# Rust Lab Document

April 25, 2025

## 1 Introduction

This document presents the design and implementation of the Hacker Spreadsheet—a text-based, Vim-inspired spreadsheet editor for terminal users. Aimed at keyboard-driven workflows and remote usage, it prioritizes speed, privacy, and minimalism.

We outline the architecture, key design choices, challenges faced, and reasons why some proposed features couldn't be completed. The document also explores possible future extensions and provides instructions for running the current version.

## 2 Implementation Challenges

### 2.1 Undo/Redo Operations

While the codebase includes implementation for undo and redo operations, it was originally designed for single-cell operations. The function push_undo() was set up to handle individual cell changes, but proved insufficient when handling batch operations.

Later, a push_undo_sheet() function was implemented to handle multi-cell operations. Due to time constraints, the team couldn't properly integrate this function for all multi-cell operations, leading to inconsistent behavior when using undo/redo with features like sorting or multi-cell insertions.

### 2.2 Filter Functionality

The filter functionality was partially implemented but couldn't be completed. The main challenges were:

- Displaying only filtered rows while maintaining the overall structure.
- Creating an efficient mechanism to exit filter mode and restore the full view.

The team was able to implement the actual filtering logic but couldn't complete the user interface components necessary for a seamless experience within the time constraints.

## 3 Extra Extensions Implemented

### 3.1 Horror Theme

An interesting extension implemented was a "haunted" mode that adds a horror theme to the spreadsheet. This feature is activated through a **:haunt** command in the spreadsheet in Normal Mode.

The horror theme plays spooky sounds and displays unsettling messages and flickering during certain actions.

## 4 Primary Data Structures

### 4.1 Cell Structure

Each cell in the spreadsheet stores user input, formatting, and evaluation results. It includes:

- **Raw value**: The text or formula entered by the user
- **Display value**: The evaluated result shown on screen.
- **Formula**: Stores a formula reference if present, otherwise empty.
- **Lock status**: Indicates if the cell is editable.
- **Alignment**: Controls text alignment (left, center, right).
- **Width and height**: Define the cell's display size.

### 4.2 Spreadsheet Structure

The Spreadsheet struct manages the overall state and behavior of the application. It maintains:

- A mapping of cell addresses to Cell objects, representing the spreadsheet grid.
- The current cursor position and mode (e.g., normal, insert, command).
- Dimensions of the spreadsheet (max rows and columns).
- A command buffer and status message for user interaction.
- Undo/redo stacks to support reversible actions.
- Find-related data, including query matches and navigation.
- Dependency graphs to track formula relationships and enable reactive updates.
- Update tracking to avoid circular or redundant evaluations.
- Fields for haunt mode, enabling streaming interactions via audio or external feedback.

### 4.3 Cell Addressing

The CellAddress struct handles the mapping between standard spreadsheet notation (like "A1") and the internal row-column index representation. It stores the column and row as numeric indices, enabling efficient lookup, navigation, and formula evaluation. This abstraction simplifies the conversion between user-facing cell references and internal data structures.

### 4.4 Dependency Tracking

To support automatic formula updates, the spreadsheet maintains two hash maps:

- Dependencies: Tracks which cells a given cell depends on (i.e., inputs to its formula).
- Dependents: Tracks which cells depend on a given cell (i.e., outputs affected by changes).

# 5 Interfaces Between Software Modules

## 5.1 Formula Processing and Cell Updates

The interface between cell data and formula processing is handled through the update_cell() method. This function serves as a critical interface, handling:

- Cell value validation
- Formula parsing and evaluation
- Dependency management
- Circular dependency detection
- Undo state management

## 5.2 Dependency Management Interface

Methods like `add_dependency()`, `remove_dependencies()`, and `propagate_changes()` form an interface for managing relationships between cells. This interface ensures that changes propagate correctly through the dependency graph. To maintain consistency and avoid circular updates, the system uses a topological sort algorithm implemented with a **stack-based** approach, ensuring that dependent cells are updated in the correct order.

## 5.3 File I/O Interface

The application supports saving and loading spreadsheet data using JSON. The `save_json` method serializes the internal data structure and writes it to a file in a readable JSON format. The `load_json` method reads from a JSON file and reconstructs the data, allowing users to persist and restore their work across sessions.

# 6 Design Evaluation

## 6.1 Strengths of Design

**Effective Use of Data Structures**

The implementation makes excellent use of HashMaps and HashSets:

1. HashMap<String, Cell>provides O(1) access to cells by their address
2. HashSet<String>efficiently stores and checks cell dependencies
3. VecDeque<UndoAction>effectively manages the undo/redo history with bounds

**Modular Function Design**

Functions are well-scoped with clear responsibilities:

– update_cell() handles all cell modification logic

– propagate_changes() manages dependency propagation

– parse_range() encapsulates range parsing

**Strong Encapsulation**

The design effectively encapsulates:

- Cell addressing complexity
- Formula evaluation
- Dependency tracking
- Display formatting

**Practical Features**

The implementation includes practical spreadsheet features needed in daily use:

- Formula evaluation with SUM, MIN, MAX, STDEV
- Cell locking for data protection
- Text alignment options
- Configurable cell dimensions
- Finding and navigation
- Sorting capabilities

## 7 Guide to Our Extension

This section outlines the custom extension we developed for the spreadsheet program, introducing interactive terminal-based functionality with several advanced features and commands.

### 1. Overview

Upon invoking the `make ext1` command, a spreadsheet of size $100 \times 100$ is initialized within the terminal. The viewport is fixed to a $10 \times 10$ window, through which users can navigate and interact with the sheet.

### 2. Navigation

Users can control the cursor using either of the following key bindings:

**h, j, k, l**: Move left, down, up, and right respectively.

**w, a, s, d**: Shift the viewport in the respective direction.

`:j <cell>`: Jump directly to a specific cell (e.g., `:j A1`).

`:hh`, `:ll`: Move to the extreme left or right of the current row.

`:kk`, `:jj`: Move to the top or bottom of the current column.

## 3. Insert Mode

Pressing `:i` or `:i <cell>` enables insert mode. If no cell is specified, the currently selected cell is modified. An "Inserting..." prompt appears at the bottom right. Users can then:

Enter values or strings (e.g., `2`, `abc`).

Use formulas (e.g., `=SUM(A1:B1)`, `=sqrt(A1)`, etc.).

Perform arithmetic operations (e.g., `=(A1+1)`).

Refer to other cells (e.g., `=(A1)`).

Exit insert mode using the `Esc` key.

## 4. File Operations

`:load <path>`: Loads a JSON file into the spreadsheet.

`:saveas_json <path>`: Saves the current sheet as a JSON file.

`:saveas_pdf <path>`: Saves the sheet as a PDF document.

## 5. Batch Insertion and Search

`:mi [Range] <value>`: Inserts a value or formula across a specified range (e.g., `:mi [A1:B1] 2`, `:mi [A1:B1] =SUM(C1:D1)` ).

`:find <value>`: Highlights matches and displays the count. Use `n` and `p` to cycle through matches.

## 6. Sorting and Formatting

`:sort [Range] <flag>`: Sorts the given row range in ascending (1) or descending (0) order.

`:align <cell> (l/r/c)`: Aligns content in a specified cell or the current cell (left, right, center).

Cell sizes are fixed. Overflowing content is truncated with ellipsis (e.g., "alphabet" becomes "alp..").

## 7. Cell Dimensions and Locking

`:dim <cell> (h,w)`: Modifies the row and column dimensions of the specified or current cell.

`:lock <cell>`, `:unlock <cell>`: Locks or unlocks the specified or current cell.

### 8. Thematic Modes

As part of a theme-based extension system, we propose the integration of dynamic modes that enhance the user experience with visual and auditory effects. Currently, the first implemented theme is **Haunt Mode**, a playful and immersive feature designed for entertainment.

> `:haunt`: Activates haunt mode. For the best experience, users are advised to wear headphones at full volume.
>
> Once activated, the sheet begins to flicker and display eerie or spooky messages at random intervals.
>
> Typing in this mode triggers visual glitches, and occasional jump scares designed to surprise the user.
>
> `:dehaunt`: Deactivates haunt mode and restores the interface to its normal state.

This mode demonstrates the flexibility of our extension framework, allowing for creative, theme-based user experiences that go beyond traditional spreadsheet functionality.

## 8 Conclusion

This document outlined the development of a feature-rich, terminal-based spreadsheet application. With support for advanced navigation, editing, formatting, and batch operations, the extension enhances both functionality and user experience.

The introduction of theme-based modes, exemplified by Haunt Mode, demonstrates the system's flexibility and potential for creative expansion. Overall, this project combines practicality with interactivity, offering a solid foundation for future enhancements.

## 9 Links

Github Repo link: GitHub Repository