Documentation of `ABM-Hescor`

Daniel Nogues
University of Cologne

October 1, 2025

Contents

1 Introduction

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), kinship relations, 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.
The simulation is an agent-based model. 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, Brownian molecular motion in an agent-based model is not simulated by solving a PDE over time, but by generating random movement vectors for all simulated particles. Time is then represented by iterating this procedure.
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), kinship relations, 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.
The simulation is an agent-based model. 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, Brownian molecular motion in an agent-based model is not simulated by solving a PDE over time, but by generating random movement vectors for all simulated particles. Time is then represented by iterating this procedure.

2 Installation

Since this Program is used only internly of the Hescor Group. We assume that you use Ubuntu. The program can be found in the folder `hep_extension` inside the directory `dnoguesk`, which itself is located in the HESCOR work group folder. To set up the program on your machine, copy the folder `hep_extension` to your own workspace. You may rename it if you wish. Then, navigate into this folder on your machine.

2.1 Requirements

Make sure python and fortran, pip and venv are installed:

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

2.1.1 Fortran Packages

So far the Program requieres no Fortran Packages that need to be installed.

2.1.2 Python Packages

Make sure that the following Python Packages are installed:

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

2.2 Setup

Once inside the project folder:

1. Activate the virtual environment:

   ```
   source venv/bin/activate
   ```

2. Build the project:

   ```
   make all
   ```

3. If you encounter errors during the build, try cleaning first:

   ```
   make clean
   make all
   ```

2.3 Running the Program

After a successful build, you can run the program using:

```
./bin/main_agb
```

or alternatively (requieres pandas and numpy):

```
./run_all.sh main_agb
```

2.4 Optional: Python for Visualization

Some of the provided Python scripts are used for visualization and data analysis. Depending on your system setup, you may need to install additional Python packages inside the virtual environment.
To do so, first make sure the virtual environment is active:

```
source venv/bin/activate
```

Then, install the required packages with `pip`. For example:

```
pip install matplotlib numpy pandas
```

If you encounter missing-package errors while running the scripts, simply install the corresponding packages using `pip` in the same way.

3 Working Goals

3.1 Daniel

3.1.1 Structure

I would like to reach a point where one can write a function that takes an agent as the input:

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

            real(8) :: new_x, new_y
            real(8) :: old_x, old_y

            old_x = agent_ptr%pos_x
            old_y = agent_ptr%pos_y

            calculate_new_position(old_x,old_y,new_x,new_y)

            agent_ptr%pos_x = new_x
            agent_ptr%pos_y = new_y

        end subroutine agent_move
```

Then the program applies this function to every agent. I would like the main program to look something like this:

```
    do t = 1, TN

        current_agent = agent_head

        do while associated(current_agent)

            agent_move(current_agent)

            agent_reproduce(current_agent)

            ...

        enddo

    enddo
```

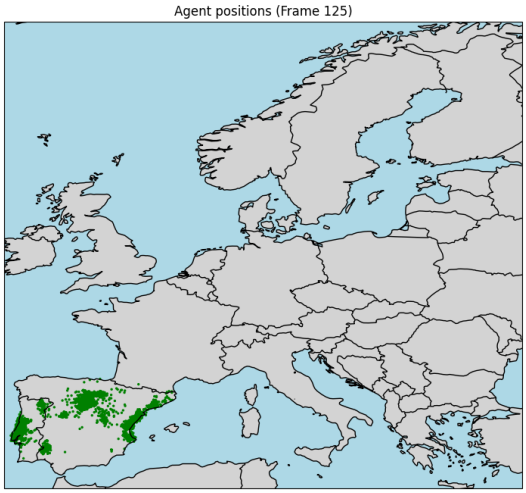
I think that once this is achieved many people can work simultaniously on the actual simulation. And their work can easily be integrated.
One Simulation module would be represented by (ideally one) subroutine(s).
Also this structure makes it very easy to see what parameters the different simulation modules need, since that would be all variables the function uses other than its input.

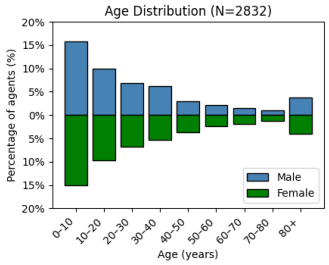
3.1.2 Visualisation

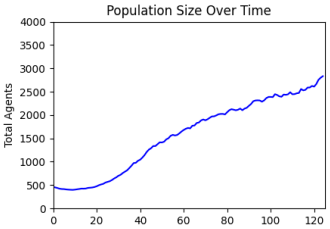
I would like to visualise as much as possible of each simulation. I believe that this is essential for a successfull and wuick development of the model. For example a good visualization of the age demographic makes it much easier to develop a good agent based reporduction model. Especially if we have a certain age cure in mind. There is already visualization for:

- Positions of the agents

- age demographics of population
- number of agents alive accros time (population curve)







Next I want to realise a visualisation of the following:

- ...

3.2 Ruben

3.3 Y. Shao

4 Program Structure

4.1 General Structure

The project directory is organized into several subfolders. Understanding their purpose will help you navigate the code, run simulations, and analyze results.

`src` Contains the Fortran source code of the simulation. This is where the main computational logic is implemented.

`build` When running `make all`, the compiled Fortran code is placed here before being linked into the final executables.

`bin` Contains the final executable files produced by the build process (e.g., `main_agb`). We might want different executables for the same simulation:

- While developing you might want to run only $n < 10, 100, 1000$ time ticks.
- while developing you might want to run the simulation without saving the data.
- maybe we endup with different simulations that we compare. (One with movement pattern X and one with movement pattern Y
- ...

`output` The simulation results produced by the Fortran code are written into this folder.

`python` Contains Python scripts for analysis and visualization of the simulation outputs.

`animation_output` Stores animations and visualizations generated by the Python scripts.

`hep_animation_output` Used for debugging purposes. Contains visualizations of the HEP that are generated by Python.

`hep_control` Also used for debugging. Stores the HEP currently being used in CSV format.

`Example_Animations` A collection of sample animations that illustrate the progress of the project's development.

4.2 Structure of the Fortran Program

The Fortran code is organized into multiple modules and source files inside the `src` folder. Modules typically contain reusable functions and subroutines, while the main program orchestrates the simulation workflow.

4.2.1 Generating Documentation with `ford`

To help understand and navigate the Fortran code, you can use `ford` to automatically generate documentation.

1. Install `ford` if you do not have it already. On Linux:

```
pip install ford
```

2. Navigate to the project folder (the one containing `src`):

```
cd path/to/hep_extension
```

3. Run `ford` to generate HTML documentation:

```
./run_ford.sh
```

   This will create a folder named `doc` (by default) containing an HTML overview of the Fortran code, including modules, functions, and subroutines.

4. Open `doc/index.html` in a web browser to explore the generated documentation.

Using `ford` is especially helpful for new contributors to quickly get an overview of how the program is structured and understand the relationships between modules. It generates images like the following, which is the program this manual talks about at the time when this manual was written:

4.3 General Structure

The structure of the dependencies in the `main` program reflects how this project was developed as an extension of Konstantin Klein's original program.

- **Top-left corner:** Modules responsible for updating the HEP. This part of the program has been left mostly untouched from Klein's original implementation.
- **Center:** Modules providing the infrastructure for the agent-based model. The core of this infrastructure consists of `mod_agent_class` and `mod_grid`, which define the agent class and the grid on which the agents move.
- **Left:** The `globals` module, containing all global parameters and variables used throughout the simulation.

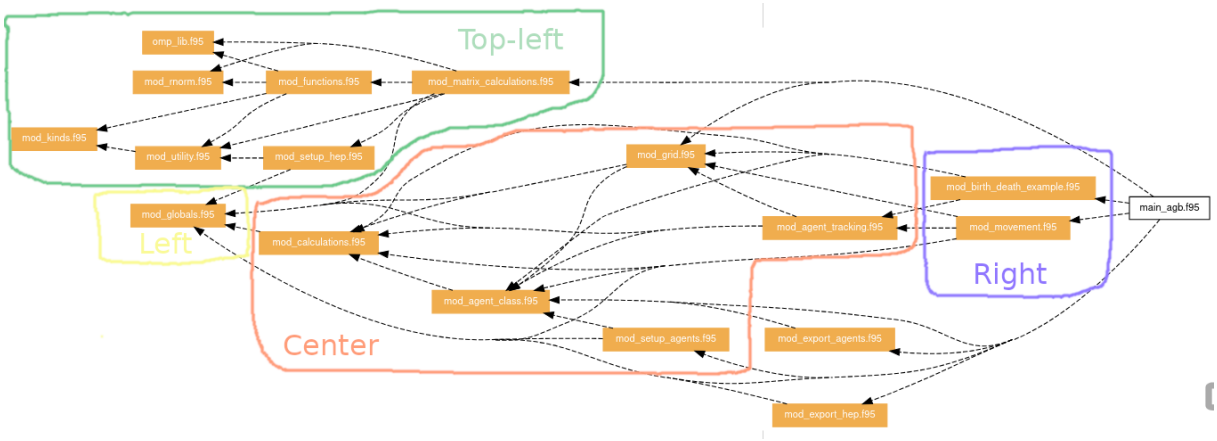


Figure 1: Dependencys of main.f95

- **Right, next to `main`:** Two example `simulation_modules`, which implement the actual simulation logic.

This layout illustrates both the preserved elements from Klein's program and the new structures introduced for the agent-based extensions.

4.4 Strucutre of the main

The actual simulation is implemented in `main`. It consists of a large loop over the simulation time steps (`t`). For each time step, different processes are executed that represent what happens during that step. For example, the movement of agents is handled individually: we loop over all agents to update their positions.
Since agents are organized into different populations, this loop is actually nested: an outer loop over populations and an inner loop over the agents in each population.
Some processes, such as agent death, are handled differently. Currently, agents die when too many individuals occupy a single grid cell. This does not require looping over all agents individually, but rather looping over the grid cells. For example, in the `death_example` function, there is a loop over all grid cells to remove agents if necessary.

5 Usage - new simulation modules

5.1 Command-line Interface

The main program that is compiled using *make all* can be run using one of the two:

```
./bin/main_agb
./run_all.sh main_agb
```

The second option automatically runs python scripts that generate a visualization of the movement of the agents. This is great to see wether the additions you made work as intended.

5.2 You first simulation module

You are now ready to write your first simulation module, that simulates one aspect of the agents life. How do you start?

1. Create a .f95 file and put it in the folder src/simualtion_modules
2. add it into the make file, so that it is compiled when you type make all.

3. write functions that mess with the agents.

4. include it in the main and run the program

1,2 and 4 are really straight forward. 3 can seem a little demanding in the beginning but once you understood how to acess agents this should be easy.

5.3 Accessing agents

In general, there are multiple ways to access agents, depending on which aspect of their behavior or state we want to simulate. The examples in section 4.4 illustrate this principle. The following provides an overview of the main access methods used in the program.

5.3.1 Pointers and Data Structures

To efficiently manage agents, we use a combination of a doubly linked list and several arrays of pointers to the objects in this list. These arrays allow us to "sort" the list logically and quickly access agents with certain properties. A pointer is essentially a variable that stores the memory address of another variable or object. Using pointers allows the program to access or modify an object from multiple places without creating copies. All agents are stored in a single location using a doubly linked list. During the simulation, references to these agents are organized into various matrices and arrays. These matrices and arrays allow the program to efficiently access specific agents as needed, without copying or moving the actual data in memory. This approach ensures that agent data remains in one place while still providing flexible access patterns throughout the simulation.

1. **Agent list:** A doubly linked list through which we can iterate from start to end, ensuring that every agent can be reached. Agents are appended to the end of the list when they are born and removed when they die. This access method is only used in rare cases due to its generality.

```
! Example: Code that increases age of all agents by one

type(Node), pointer :: current_agent
current_agent => head_agent
do while(associated(current_agent):
    current_agent%age = current_agent%age + 1
    current_agent => current_agent%next
enddo
```

2. **Agent population matrix:** A matrix that allows access to all agents within each population. Within a population, the matrix does not follow any specific sorting logic. The matrix is always larger than the number of agents in the population, so entries at the end of each population are set to `Null()`.

```
! Example: Population 3 is affected by a virus that changes
    gender of all men of that population


! defined and intilized somwhere in program:
! type(pointer_node), allocatable ::
   population_agents_matrix(:,:)
! integer :: num_humans_in_pop(3)

type(Node), pointer :: current_agent
```

```
integer :: population = 3

integer :: i = 0

do i = 1, num_humans_in_pop(population)
    current_agent => population_agents_matrix(i,population)
       %node

    if (current_agent%gender == 'M') then
        current_agent%gender = 'F'
    endif

enddo
```

3. **Agents array:** Similar to the agent list, but implemented as an array. Like the population matrix, the last entries are `Null()`. This structure is especially useful when agents need to be selected randomly, independent of population.

```
! Example: Random agent is struck by lightning
! Defined and initilized somewhere in the program:
! interger :: number_of_agents  ( = sum(num_humans_in_pop)
   )


! Generate a random number between 0 and 1
call random_number(r)

! Scale to integer between 1 and number_of_agents
rand_int = int(r * number_of_agents) + 1

call agents_array(rand_int)%agent_die()

! Remark: agent_die() kills agents but doesnt remvove it
   from grid. has to be removed from grid to avoid
   unexpected behavior.
```

4. **Grid access:** Each grid cell contains a doubly linked list of pointers to all agents currently located in that cell. This allows efficient processing of spatially-dependent events, such as agent interactions or death due to overcrowding.

```
! Example: The Black Death breaks out in the grid cell with
    the most agents in it. 30% of all agents in that grid
   cell die.

! Do this yourself as a little exercide.
```

These data structures together provide flexible and efficient ways to iterate over agents depending on the simulation task at hand.

6 Extending and Customizing

Describe the structure of the code (e.g., directory layout, main modules) and provide guidance on how contributors can extend or adapt the code.

7 Troubleshooting and FAQ

Common issues and their solutions.

8 References

Include papers, websites, or other resources relevant to the project.