HEP Agent Simulation Framework

Code Documentation

February 19, 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.

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.

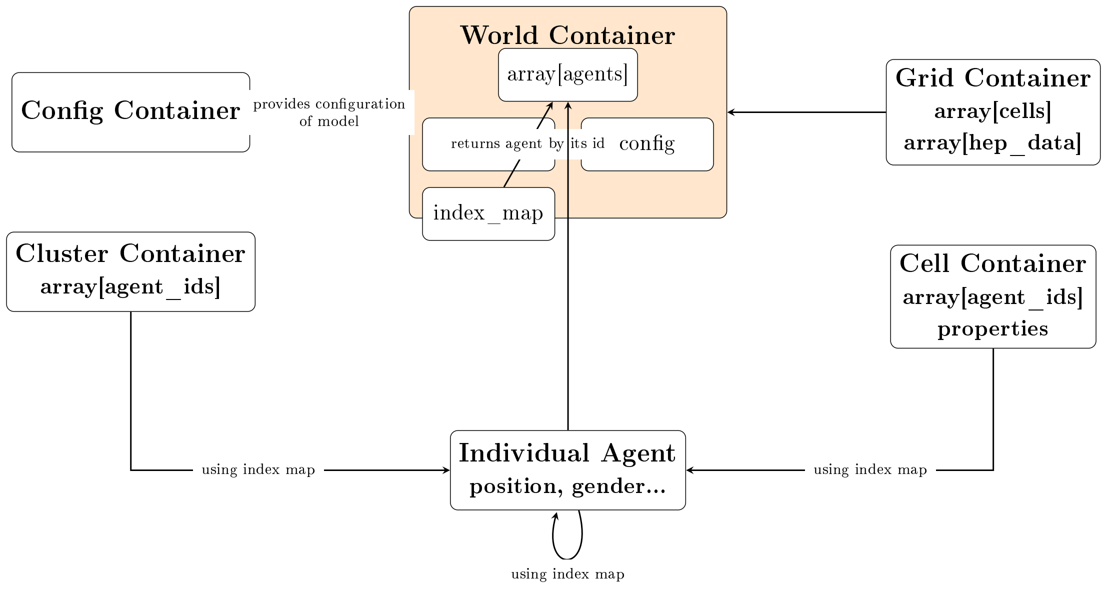

Figure 2.1: Visual representation of the hybrid data structure. The `world_container` acts as the root, managing contiguous memory arrays for agents while maintaining spatial (Grid) and logical (Hashmap) indices for efficient access.

2.1.2 The Agent Array

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

```fortran
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.

```fortran
    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`.

```fortran
    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.

2.4 Watershed Clustering

The simulation includes a watershed-based spatial clustering module (`mod_watershed` + `mod_clustering`) that groups grid cells into clusters based on the HEP surface.

2.4.1 Algorithm

The clustering proceeds in three steps:

1. **Smoothing**: A box filter with half-width `watershed_smooth_radius` is applied to the HEP surface to reduce over-segmentation from noise.
2. **Local maxima detection**: Cells strictly higher than all 8 neighbours (and above `watershed_threshold`) become cluster seeds.
3. **Gradient ascent**: Every remaining unlabelled cell above the threshold follows the steepest-uphill neighbour until a labelled cell is reached. All cells on the path inherit that label.

Cells below the threshold remain unlabelled (noise/water). Cluster IDs are persistent across re-clustering via cell-overlap matching.

2.4.2 Configuration

The watershed clustering parameters are stored in `world_config` and read from the namelist file `basic_config.nml`:

Parameter	Type	Default	Description
`watershed_smooth_radius`	integer	2	Half-width of the box-filter used for smoothing the HEP surface before clustering. Set to 0 to disable smoothing.
`watershed_threshold`	real(8)	0.05	Cells with HEP values at or below this threshold are classified as noise and excluded from clustering.

Example configuration in `basic_config.nml`:

```
!=====================================================
!============ Watershed Clustering ====================
!=====================================================
watershed_smooth_radius = 2
watershed_threshold = 0.05
```

Chapter 3

Developing New Modules

3.1 Overview

A "Module" in this framework is a self-contained behavior that can be applied to agents or to the grid (e.g., Movement, Aging, Mating). Modules are written in Fortran, registered in the interface layer, and toggled on/off from the Python UI.

There are **two ways** to develop new simulation logic, depending on your experience level and workflow preference:

Path A — Independent Development
You create a full module from scratch: write the Fortran subroutine, register it, wire it into the dispatch loop, and add it to the UI. Best for users comfortable with the full codebase.

Path B — Collaborative (Test Modules)
You write your logic directly in the pre-wired `test_modules.f95` file, test it, and send the file to Daniel. Daniel then turns your prototype into a proper, permanent module. Best for users who want to focus on the science without touching multiple files.

3.2 Path A: Independent Module Development

Use this path when you are confident editing multiple Fortran and Python files. Figure 3.1 shows the full workflow.

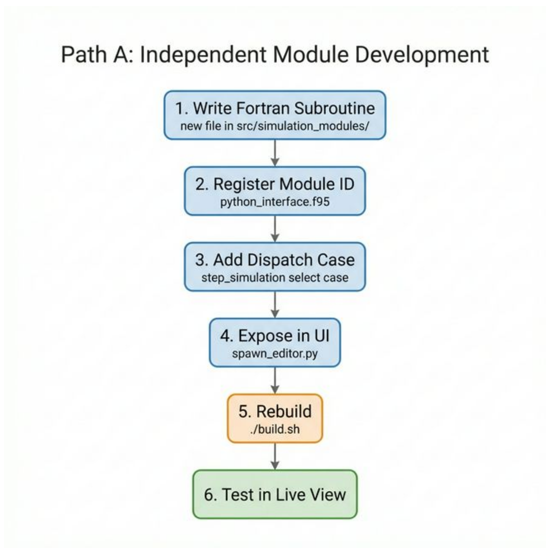

Figure 3.1: Path A: Creating a standalone module requires changes in four files.

3.2.1 Step 1: Implement the Logic (Fortran)

Create a new subroutine. There are two patterns depending on what your module operates on:

Agent-Centric Module

Runs once per living agent. The subroutine receives a pointer to the current agent.

```fortran
subroutine my_agent_behavior(agent_ptr)
    use mod_agent_world
    implicit none
    type(Agent), pointer, intent(inout) :: agent_ptr

    ! Example: age-based death
    if (agent_ptr%age > 3000) then
        call agent_ptr%agent_dies(reason=6)
    end if
end subroutine my_agent_behavior
```

Grid-Centric Module

Runs once per tick over the spatial grid. Receives the full world container.

```fortran
subroutine my_grid_behavior(w)
    use mod_agent_world
    implicit none
    class(world_container), target, intent(inout) :: w

    integer :: gx, gy
    do gy = 1, w%config%dlat_hep
```

```
        do gx = 1, w%config%dlon_hep
            ! operate on w%grid%cell(gx, gy)
        end do
    end do
end subroutine my_grid_behavior
```

3.2.2 Step 2: Register the Module ID

In `src/interfaces/python_interface.f95`, define a new integer constant. Pick the next available number:

```
integer, parameter :: MODULE_MY_BEHAVIOR = 17
```

3.2.3 Step 3: Add the Dispatch Case

In the `step_simulation` subroutine, add a `case` to the `select case` block:

```
case (MODULE_MY_BEHAVIOR)
    ! For agent-centric:
    call apply_module_to_agents(my_agent_behavior, t)
    ! For grid-centric:
    ! call my_grid_behavior(world)
```

3.2.4 Step 4: Expose in the UI

In `python/spawn_editor.py`, add your module to the `available_modules` dictionary (ID must match the Fortran constant):

```
self.available_modules = {
    # ... existing modules ...
    "My New Behavior": 17,
}
```

3.2.5 Step 5: Rebuild & Test

```
./build.sh
python3 python/application.py
```

Add your module from the Spawn Editor and run the simulation to verify.

3.3 Path B: Collaborative Workflow (Test Modules)

Use this path when you want to **focus on writing your logic** without needing to touch the interface, build scripts, or registration code. Everything is already wired up for you.

Figure 3.2 shows the collaborative workflow.

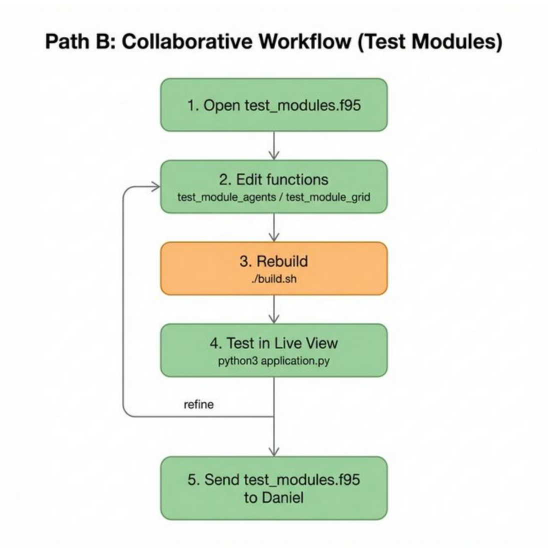

Figure 3.2: Path B: Edit the test module, rebuild, test, and iterate until ready.

3.3.1 Step 1: Open the Test Module File

Open `src/simulation_modules/mod_test_modules.f95`. You will find two subroutines:

`test_module_agents` Runs your code on **every living agent**, once per tick.

`test_module_grid` Runs your code **once per tick** on the entire grid.

Choose the one that fits your use case (or use both).

3.3.2 Step 2: Write Your Logic

Each subroutine has a clearly marked section:

```
! ==========================================
! DECLARE YOUR VARIABLES HERE
! ==========================================

! ==========================================
! YOUR CODE HERE
! ==========================================
```

Important:

- Declare all variables you need at the top of the subroutine (after `implicit none`).

- You do **not** need to modify any config files.
- Use `print *, "..."` followed by `call flush(6)` for debug output.
- The file contains detailed comments listing all available agent fields, grid accessors, and example code.

3.3.3 Step 3: Rebuild & Test

```
./build.sh
python3 python/application.py
```

In the Spawn Editor, add **"Test Module (Agents)"** or **"Test Module (Grid)"** to the active module list, then run the simulation. Check the terminal for your `print` output.

3.3.4 Step 4: Iterate

Repeat steps 2–3 until your results are correct. When you are satisfied, move on to step 5.

3.3.5 Step 5: Hand Off to Daniel

Send your modified `mod_test_modules.f95` file to Daniel. He will:

1. Review and optimize your code.
2. Extract it into a proper, standalone module with its own file and config parameters.
3. Merge it into the main codebase so it becomes a permanent, selectable module.

3.4 Best Practices

- **Statelessness**: Keep modules stateless where possible. Store state in agent fields or in module-level `save` variables.
- **Grid Access**: Use `w%grid` to access spatial neighbour information.
- **Performance**: Avoid file I/O inside per-agent loops. Use `print` sparingly (e.g., only on specific ticks).
- **Debug Output**: Always call `flush(6)` after `print` statements to ensure output appears before a potential crash.

Chapter 4

Fortran Only Version

The simulation framework includes a standalone Fortran executable, allowing users to run the model without the Python interface or GUI. This is particularly useful for high-performance computing (HPC) environments, batch processing, or profiling.

4.1 Building the Executable

The standalone version is built using the provided shell script `build_fortran.sh`. This script compiles all necessary modules, creates a static library (`libhep.a`), and links the final executable.

4.1.1 Prerequisites

- **Compiler**: `gfortran` (or compatible Modern Fortran compiler).
- **Libraries**: NetCDF and NetCDF-Fortran development libraries (`libnetcdf-dev`, `libnetcdff-dev`).

4.1.2 Build Command

To build the project, run the following command from the project root:

```
./build_fortran.sh
```

This will create:

1. A `build/` directory containing compiled object files (`.o`) and module files (`.mod`).
2. A static library `build/libhep.a`.
3. The standalone executable `build/main_fortran`.

4.2 Running the Simulation

The executable can be run directly from the command line. It reads the specific configuration file located at `input/config/main_fortran_config.nml`.

4.2.1 Execution

```
./build/main_fortran
```

4.2.2 Command Line Arguments

The standalone version supports command-line arguments to override specific runtime parameters:

- `-output_interval <integer>`: Sets the interval (in ticks) at which simulation status is printed to the console. Default is 1000.

Example:

```
./build/main_fortran --output_interval 500
```

4.3 Configuration

Unlike the Python version which uses `basic_config.nml`, the standalone Fortran version hardcodes the configuration path to:

`input/config/main_fortran_config.nml`

Ensure this file exists and contains valid namelists (`&dims` and `&config`) before running. The structure of this file is identical to the standard configuration files used by the GUI.