HEP Agent Simulation Framework

Interface Documentation

February 18, 2026

Contents

Chapter 1

Introduction

The HEP Agent Simulation Framework provides a graphical user interface (GUI) to configure the fortran simulation.

Chapter 2

Main Graphical Interface

When launching ‘application.py‘, the user is presented with the main window containing four primary tabs (see Figure 2.2):

- **Configuration**: For managing simulation settings and HEP inputs.
- **Spawn Editor**: For defining initial agent distribution.
- **View Editor**: For configuring visualization settings.
- **Full Simulation**: For running headless, long-duration simulations.

At the bottom of the window, buttons are available to launch the simulation (Figure 2.1).

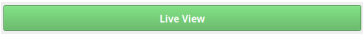

Figure 2.1: Live View Button

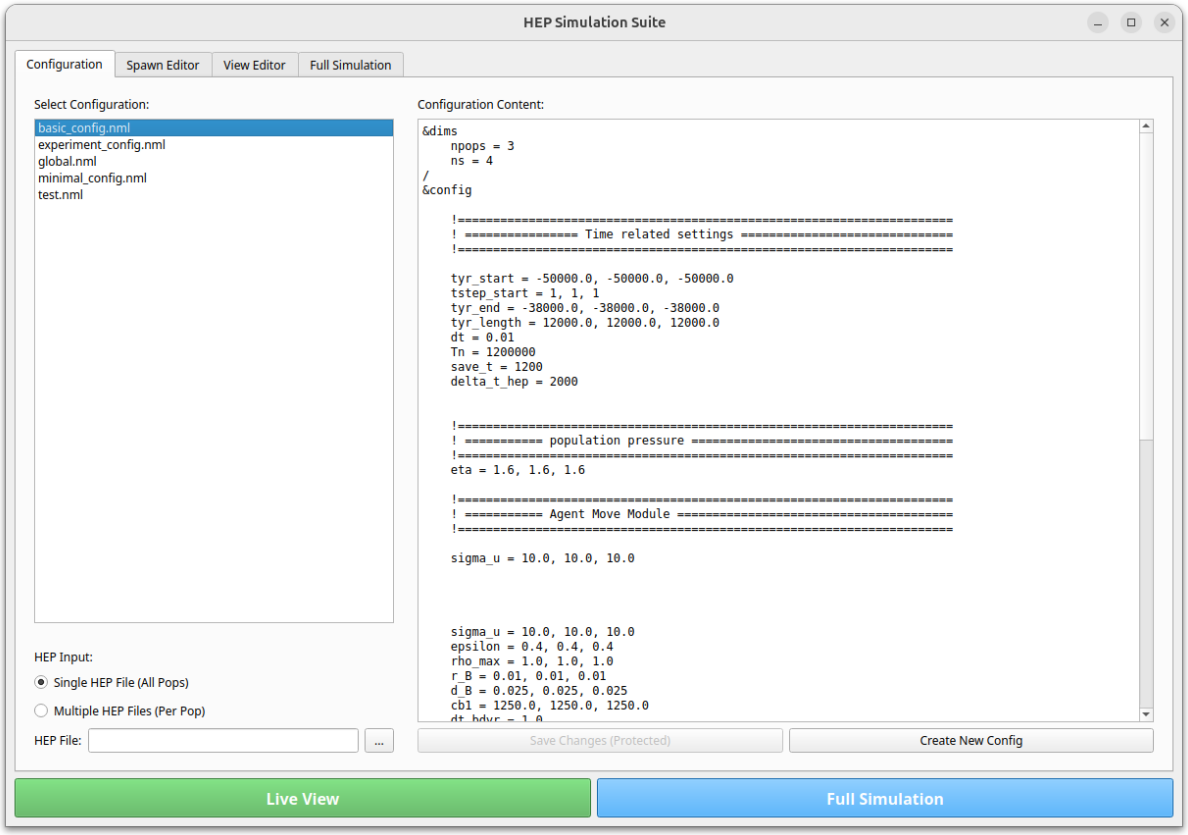

Figure 2.2: Overview of the Main Application Window (Configuration Tab)

Chapter 3

Configuration Tab

The Configuration tab serves as the entry point for setting up the simulation parameters.

3.1 Configuration List

On the left side, a list of available '.nml' configuration files is displayed (see Figure 3.1). Users can select a configuration to view and edit its content in the right-hand text editor.

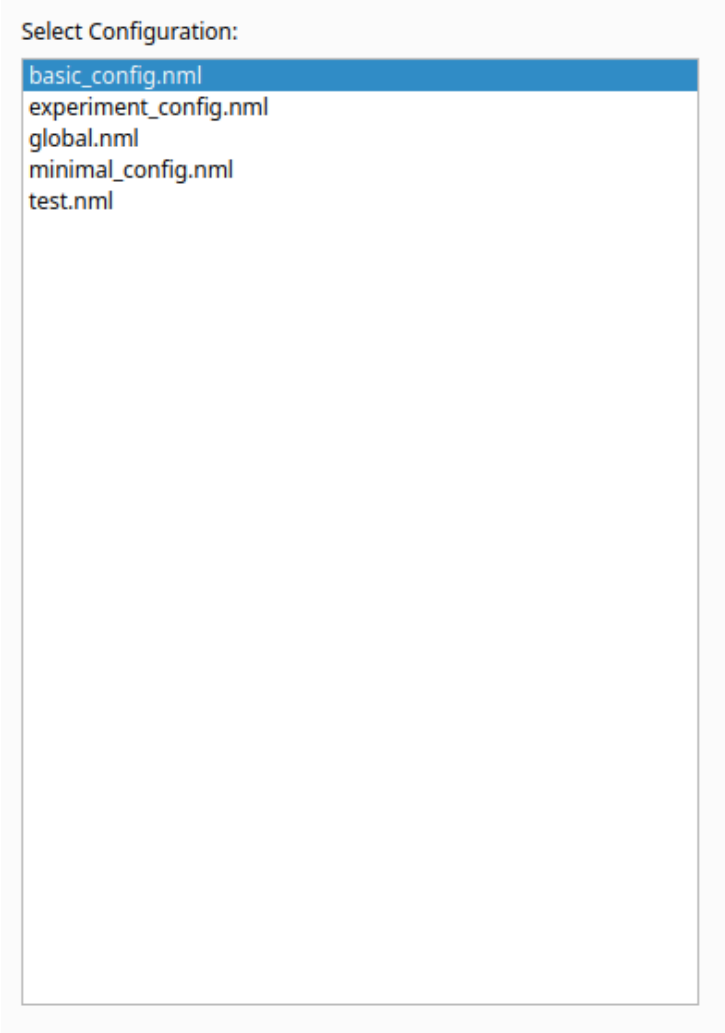

Figure 3.1: Selecting a Configuration File

The editor allows direct modification of the simulation parameters within the application (Figure 3.2). Note that the `basic_config.nml` serves as a template and cannot be modified directly in the editor.

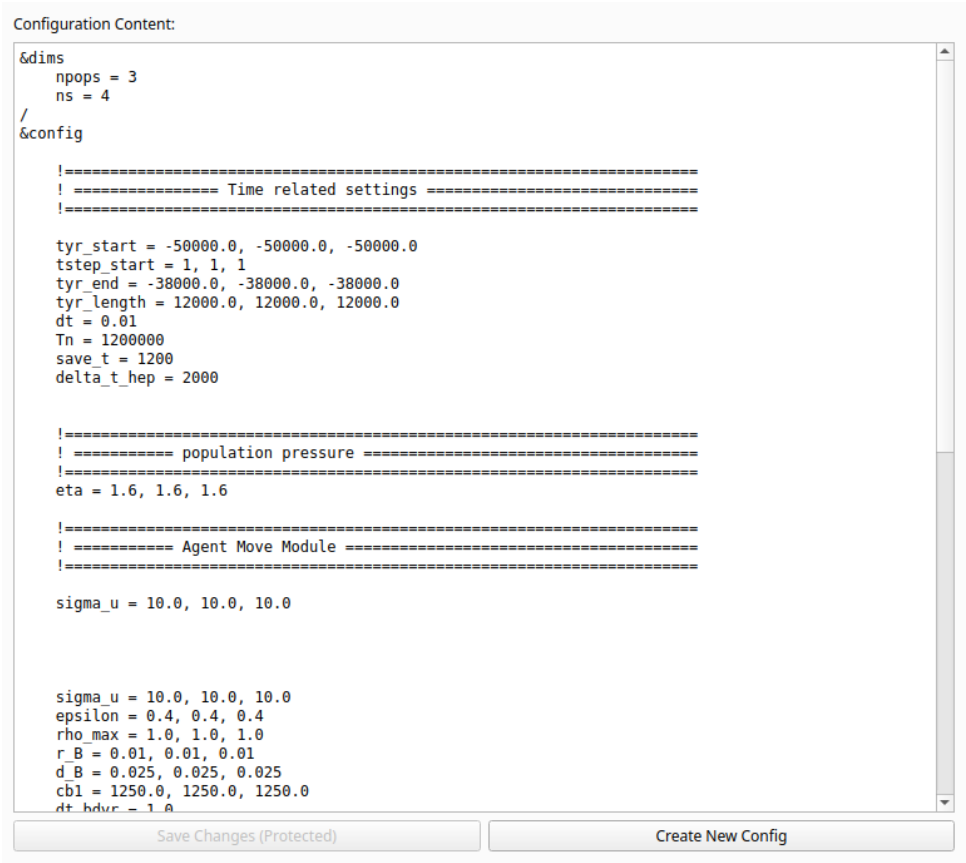

Figure 3.2: Editing Configuration Parameters

3.2 HEP Input Selection

Users can select the HEP input files (Figure 3.3):

- **Single HEP File**: Uses the same hep file for all populations.
- **Multiple HEP Files**: Allows specifying distinct hep files for each population.

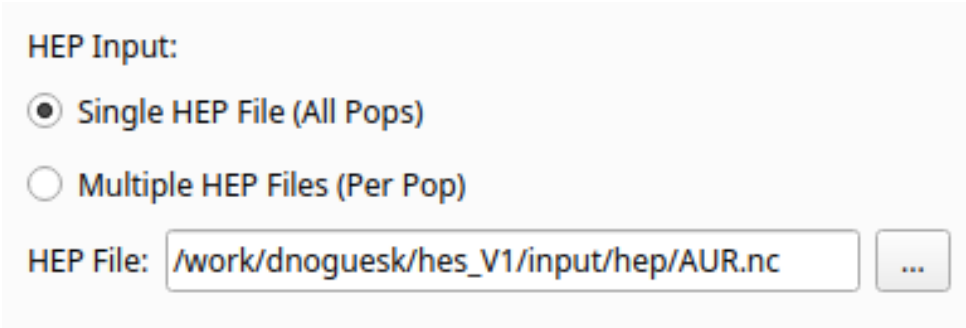

Figure 3.3: Selecting HEP Input Files

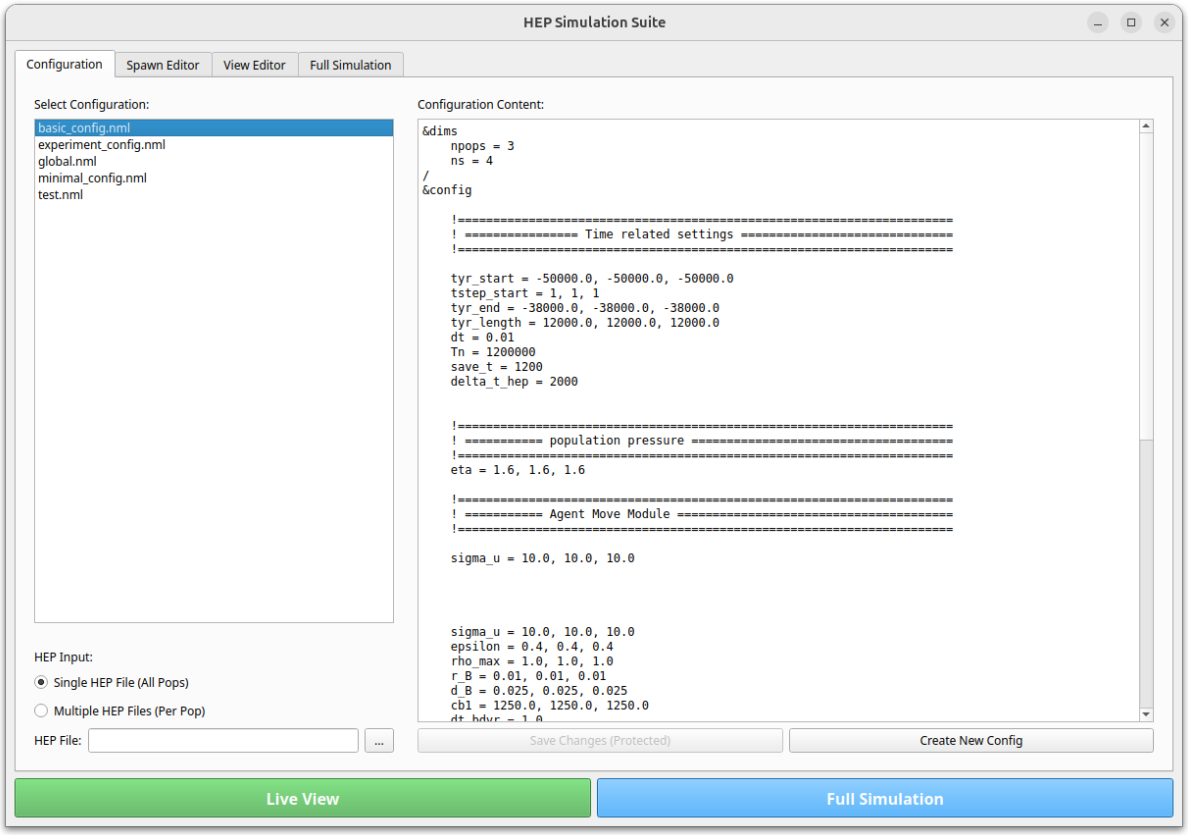

Figure 3.4: Configuration Tab showing file selection and editor

Chapter 4

Spawn Editor Tab

The Spawn Editor allows users to define where agents are initially placed on the map, configure active simulation modules, and set initial age distributions (Figure 4.1). This tab is only available after a HEP file and a configuration file have been selected. The left panel is scrollable to accommodate all controls on smaller screens.

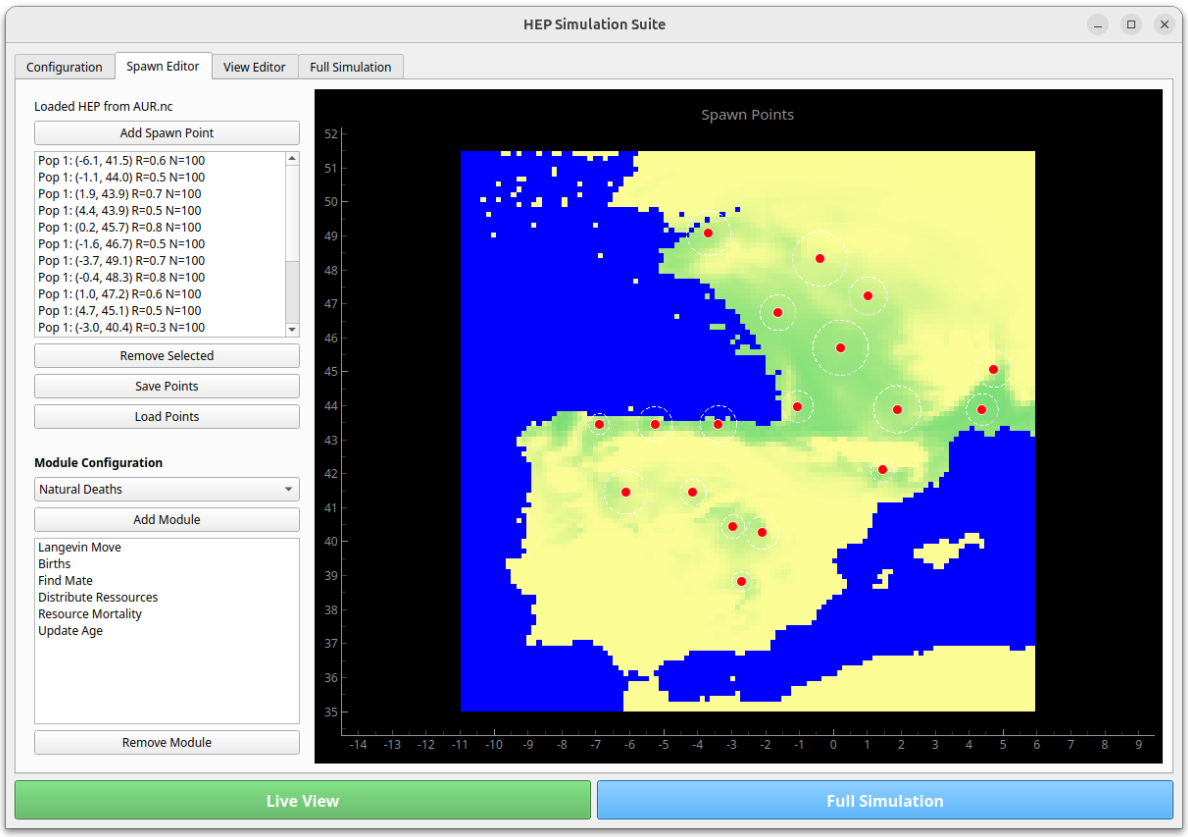

Figure 4.1: Spawn Editor Interface

4.1 Module Configuration

The user can activate or deactivate specific simulation modules for the agent population (see Figure 4.2). Modules are displayed in a **grouped tree view** with three columns:

Module The name of the module.

Author Who developed the module.

File The Fortran source file where the module is implemented.

Modules are organised into groups such as *Core*, *ProofOfConcept*, *Movement*, *Templates*, and *YapingDevelopment*. To add a module, select it in the tree and click "Add Module" (or double-click it). Active modules appear in the list below and can be reordered by drag-and-drop.

Note that new modules only appear in this tree after the Fortran code has been recompiled and registered in the framework.

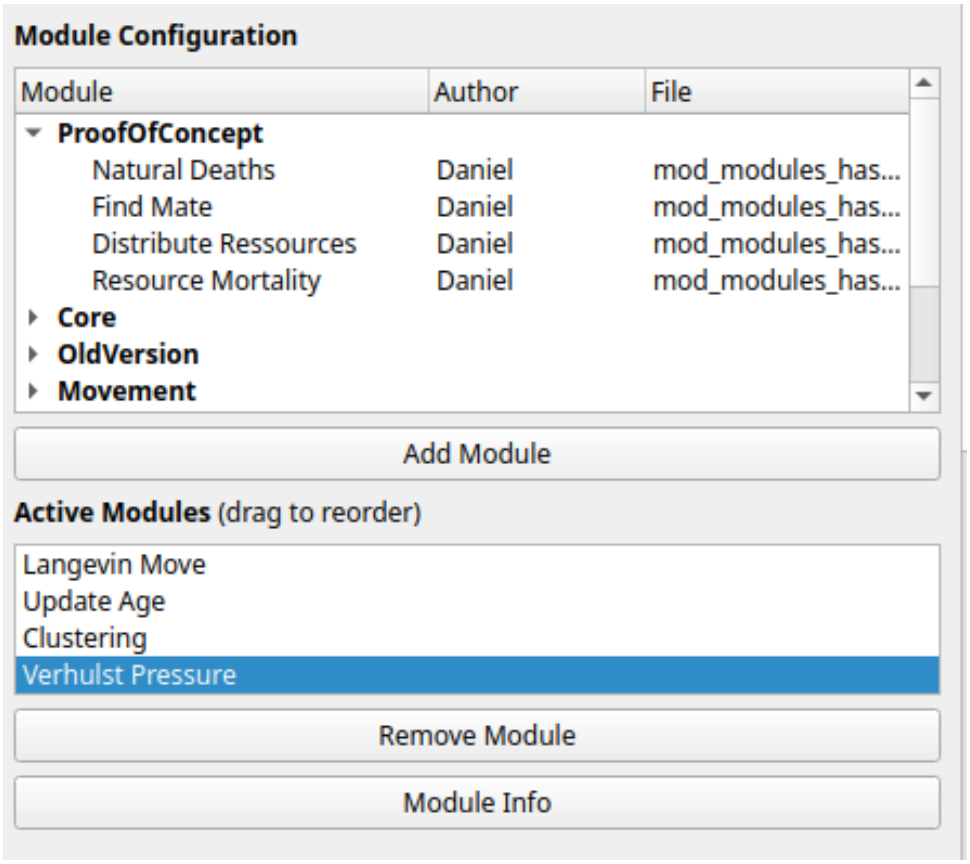

Figure 4.2: Module Configuration Panel showing grouped modules with Module, Author, and File columns.

4.1.1 Test Modules

Two special template modules are included for rapid prototyping:

Test Module (Agents) Runs user-defined code once per tick on every living agent. Use this when your logic operates on individual agents (e.g., movement rules, death conditions).

Test Module (Grid) Runs user-defined code once per tick on the entire spatial grid. Use this when your logic depends on spatial relationships between agents or grid-cell properties.

To use them, edit `src/simulation_modules/mod_test_modules.f95`, write your logic in the clearly marked sections, rebuild with `./build.sh`, and toggle the module on in this panel. For a detailed walkthrough, see Chapter 3 of the *Code Documentation*.

4.1.2 Yaping Development Modules

Four development modules are available for Prof. Yaping's research group, all implemented in `mod_yaping_development.f95`:

Yaping Move Agent-centric movement module.

Yaping Birth Grid Grid-centric birth module.

Yaping Death AGB Agent-centric death module (age/gender-based).

Yaping Death Grid Grid-centric death module.

These follow the same template pattern as the test modules: edit the source file, rebuild, and activate in the module panel.

4.2 Spawn Point Management

Users can add, remove, and manage spawn points directly from the list (Figure 4.3).

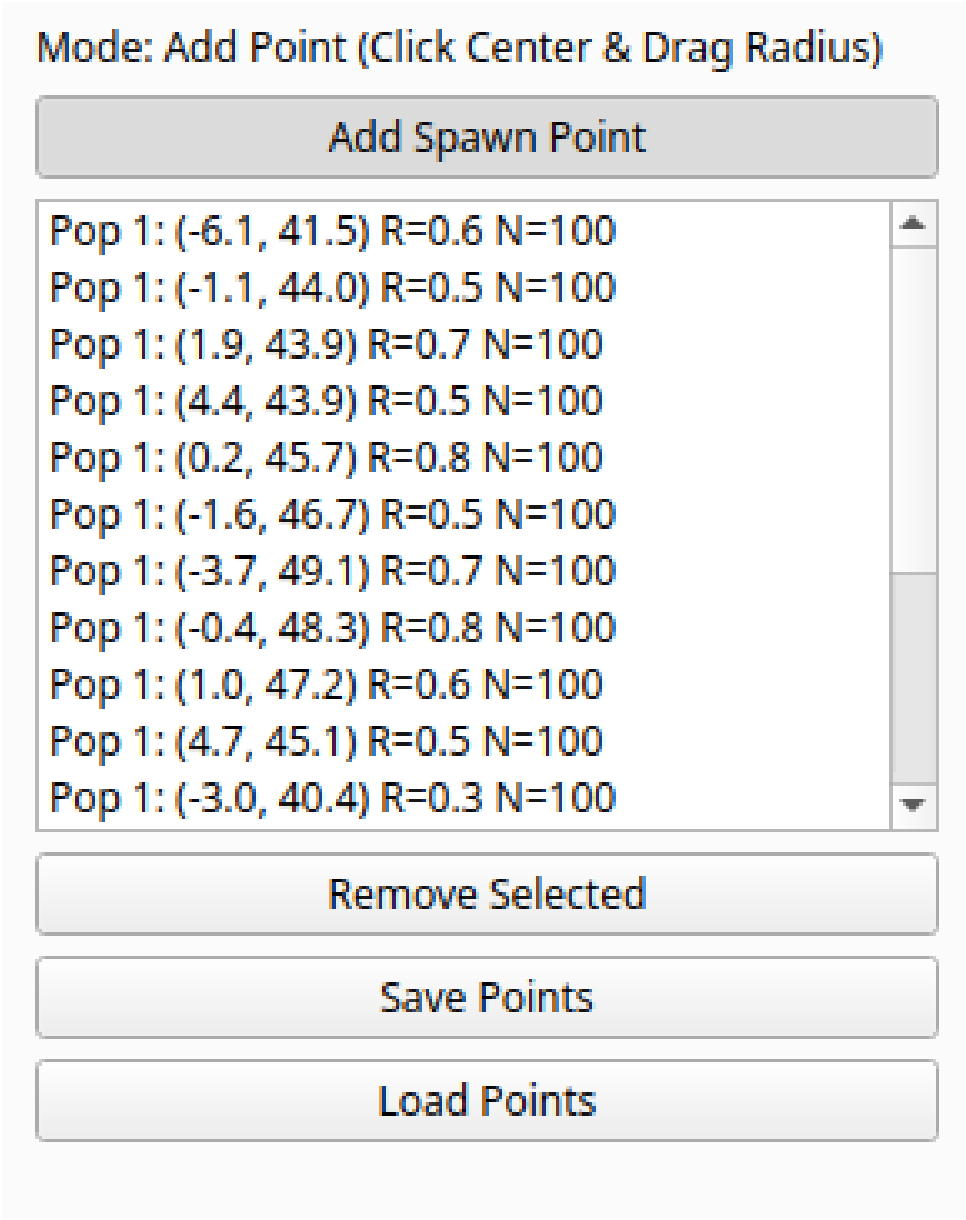

Figure 4.3: Spawn Point Management List

4.3 Preview Context

The editor allows previewing spawn locations overlaid on the selected HEP environment (Figure 4.4).

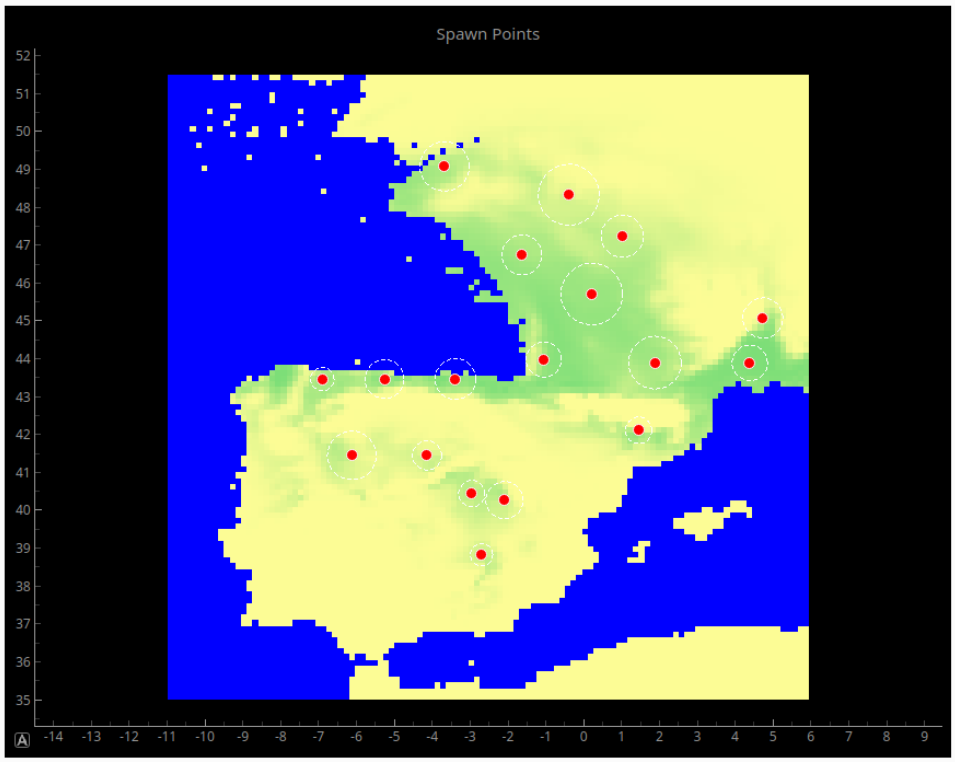

Figure 4.4: Preview of Spawn Points on HEP Map

4.4 Initial Age Distribution

The Spawn Editor includes an optional age distribution feature (Figure **??**). When enabled, agents are spawned with ages drawn from a configured distribution rather than all starting at age zero.

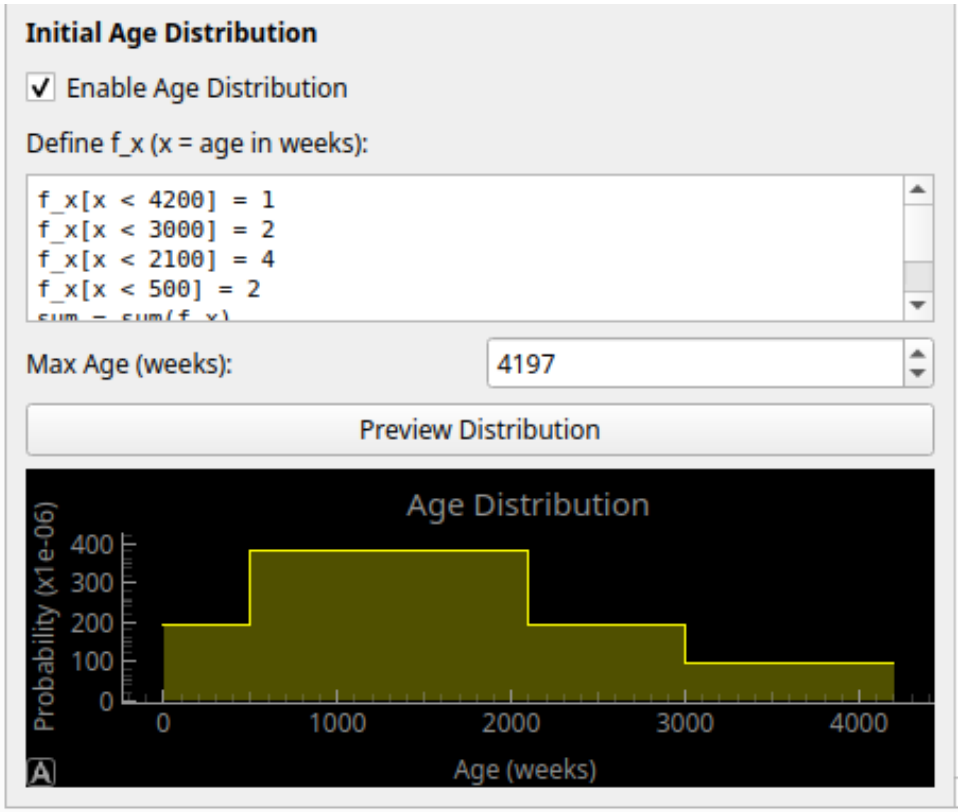

Figure 4.5: Initial Age Distribution configuration and preview.

Users can:

- Enable or disable the age distribution with a checkbox.
- Configure the distribution parameters.
- Preview the resulting distribution as a histogram before starting the simulation.

Chapter 5

View Editor Tab

The View Editor controls the visualization parameters for the Live View execution (see Figure 5.1).

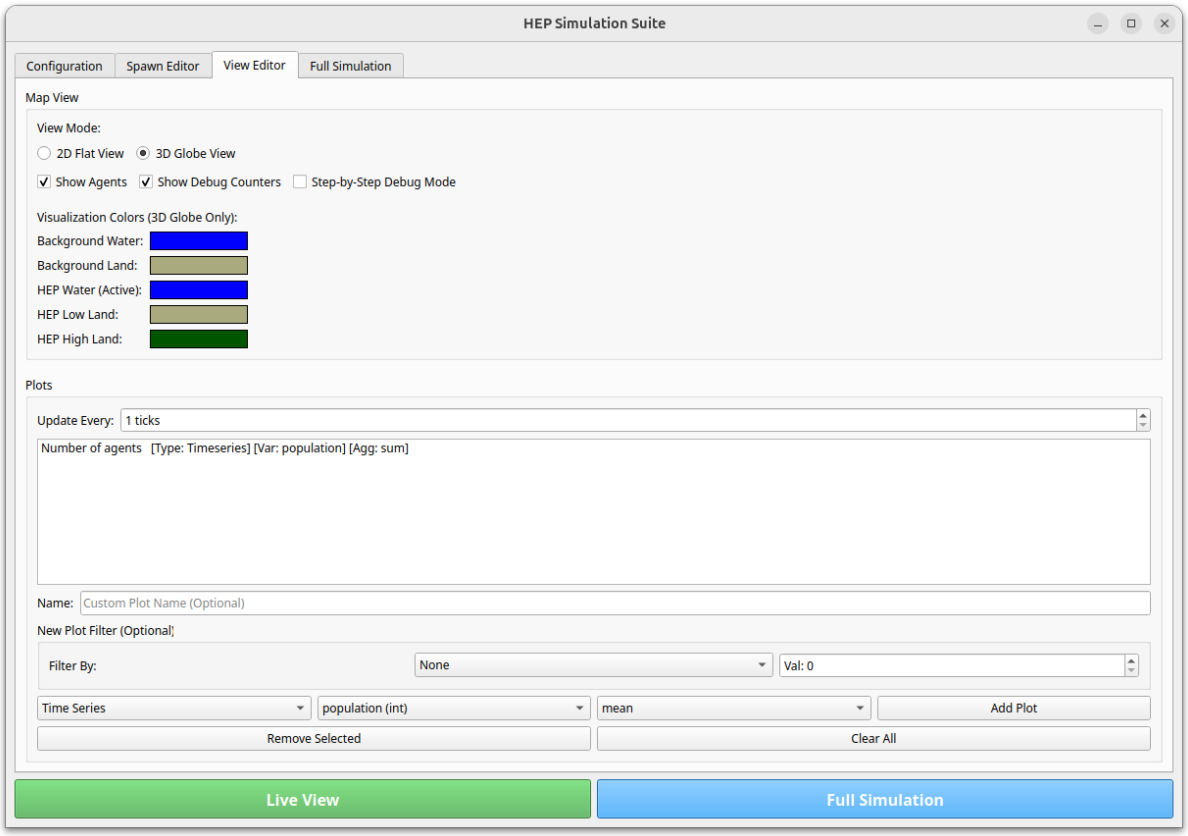

Figure 5.1: View Editor Overview

5.1 Map View Settings

Users can toggle between visualization modes (Figure 5.2):

- **2D Flat View**: Standard map projection.
- **3D Globe View**: Spherical projection of the simulation data.

Additional toggles allow showing/hiding agents and debug counters.

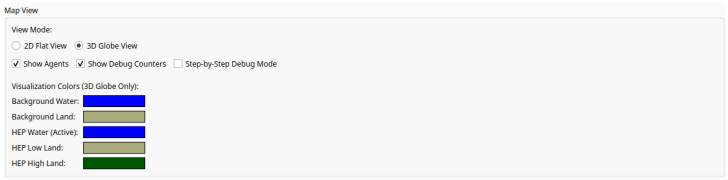

Figure 5.2: Map View and Visualization Settings

5.2 Plot Configuration

A dynamic plotting interface allows users to add real-time analysis charts (Figure 5.3). Supported plot types include:

- **Time Series**: Tracks variable evolution over time (e.g., Total Population).
- **Bucket Plot**: Demographics distribution (e.g., Age cohorts).
- **Count Plot**: Conditional counts (e.g., Number of pregnant agents).

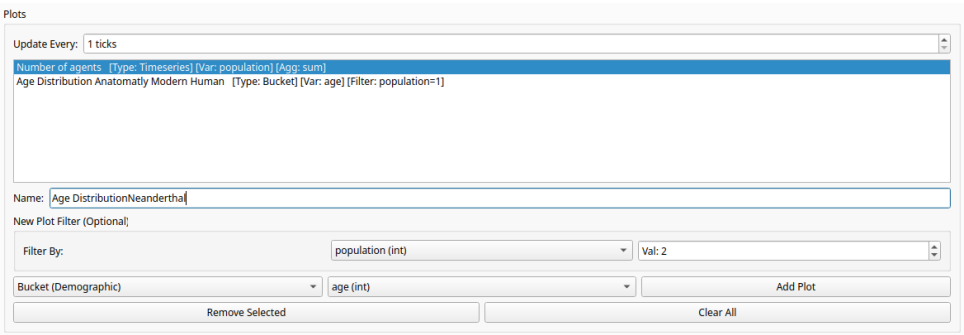

Figure 5.3: Plot Creation Tool

Chapter 6

Full Simulation Tab

The Full Simulation tab is designed for running high-performance, non-interactive simulations (see Figure 6.1).

Disclaimer: This feature has not been finished yet and is very much a work in progress.

6.1 Parameters

- **Start/End Year**: Defines the simulation timeline.
- **Output File**: Specifies the path for the simulation recording (.gif).

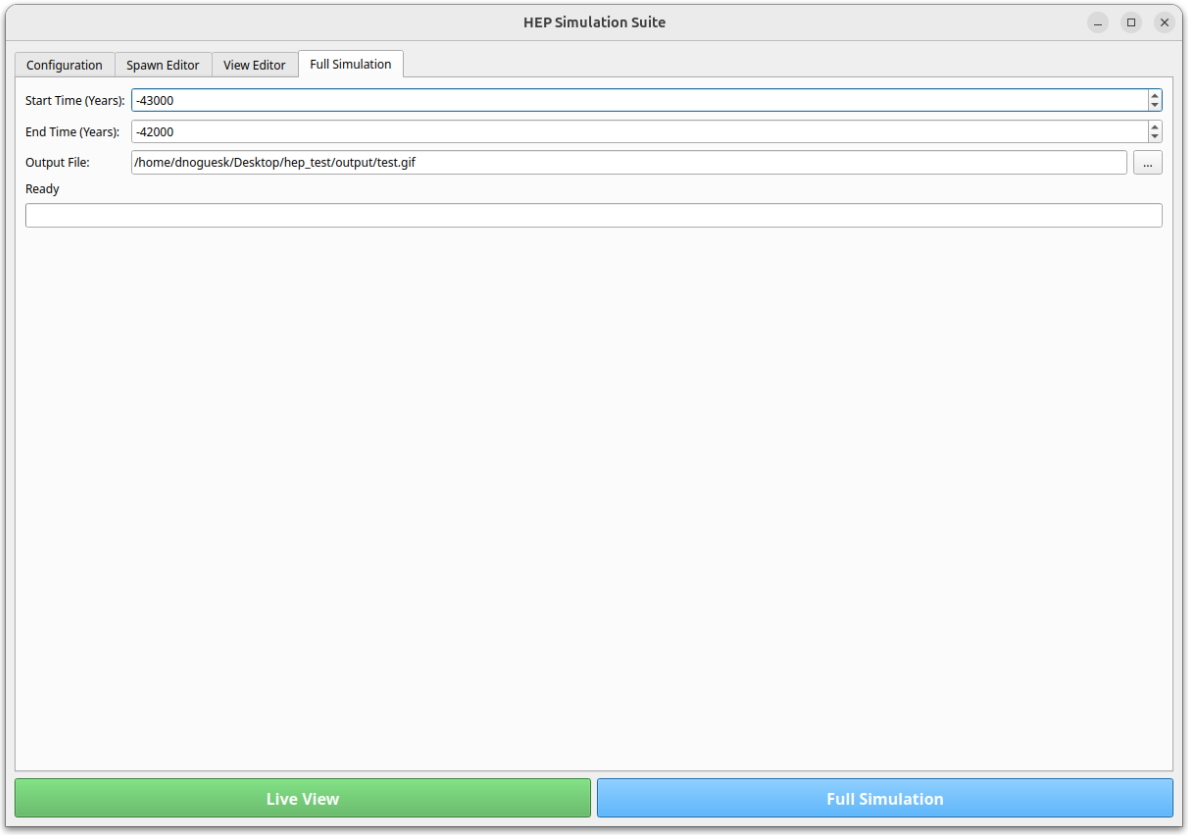

Figure 6.1: Full Simulation Execution Interface

Chapter 7

Debugging Fortran Modules

For instructions on how to **create** a new simulation module and register it in the framework, please refer to the *Code Documentation* (Chapter 3: "Developing New Modules"). This chapter assumes that you are using the **test modules** (`mod_test_modules.f95`) for the development of your new modules.

7.1 Development Workflow Overview

Developing and debugging a Fortran module requires switching between the Fortran source code, the build system, and the Python UI. Figure 7.1 illustrates the typical cycle when working with the test modules.

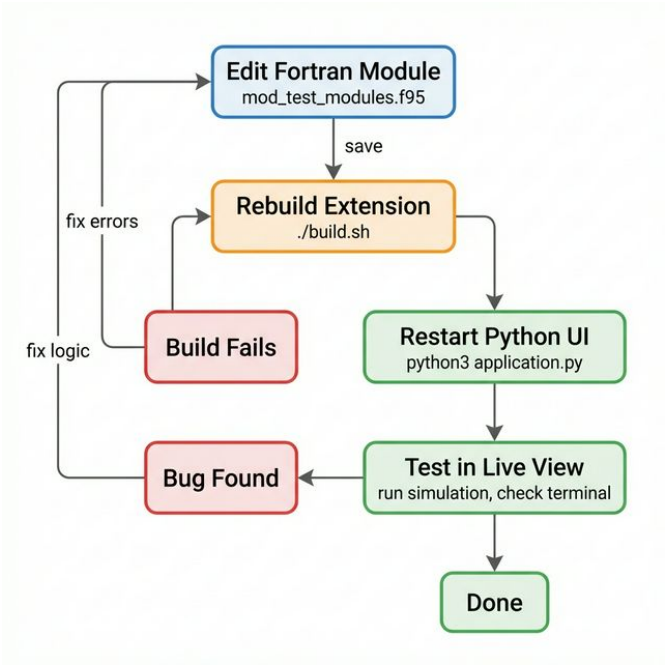

Figure 7.1: The edit–build–test cycle for Fortran module development. After editing Fortran source files, the extension must be rebuilt and the Python UI restarted before changes take effect.

7.2 Print-Based Debugging

Fortran `print` statements are the simplest way to inspect values at runtime. Their output appears in the terminal that launched `application.py`.

```
subroutine my_module(agent_ptr)
    type(Agent), pointer, intent(inout) :: agent_ptr

    print *, "Agent ID:", agent_ptr%id, &
             "pos:", agent_ptr%pos_x, agent_ptr%pos_y
    call flush(6)    ! force output immediately
end subroutine my_module
```

Tip: Always call `flush(6)` after print statements, otherwise output may be lost if the program crashes before the buffer is flushed.

7.3 Common Crash Causes

The most frequent runtime errors when developing Fortran modules are:

Segmentation Fault Usually caused by accessing an unallocated or out-of-bounds array, or dereferencing a null pointer. Common sources:

- Accessing `agents(k, pop)` with `k` beyond `num_humans(pop)`.
- Using grid indices `(gx, gy)` that are outside the grid dimensions.
- Referencing a dead agent (`is_dead = .true.`) that has been compacted away.

Array Bounds Errors Enable runtime bounds-checking by adding `-fcheck=all` to the Fortran compiler flags in the build script. This turns out-of-bounds accesses into clear error messages instead of silent corruption:

```
# In build_fortran.sh, add to FFLAGS:
FFLAGS="$FFLAGS -fcheck=all -g -fbacktrace"
```

`-g` includes debug symbols, and `-fbacktrace` prints a stack trace on crashes.

NaN / Infinity Floating-point issues can propagate silently. Use `-ffpe-trap=invalid` to trap these at the point of origin:

```
FFLAGS="$FFLAGS -ffpe-trap=invalid,zero,overflow"
```

7.4 Configuration Validation

When adding new configurable parameters to `world_config` (see the Code Documentation), verify that:

1. The variable is declared in `mod_config.f95` with a sensible default.
2. The variable is added to the `namelist /config/` block in `mod_read_inputs.f95`.
3. A default value is set **before** the `read(unit, nml=config)` call.
4. The variable is assigned to the `cfg%` struct **after** the read.
5. The variable appears in `basic_config.nml`.

If any step is missing, the parameter may silently retain an uninitialised value.

7.5 Debug Counters Overlay

The Live View provides a "Show Debug Counters" checkbox (in the View Editor tab) that displays an overlay with runtime statistics including death causes, grid-index errors, and movement calls. These counters are updated

each simulation tick and are useful for verifying that a module's effects are applied as expected.

To add a custom counter:

1. Add an integer field to the `counter` type in `mod_agent_world.f95`.
2. Increment it inside your module logic.
3. Expose it via `get_debug_stats` in `python_interface.f95`.

7.6 Rebuild & Test Cycle

After making changes to Fortran source files, always:

1. Rebuild the extension:

   ```
   ./build.sh
   ```

2. Check the terminal output for compilation warnings — they often indicate real bugs.
3. Restart `application.py` (the old `.so` is cached by the Python process).