HEP Agent Simulation Framework

Code Documentation

January 20, 2026

Contents

Chapter 1

Introduction

1.1 Overview

The HEP Agent Simulation Framework (`ABM-Hescor`) is a high-performance simulation environment designed to model agent dynamics over large-scale geospatial grids (Human Evolution Potential - HEP data). The core simulation logic is written in modern **Fortran 2003/2008** for maximum performance, while the user interface and high-level control are implemented in **Python** (via PyQt5).

1.1.1 Scientific Goal

The present program is intended to simulate various aspects of the lives of prehistoric humans—such as Neanderthals and other human species. These aspects include movement, life cycle (reproduction and death), cultural development, and genetic inheritance. The goal is to compare the results of the simulation with archaeological data and then use the simulation data to fill gaps in the fossil record. In the past, a predecessor of this simulation already demonstrated, using climate data, how the settlement of Europe by modern humans might have taken place.

1.1.2 Agent-Based Modeling

The simulation is an agent-based model (ABM). This means that individual acting entities within the simulation are treated separately, and the agents' decisions are not aggregated but made individually for each agent. For example, rather than solving a partial differential equation (PDE) for population density, the model simulates millions of individual agents moving, reproducing, and interacting.

1.2 Architecture

The system follows a hybrid architecture (Fortran + Python):

- **Fortran Backend**: Handles all heavy lifting including:
  - Memory management for millions of agents (using `mod_agent_world`).
  - Spatial indexing (Grid) and Agent-ID mapping (Hashmap).
  - Simulation modules (Movement, Births, Deaths, Resources).
  - High-speed I/O for HEP NetCDF files.
- **Python Frontend**: Provides a user-friendly interface for:

- configuring simulations.
- Real-time visualization of agents and data on a 3D/2D globe.
- Controlling the simulation loop (Start, Stop, Step).
- Debugging and verification.

Interaction between the two layers is handled via `f2py` generated wrappers, exposing Fortran subroutines directly to Python as a complied extension module (`mod_python_interface`).

1.3 Installation

The program requires Python, Fortran (gfortran), and `pip`.

1.3.1 Requirements

```
sudo apt install gfortran
sudo apt install python3 python3-venv python3-pip
```

Python Packages

Visualisation requires:

```
pip install pandas matplotlib pillow cartopy numpy imageio
```

1.3.2 Building the Project

To compile the Fortran extension and link it to Python:

1. Navigate to the project root.
2. Run the build script:
   ```
   ./build_extension.sh
   ```
3. Verify that `mod_python_interface*.so` exists in the root directory.

1.4 Running the Simulation

1.4.1 Python UI (Recommended)

The preferred way to run the simulation, especially for development and debugging, is via the Python user interface:

```
python3 python/application.py
```

This opens a GUI that allows you to:

- Load config files and HEP maps.
- Start/Stop/Restart the simulation.
- Visualize agent positions in real-time.
- Select and verify active simulation modules.

1.4.2 Legacy/Batch Mode

For batch processing or headless server runs, you can invoke the compiled binary directly (if built via `make`) or use python scripts configured for headless execution.

Chapter 2

Data Structures & Storage

2.1 Data Storage Logic

Efficiently storing and retrieving agents is critical for performance. The framework uses a **Structure-of-Arrays (SoA)** approach mixed with object-oriented containers to balance cache efficiency and code maintainability.

2.1.1 Design Choice: Hybrid Storage

The simulation needs to handle millions of agents. Storing each agent as a separate object scattered in memory (as in a pure linked list) would cause excessive cache misses. Conversely, a pure structure-of-arrays can make code harder to read. We compromise by using a `world_container` that holds global arrays, but we expose individual agents via pointers to derived `Agent` types that "view" the array data.

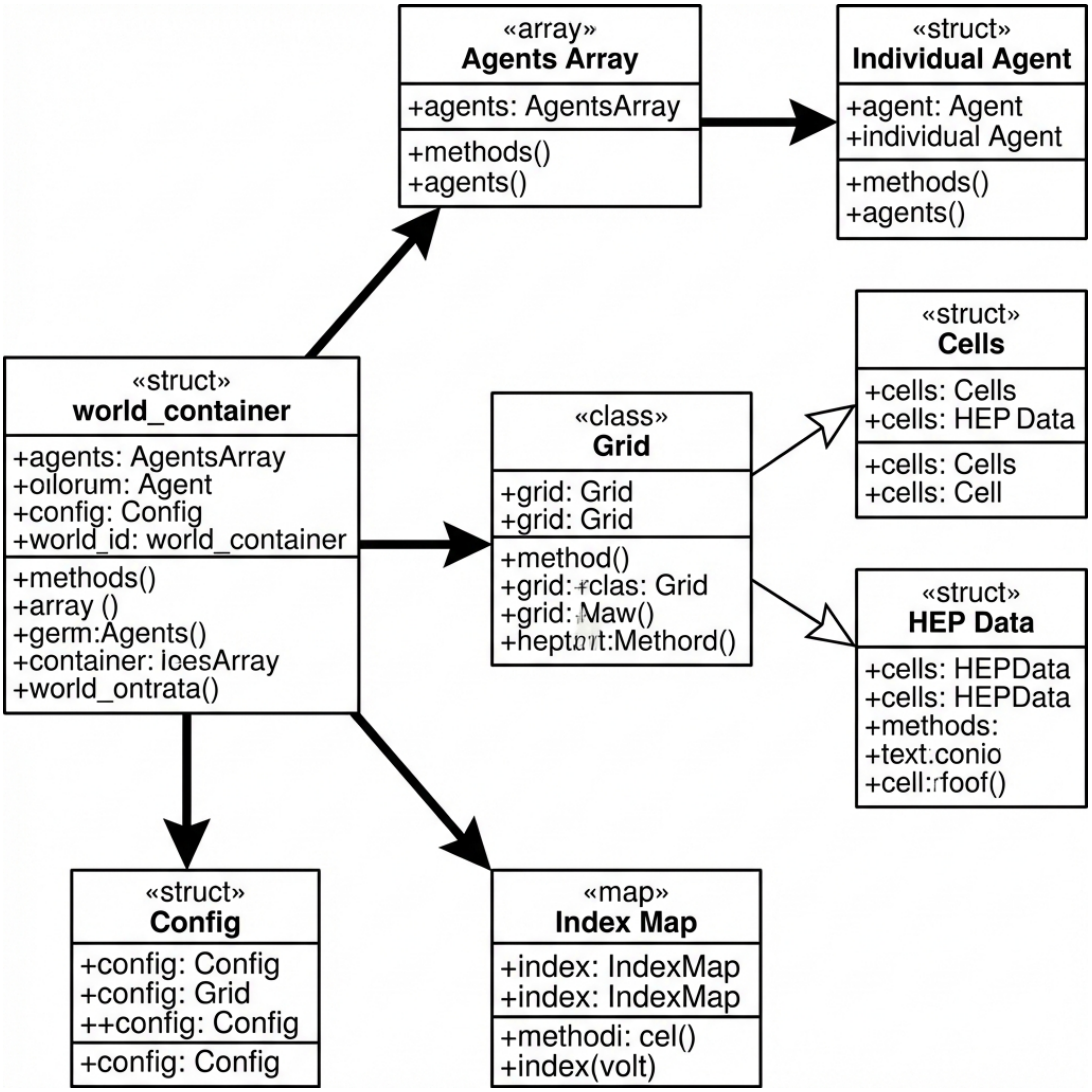

Figure 2.1: Visual representation of the hybrid data structure. The `world_container` acts as the root, managing contiguous memory arrays for agents while maintaining spatial (Grid) and logical (Hashmap) indices for efficient access.

2.1.2 The Agent Array

Agents are stored in a large, pre-allocated 2D array within the `world_container`:

```
type(Agent), allocatable :: agents(:,:) ! (max_size, npops)
```

- **Rows**: Iterate through individual agents (up to `max_size`).
- **Columns**: Separate different populations (e.g., `npops`).

This allows for contiguous memory access when iterating over a single population, which is the most common operation.

2.1.3 Spatial Indexing: The Grid

For spatial interactions (e.g., mating, conflict), agents are indexed by their geospatial location using the `Grid` structure.

- Each grid cell (`grid_cell`) contains a dynamic list of Agent IDs currently located in that cell.
- When an agent moves, it is removed from the old cell's list and added to the new cell's list.

Module Types: Agent-Centric vs Cell-Centric

When developing modules, you must decide the primary access pattern:

- **Agent-Centric Modules**: Functions that apply to a specific agent (e.g., Movement). These iterate over the `agents` array.

```
    subroutine agent_move(agent_ptr)
        type(Agent), pointer :: agent_ptr

```

- **Cell-Centric Modules**: Functions that apply to a location (e.g., Overpopulation Death). These iterate over the `grid`.

```
    subroutine death_overpopulation(cell_ptr)
        type(cell), pointer :: cell_ptr

```

2.2 Module `mod_agent_world`

This module defines the central `world_container` which holds:

- `agents`: The main agent array.
- `grid`: The spatial grid.
- `index_map`: The ID lookup hashmap.
- `config`: Global configuration parameters.

It serves as the "Hub" for all data access.

2.2.1 Agent Lookup: Hashmap

To find an agent by its unique ID (without searching the entire array), the system uses a custom integer hashmap (`mod_hashmap`).

- **Key**: Agent ID (Integer).
- **Value**: A composed integer packing both `population` index and `array_index`.

This allows $O(1)$ access to any agent pointer given its ID.

2.3 Compaction

When agents die, they are marked with `is_dead = .true.`. They are NOT removed immediately to avoid array resizing costs. Instead, a `compact_agents` subroutine is called periodically (e.g., end of timestep).

1. It iterates from both ends of the array.
2. Dead agents on the left are overwritten by living agents from the right.
3. The array size is logically reduced (tracking integer), but physically remains allocated.

Chapter 3

Developing New Modules

3.1 Concept

A "Module" in this framework is a self-contained behavior that can be applied to agents (e.g., Movement, Aging, Mating). Modules are written in Fortran and exposed to the scheduling logic via a standardized interface.

3.2 Step-by-Step Guide

3.2.1 1. Implement the Logic (Fortran)

Create a new subroutine in `src/simulation_modules/mod_modules.f95` (or a new file). the subroutine must accept an `Agent` pointer.

```
subroutine my_new_behavior(self)
    use mod_agent_world
    implicit none
    class(Agent), intent(inout) :: self

    ! Implement logic here
    self%pos_x = self%pos_x + 0.1

end subroutine my_new_behavior
```

3.2.2 2. Register the Module ID (Python Interface)

In `src/interfaces/python_interface.f95`, define a new integer constant for your module ID.

```
integer, parameter :: MODULE_MY_BEHAVIOR = 9
```

Then, add a case to the `step_simulation` dispatch loop:

```
select case (active_module_ids(jp))
    ! ... existing cases
    case (MODULE_MY_BEHAVIOR)
        call apply_module_to_agents(my_new_behavior, t)
end select
```

3.2.3 3. Expose to Python (UI)

In `python/spawn_editor.py`, add your new module to the `available_modules` dictionary making sure the ID matches the Fortran constant.

```
self.available_modules = {
    # ...
    "My New Behavior": 9
}
```

3.2.4 4. Compilation

Run the build script to recompile the extension:

```
./build_extension.sh
```

3.3 Best Practices

- **Statelessness**: Try to keep modules stateless where possible. Use agent variables to store state.
- **Grid Access**: Use `self%grid` to access neighbor information.
- **Performance**: Avoid heavy I/O operations inside module loops.