversing the order of a and b, it means that greater hazard scores will appear higher on the list. **.unwrap_or(std::cmp::Ordering::Equal)** will treat the two scores as equal if the comparison fails.



```rust
// cluster the asteroids based on their hazard score
pub fn cluster_asteroids_by_hazard(data: &[(String, f64, f64, String)]) -> HashMap<String, Vec<(String, f64, f64, String)>> {
    // define hazard score thresholds for clusters
    let thresholds: Vec<(f64, f64, &str)> = vec![
        (0.0, 0.01, "Negligible Risk"),       // scores between 0 and 0.01
        (0.01, 0.05, "Low Risk"), // scores between 0.01 and 0.05
        (0.05, 0.1, "Moderate Risk"),      // scores between 0.05 and 0.1
        (0.1, f64::INFINITY, "Highest Risk"), // scores above 0.1
    ];

    let mut clusters: HashMap<String, Vec<(String, f64, f64, String)>> = HashMap::new();

    // initialize the clusters
    for (_, _, label: &&str) in &thresholds {
        clusters.insert(k: label.to_string(), v: Vec::new());
    }

    // assign each asteroid to a cluster based on its hazard score
    for asteroid: &(String, f64, f64, String) in data {
        let score: f64 = asteroid.1;
        for (min: &f64, max: &f64, label: &&str) in &thresholds {
            if score >= *min && score < *max {
                clusters.get_mut(&label.to_string()).unwrap().push(asteroid.clone());
                break;
            }
        }
    }

    clusters
} fn cluster_asteroids_by_hazard
```

This function clusters the asteroids into groups based on their hazard scores.

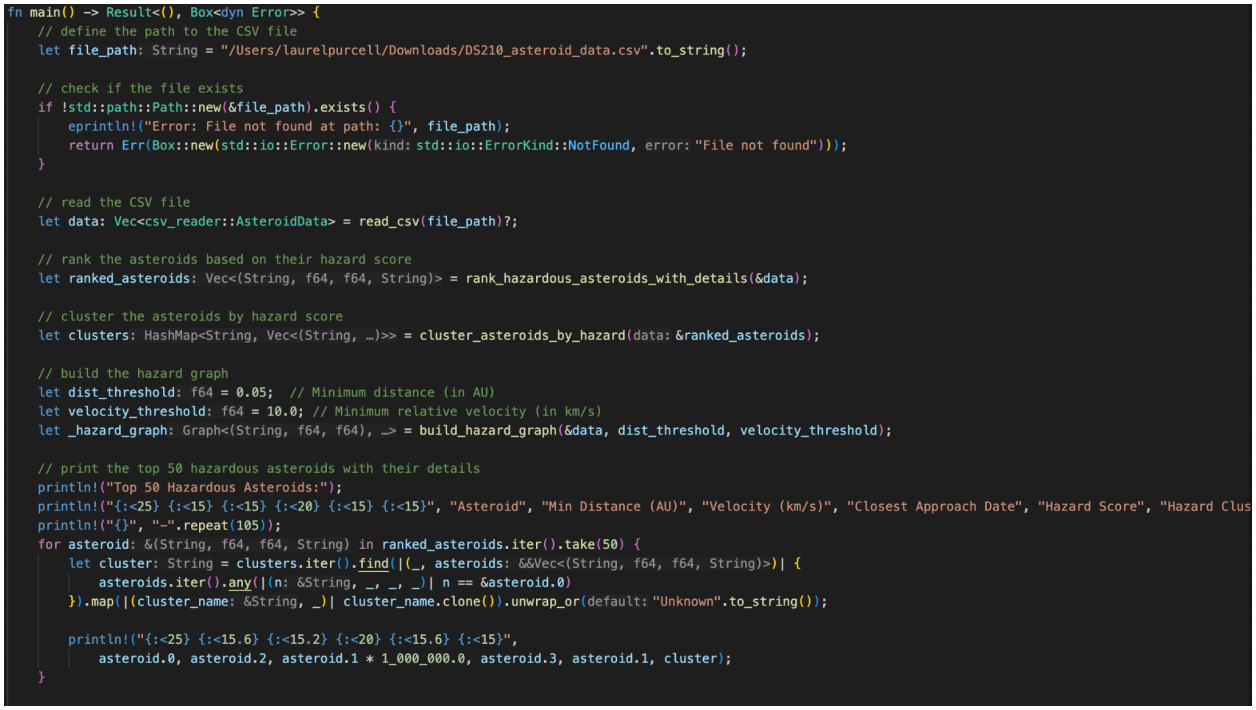
pub fn cluster_asteroids_by_hazard(data: &[(String, f64, f64, String)]) -> HashMap<String, Vec<(String, f64, f64, String)>> { takes a reference to a slice of tuples where each of the tuples contains two strings and two floating-point numbers. It stores each of these in **HashMap** where the key is the name of the risk cluster and the value is the vector of tuples containing information about the asteroid associated.

First, I define the thresholds of the hazard score. **let thresholds = vec![** creates a vector of ranges for the hazard scores and the cluster names that correspond to them. These categorize the hazard scores of the asteroid based on ranked risk levels. Each of these tuples contains the minimum score in the range, the maximum score, and the label for that range. These ranges are (0-0.01: Negligible Risk); (0.01-0.05: Low Risk); (0.05- 0.1: Moderate Risk); and (0.1-Ininity (no upper bound): Highest Risk). These are all relative to each other; the “Highest Risk” does not mean that there is imminent danger, but rather that these have the highest risk compared to the other asteroids in the dataset.

let mut clusters: HashMap<String, Vec<(String, f64, f64, String)>> = HashMap::new(); creates a HashMap to store the clusters of hazard scores. **mut** makes these clusters mutable so that we can update the entries. The HashMap stores the groups of clusters.

for (_, _, label) in &thresholds { initializes the empty clusters to be ready to store the information about the hazard. The **for** loop iterates over the thresholds vector so that it makes sure that each risk category is present. It uses _ to ignore the first two elements in the list, which are the minimum and maximum hazard scores. They are not relevant to this, as we only need the cluster label. **label.to_string()** converts the string **label** into a string, so that it can be used as the key type in the **HashMap**. Then **clusters.insert(...)** adds the entry into the clusters map. The key here is the name of the cluster of hazard and the value of the cluster is an empty vector through **Vec::new()**. This makes sure that the **HashMap** has a key for each of the clusters.

for asteroid in data creates a **for** loop to iterate over each of the asteroids in the input **data,** and each of these references a tuple that contains the details about the asteroids. **let score = asteroid.1** accesses the hazard score of the asteroid and stores it in **score** so it can be read more easily. Then with **for (min, max, label) in &thresholds** it takes the minimum and maximum of each score range, as well as the cluster label, and iterates over each of them in another for loop. **if score >= *min && score < *max {** creates an if statement that compares the asteroid's hazard score to the range of scores in the range. I use ***** to dereference the references to retrieve the actual value. Then it compares the score to the range; if the score falls within the range, the asteroid is added to the cluster through **clusters.get_mut(&label.to_string()).unwrap()**. **clusters.get_mut** fetches a mutable reference to the vector that is associated with the name of the cluster, and then converts this name to a string through **&label.to_string()**. This string will be used as the key in the HashMap, because the HashMap needs a string type as the key. Then **.unwrap()** ensures that this key will exist in the HashMap. **.push(asteroid.clone())** adds a clone of the asteroid's tuple to the vector of this cluster; it takes a clone so the ownership is not transferred and the original data can still be accessed. Then **break** will exit the inner loop after the asteroid has been assigned to its first matching cluster, so it will not be added to more than one cluster.



```rust
fn main() -> Result<(), Box<dyn Error>> {
    // define the path to the CSV file
    let file_path: String = "/Users/laurelpurcell/Downloads/DS210_asteroid_data.csv".to_string();

    // check if the file exists
    if !std::path::Path::new(&file_path).exists() {
        eprintln!("Error: File not found at path: {}", file_path);
        return Err(Box::new(std::io::Error::new(kind: std::io::ErrorKind::NotFound, error: "File not found")));
    }

    // read the CSV file
    let data: Vec<csv_reader::AsteroidData> = read_csv(file_path)?;

    // rank the asteroids based on their hazard score
    let ranked_asteroids: Vec<(String, f64, f64, String)> = rank_hazardous_asteroids_with_details(&data);

    // cluster the asteroids by hazard score
    let clusters: HashMap<String, Vec<(String, …)>> = cluster_asteroids_by_hazard(data: &ranked_asteroids);

    // build the hazard graph
    let dist_threshold: f64 = 0.05;  // Minimum distance (in AU)
    let velocity_threshold: f64 = 10.0; // Minimum relative velocity (in km/s)
    let _hazard_graph: Graph<(String, f64, f64), …> = build_hazard_graph(&data, dist_threshold, velocity_threshold);

    // print the top 50 hazardous asteroids with their details
    println!("Top 50 Hazardous Asteroids:");
    println!("{:<25} {:<15} {:<15} {:<20} {:<15} {:<15}", "Asteroid", "Min Distance (AU)", "Velocity (km/s)", "Closest Approach Date", "Hazard Score", "Hazard Clus
    println!("{}", "-".repeat(105));
    for asteroid: &(String, f64, f64, String) in ranked_asteroids.iter().take(50) {
        let cluster: String = clusters.iter().find(|(_, asteroids: &&Vec<(String, f64, f64, String)>)| {
            asteroids.iter().any(|(n: &String, _, _, _)| n == &asteroid.0)
        }).map(|(cluster_name: &String, _)| cluster_name.clone()).unwrap_or(default: "Unknown".to_string());

        println!("{:<25} {:<15.6} {:<15.2} {:<20} {:<15.6} {:<15}",
            asteroid.0, asteroid.2, asteroid.1 * 1_000_000.0, asteroid.3, asteroid.1, cluster);
    }
```

In the **main** function, I call all of the functions that I have created. This is also where I will print the results of the data analysis to display them. Here, I also created an interactive way to search manually for a specific asteroid based on its name; when the program runs, it outputs the top 50 most hazardous asteroids and also prompts the viewer to search for a specific asteroid.

let file_path = "/Users/laurelpurcell/Downloads/DS210_asteroid_data.csv".to_string(); creates the variable **file_path** to store the location of the CSV file, which has a path to the local CSV file.

if !std::path::Path::new(&file_path).exists() calls a module to handle errors in the case that the CSV file does not exist. This handles them gracefully so the entire program would not crash.

let data: Vec<csv_reader::AsteroidData> = read_csv(file_path)?; reads the CSV file with the function **read_csv** established in the **csv_reader** module. It parses the dataset into a vector of **AsteroidData**.

let ranked_asteroids = rank_hazardous_asteroids(&data); calls the **rank_hazardous_asteroids** function to rank the hazard score. It outputs a sorted vector of tuples with the relevant information.

let clusters = cluster_asteroids_by_hazard(&ranked_asteroids); calls **cluster_asteroids_by_hazard** on the **ranked_asteroids** variable. It outputs a **HashMap** where the keys are the labels of the hazard clusters and the values are the vectors of asteroid data.

let _hazard_graph = build_hazard_graph(&data, dist_threshold, velocity_threshold); calls the **build_hazard_graph** function to construct a graph of the nodes representing the asteroids and the edges as the hazard comparisons. For the two thresholds, I filter the asteroids with a minimum distance of 0.05 AU and velocity of 10km/s.

println!("Top 50 Hazardous Asteroids:"); prints the table header for the asteroid results. **println!("{:<25} {:<15} {:<15} {:<20} {:<15} {:<15}", "Asteroid", "Min Distance (AU)", "Velocity (km/s)", "Closest Approach Date", "Hazard Score", "Hazard Cluster");** control the alignment of the columns to properly format the results. Then **for asteroid in ranked_asteroids.iter().take(50) {** iterates over the top 50 asteroids, takes the first 50, and searches within the cluster **HashMap** to find which of the clusters the asteroid falls into. If the cluster name is found, it will be extracted. Then it prints the details about these asteroids **println!("{:<25} {:<15.6} {:<15.2} {:<20} {:<15.6} {:<15}"** with all of the relevant information.

```
    // establish an interactive lookup to manually search for asteroids by name
    use std::io::{self, Write};

    println!("\nEnter the name of an asteroid to retrieve details or type 'exit' to quit:");
    loop {
        print!("Asteroid name: ");
        io::stdout().flush()?;

        let mut input: String = String::new();
        io::stdin().read_line(buf: &mut input)?;
        let input: &str = input.trim();

        if input.eq_ignore_ascii_case("exit") {
            break;
        }

        if let Some(asteroid: &(String, f64, f64, String)) = ranked_asteroids.iter().find(|(name: &String, _, _, _)| name == input) {
            let cluster: String = clusters.iter().find(|(_, asteroids: &&Vec<(String, f64, f64, String)>)| {
                asteroids.iter().any(|(n: &String, _, _, _)| n == &asteroid.0)
            }).map(|(cluster_name: &String, _)| cluster_name.clone()).unwrap_or(default: "Unknown".to_string());

            println!("\nDetails for asteroid '{}':", asteroid.0);
            println!("{:<15}: {:.6} AU", "Min Distance", asteroid.2);
            println!("{:<15}: {:.2} km/s", "Velocity", asteroid.1 * 1_000_000.0);
            println!("{:<15}: {}", "Date", asteroid.3);
            println!("{:<15}: {}", "Hazard Cluster", cluster);
        } else {
            println!("Asteroid '{}' not found in the dataset.", input);
        }
    }

    Ok(())
} fn main
```

This is a continuation of my **main** function. I create an interactive lookup to be able to search for a specific asteroid by name, which will then return the information about it.

println!("\nEnter the name of an asteroid to retrieve details or type 'exit' to quit:"); will prompt the user to enter the name of an asteroid. **loop** creates a loop to handle multiple queries. **print!("Asteroid name: ")** and **io::stdout().flush()?;** deal with the user input. Then **let mut input = String::new();** will create a new mutable string to store the input. **io::stdin().read_line(&mut input)?;** retrieves the information about the asteroid whose name matches the asteroid input. **let input = input.trim();** trims the whitespace from the input. Then the **if** statement **if input.eq_ignore_ascii_case("exit")** will trigger a break in the loop if the user types “exit” ignoring the ascii case. **if let Some(asteroid) = ranked_asteroids.iter().find(|(name, _, _, _)| name == input)** iterates through the asteroids in **ranked_asteroids** to find if there is a matching name.

If there is a matching asteroid, it displays the details through a series of print statements, each corresponding to a different part of the tuples containing the information. Otherwise, it will handle the error with an **else** statement to print **println!("Asteroid '{}' not found in the dataset.", input)** so the program handles it gracefully.

This provides a way to interact with the dataset and manipulate the output according to the needs of the user; for example, if they are searching for a relevant asteroid that is not in the top 50 of the list.

Tests

I created two test cases, stored in their own **tests.rs** module for better organization.

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use csv_reader::AsteroidData;

    fn mock_asteroid_data() -> Vec<AsteroidData> {
        vec![
            AsteroidData {
                des: "2023 AB".to_string(),
                dist_min: 0.03,
                v_rel: 15.0,
                cd: "2023-01-01".to_string(),
            },
            AsteroidData {
                des: "2023 XY".to_string(),
                dist_min: 0.1,
                v_rel: 8.0,
                cd: "2023-05-05".to_string(),
            },
            AsteroidData {
                des: "2023 ZZ".to_string(),
                dist_min: 0.02,
                v_rel: 20.0,
                cd: "2023-07-07".to_string(),
            },
        ]
    }

    #[test]
    fn test_main_ranking_and_clustering() {
        let data = mock_asteroid_data();

        // test ranking
        let ranked_asteroids = rank_hazardous_asteroids(&data);
        assert_eq!(ranked_asteroids.len(), 3);
        assert!(ranked_asteroids[0].1 > ranked_asteroids[1].1, "Asteroids should be ranked by hazard score descending.");

        // test clustering
        let clusters = cluster_asteroids_by_hazard(&ranked_asteroids);
        assert!(clusters.contains_key("Highest Risk"));
        assert!(clusters.contains_key("Moderate Risk"));
        assert!(clusters["Highest Risk"].len() > 0, "At least one asteroid should be in the highest risk cluster.");
    }

    #[test]
    fn test_hazard_graph_building() {
        let data = mock_asteroid_data();
        let dist_threshold = 0.05;
        let velocity_threshold = 10.0;

        let hazard_graph = build_hazard_graph(&data, dist_threshold, velocity_threshold);

        // verify nodes
        assert_eq!(hazard_graph.node_count(), 2, "Graph should include only asteroids meeting the thresholds.");

        // verify edges
        assert!(hazard_graph.edge_count() > 0, "There should be at least one edge in the hazard graph.");
    }
}
```

The first one creates a set of mock asteroid data, simulating the struct **AsteroidData**. It tests the ranking function to ensure the hazard scores are calculated and sorted correctly. It also tests the clustering, which checks to see that the asteroids are assigned to the correct cluster based on their hazard score. Then it tests the hazard graph, to make sure that the graph built will contain the correct nodes and edges based on the given thresholds.

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use csv_reader::AsteroidData;

    // helper function to generate mock asteroid data
    fn mock_asteroid_data() -> Vec<AsteroidData> {
        vec![
            AsteroidData {
                des: "Asteroid A".to_string(),
                dist_min: 0.02,
                v_rel: 20.0,
                cd: "2024-01-01".to_string(),
            },
            AsteroidData {
                des: "Asteroid B".to_string(),
                dist_min: 0.05,
                v_rel: 10.0,
                cd: "2024-02-01".to_string(),
            },
            AsteroidData {
                des: "Asteroid C".to_string(),
                dist_min: 0.1,
                v_rel: 5.0,
                cd: "2024-03-01".to_string(),
            },
        ]
    }

    #[test]
    fn test_rank_hazardous_asteroids() {
        let data = mock_asteroid_data();

        // call the function
        let ranked = rank_hazardous_asteroids(&data);

        // verify the results
        assert_eq!(ranked.len(), 3, "There should be three asteroids ranked.");
        assert_eq!(ranked[0].0, "Asteroid A", "Asteroid A should be ranked first due to highest hazard score.");
        assert!(ranked[0].1 > ranked[1].1, "Hazard score of the first asteroid should be greater than the second.");
        assert!(ranked[1].1 > ranked[2].1, "Hazard score of the second asteroid should be greater than the third.");

        // verify the hazard score computation
        let asteroid_a_score = (data[0].v_rel / data[0].dist_min) / 1_000_000.0;
        assert!((ranked[0].1 - asteroid_a_score).abs() < 1e-6, "Hazard score of Asteroid A should match expected value.");
    }
```

```rust
#[test]
fn test_cluster_asteroids_by_hazard() {
    let data = mock_asteroid_data();
    let ranked = rank_hazardous_asteroids(&data);

    // call the function
    let clusters = cluster_asteroids_by_hazard(&ranked);

    // verify the results
    assert_eq!(clusters.len(), 4, "There should be four clusters.");

    // check the clustering of each asteroid
    for asteroid in ranked {
        let hazard_score = asteroid.1;
        let assigned_cluster = clusters.iter().find(|(_, asteroids)| {
            asteroids.iter().any(|(name, _, _, _)| name == &asteroid.0)
        });
        assert!(assigned_cluster.is_some(), "Each asteroid should belong to a cluster.");

        // verify correct cluster assignment based on hazard score
        if hazard_score < 0.01 {
            assert!(assigned_cluster.unwrap().0 == "Negligible Risk", "Asteroid with low hazard score should be in 'Negligible Risk'.");
        } else if hazard_score < 0.05 {
            assert!(assigned_cluster.unwrap().0 == "Low Risk", "Asteroid with moderate hazard score should be in 'Low Risk'.");
        } else if hazard_score < 0.1 {
            assert!(assigned_cluster.unwrap().0 == "Moderate Risk", "Asteroid with higher hazard score should be in 'Moderate Risk'.");
        } else {
            assert!(assigned_cluster.unwrap().0 == "Highest Risk", "Asteroid with the highest hazard score should be in 'Highest Risk'.");
        }
    }
}
```

The second set of test cases focuses on the **rank_hazardous_asteroids** and **cluster_asteroids_by_hazard** functions specifically. **test_rank_hazardous_asteroids** confirms that the asteroids are ranked correctly based on their hazard scores, confirms that the hazard scores are computed and are scaled correctly, and that the ranking matches what is expected. **test_cluster_asteroids_by_hazard** confirms that all of the necessary cluster labels are present and that the asteroids are assigned to the correct range. It also ensures that each asteroid belongs to a cluster once the function is run.

Dependencies

The dependencies lists the external dependencies and crates that I implemented into my project. **csv = "1.1"** is the library that I used when working with the asteroid dataset, which was in CSV form. It allows the **ReaderBuilder** to function and open the file, read the rows, and extract information about the headers.

petgraph = "0.6" helps me to create my graph data structure. It provides the option for directed and undirected graphs, as well as weighted graphs. This is useful when I am constructing the graph of information about the asteroids. This helps me to model the relationships between items in my graph.

serde = { version = "1.0", features = ["derive"] } helps me to deal with the data structures in the CSV files. It deserializes the CSV file, which helps to convert it into a Rust type so that I can perform analysis in Rust.

Output & Results

My code outputs a result of the top 50 closest asteroids, including their name, distance from Earth, velocity, the date that they will pass, hazard score, and the hazard cluster it falls into. This includes one notable asteroid that passed by about 8,000 km from Earth's center, 2024 LH1, which is the second-closest observed flyby of an asteroid

(https://neo.ssa.esa.int/-/2024-lh1-the-second-closest-close-approach). 0.000054AU is about 8080 kilometers; this hazard level is reflected in the output of my data.

Top 50 Hazardous Asteroids:

Asteroid	Min Distance (AU)	Velocity (km/s)	Closest Approach Date	Hazard Score	Hazard Cluster
2024 LH1	0.000054	321637.51	2024-Jun-06 14:02	0.321638	Highest Risk
2023 RS	0.000069	196494.21	2023-Sep-07 14:26	0.196494	Highest Risk
2024 HA	0.000104	187234.17	2024-Apr-16 17:42	0.187234	Highest Risk
2020 FD	0.000091	165929.45	2024-Mar-18 19:11	0.165929	Highest Risk
2023 AV	0.000103	146162.50	2023-Jan-12 20:09	0.146163	Highest Risk
2023 BU	0.000067	139096.04	2023-Jan-27 00:29	0.139096	Highest Risk
2023 UR10	0.000093	131440.04	2023-Oct-20 04:24	0.131440	Highest Risk
2024 GJ2	0.000125	115197.78	2024-Apr-11 18:30	0.115198	Highest Risk
2023 WA	0.000183	98106.79	2023-Nov-17 03:41	0.098107	Moderate Risk
2023 VB2	0.000219	82321.83	2023-Nov-07 07:00	0.082322	Moderate Risk
2017 SA20	0.000087	71509.68	2024-Apr-19 09:48	0.071510	Moderate Risk
2024 RC42	0.000145	66322.29	2024-Sep-12 19:32	0.066322	Moderate Risk
2024 JT3	0.000172	63238.40	2024-May-09 09:36	0.063238	Moderate Risk
2023 VA	0.000179	63204.28	2023-Nov-02 05:37	0.063204	Moderate Risk
2024 EL3	0.000241	56912.38	2024-Mar-11 10:31	0.056912	Moderate Risk
2024 JN16	0.000167	55498.10	2024-May-14 09:49	0.055498	Moderate Risk
2024 EJ4	0.000175	49522.69	2024-Mar-13 17:22	0.049523	Low Risk
2023 TO17	0.000363	41779.85	2023-Oct-14 14:35	0.041780	Low Risk
2023 SL5	0.000345	40213.83	2023-Sep-20 07:53	0.040214	Low Risk
2023 UB	0.000392	39704.65	2023-Oct-15 19:08	0.039705	Low Risk

2024 FQ5	0.000218	39412.09	2024-Mar-31 15:32	0.039412	Negligible Risk
2024 HO2	0.000249	37031.93	2024-Apr-29 15:18	0.037032	Low Risk
2024 CH4	0.000726	33083.66	2024-Feb-11 06:10	0.033084	Low Risk
2023 MJ20	0.000297	32158.77	2023-Jun-18 20:59	0.032159	Low Risk
2023 VO1	0.000522	31366.75	2023-Nov-04 20:19	0.031367	Low Risk
2023 JO	0.000615	30602.18	2023-May-08 20:14	0.030602	Low Risk
2023 XJ1	0.000656	30557.70	2023-Dec-06 18:57	0.030558	Low Risk
2023 QY	0.000443	29824.94	2023-Aug-18 15:49	0.029825	Low Risk
2024 RL3	0.000273	29622.24	2024-Sep-04 17:15	0.029622	Low Risk
2018 NW	0.000737	29617.31	2023-Jul-10 05:48	0.029617	Low Risk
2023 VE1	0.000230	28019.06	2023-Nov-04 00:36	0.028019	Low Risk
2023 TR1	0.000720	27754.44	2023-Oct-07 05:25	0.027754	Low Risk
2023 DR	0.000507	27541.76	2023-Feb-25 05:47	0.027542	Low Risk
2023 TV3	0.000556	25557.50	2023-Oct-12 12:52	0.025558	Low Risk
2012 KP24	0.000491	25185.78	2023-May-31 16:26	0.025186	Low Risk
2023 SH7	0.000359	23983.15	2023-Sep-25 16:50	0.023983	Low Risk
2023 GQ	0.000894	23013.54	2023-Apr-11 11:35	0.023014	Low Risk
2024 EF	0.000385	22442.01	2024-Mar-04 07:00	0.022442	Low Risk
2023 XJ	0.000705	22309.57	2023-Dec-02 19:20	0.022310	Low Risk
2023 QS1	0.000715	21973.13	2023-Aug-19 18:22	0.021973	Low Risk
2024 MW	0.000620	21918.06	2024-Jun-28 07:11	0.021918	Low Risk
2021 JG9	0.000736	21908.71	2024-May-10 11:13	0.021909	Low Risk
2023 AN6	0.000647	21713.15	2023-Jan-15 12:51	0.021713	Low Risk
2023 VD2	0.000375	21612.71	2023-Nov-07 12:53	0.021613	Low Risk
2024 GX3	0.000478	21283.97	2024-Apr-10 08:27	0.021284	Low Risk
2023 TZ65	0.001200	17986.61	2023-Oct-10 02:27	0.017987	Low Risk

2024 CY1	0.000811	17909.99	2024-Feb-12 07:32	0.017910	Low Risk
2023 NT1	0.000674	16717.47	2023-Jul-13 10:13	0.016717	Low Risk
2019 DA1	0.000815	16425.71	2024-Mar-03 20:07	0.016426	Low Risk
2023 VD4	0.000964	16144.69	2023-Nov-10 16:24	0.016145	Low Risk

Enter the name of an asteroid to retrieve details or type 'exit' to quit:

Asteroid name: