Documentation of `ABM-Hescor`

Daniel Nogues
University of Cologne

October 28, 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), 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.

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

If you want to visualize the simulations using the python scripts make sure you have the following packages installed:

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

Instead of installing the packages on your local computer, it might be more convenient to install them on the virtual machine.

2.2 Setup

Once inside the project folder:

1. If you installed packages in virtual machine: 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_demo
```

This will run one simulation until you stop it or it reaches the simulation horizon. Data is stored in /data. Alternativly (requires python) you can run:

```
./run_all.sh main_demo
```

This will run the script for 30 seconds and then automatically generate a animation of the data stored in data/. This option has to be debugged. The python script has to be updated to include the new data added. (I think, DN 10.25Last but not least you can also run the simulation running the following python scipt:

```
python3 python/live_visualize_demo.py
```

This will open a python UI that lets you

- start/stop/restart the simulation
- visualize the data generated while running the simulation
- set the update rate at which the data is updated (for performance)

The first two options are usually the preffered way to run the simulation when you are developing a part of the simulation. The third option is usually the preferred way once you are done developing the simulation and you are calibrating the parameters that feed the model.

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
```

Or the Program takes a cell of the grid on which the agents move as input:

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

    integer :: excess_population

    excess_population = cell_ptr%number_of_agents -
       max_agents_per_cell

    if (excess_population <= 0) then
        ! Nothing to be done
        return
    endif

    call kill_n_agents_in_cell(cell_ptr)

end subroutine agent_move
```

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

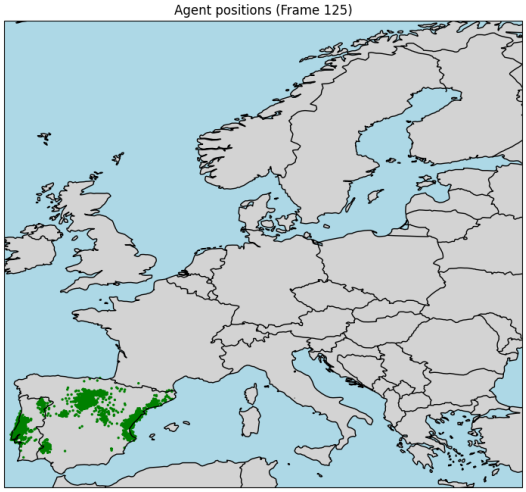
```
do t = 1, TN

    call apply_module_to_agents(agent_move)

    call apply_modules_to_agents(agents_reproduce)
        ...

    call apply_modules_to_cells(death_overpopulation)
        ...

enddo
```

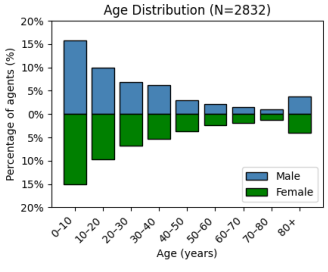
I think that once this is achieved many people can work simultaniously on the actual simulation. And their work can easily be integrated.
One Simulation module would be represented by (ideally one) subroutine(s).
Also this structure makes it very easy to see what parameters the different simulation modules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

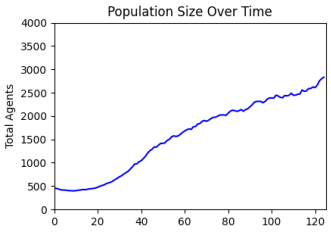
3.1.2 Visualisation

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:

- visualization of the hep
- ...

3.2 Ruben

3.3 Y. Shao

4 Program Structure

This chapter presents the structure of the program. Generally speaking we can sort the goals of the users of this source code into two groups:

- develop a module using the existing infrastructure of the program
- expand the infrastructure of the program to
  - provide new functionality

– connect it to other models ...

If your goal is to develop a module using the existing infrastructure, then you will only have to write code in very few parts of the program. In this chapter the parts in which you will write code are green. The parts that you will not have to touch are marked red.

4.1 General File 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.

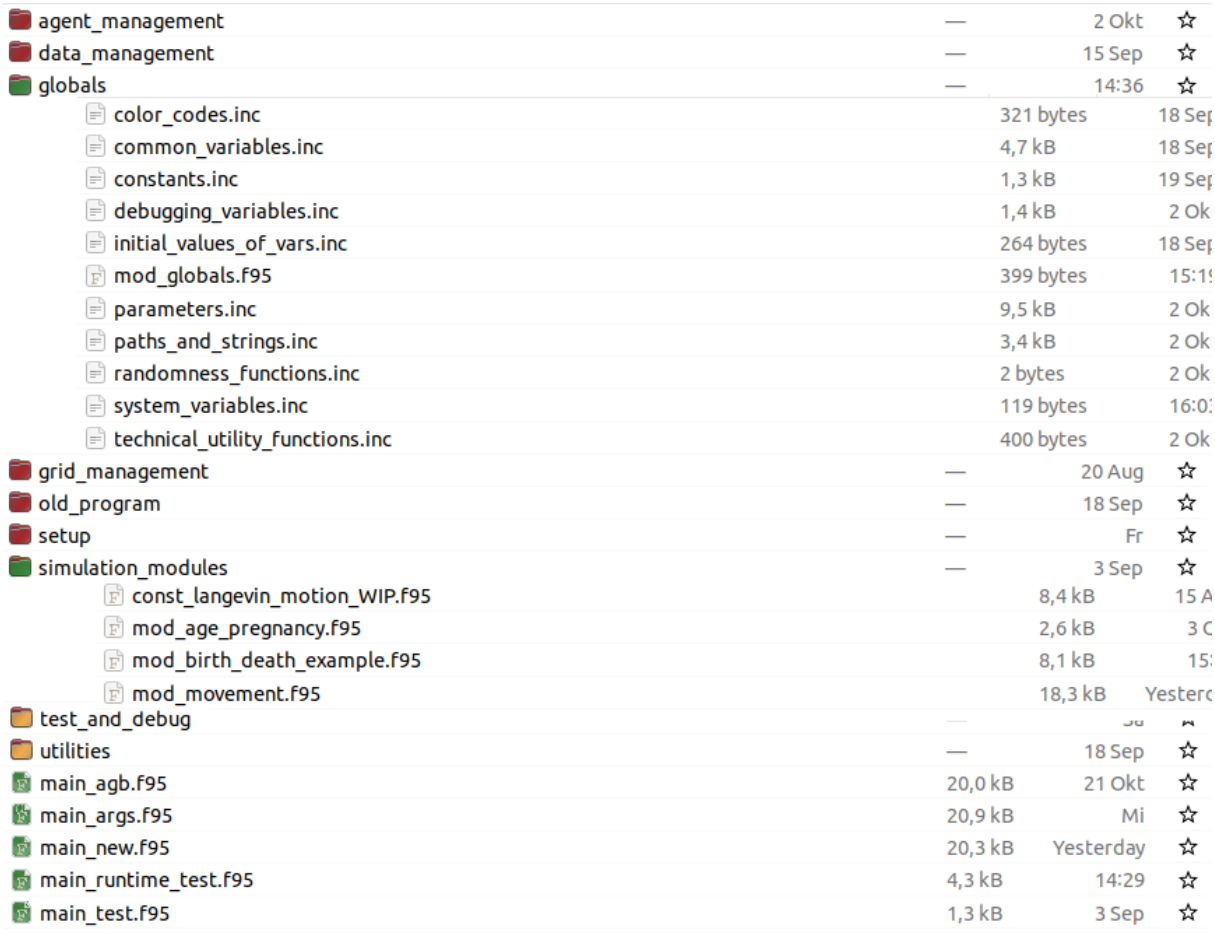


Figure 1: File Structure of src

4.2 Simplified structure of the program

Before trying to understand the whole sourcecode, let us concentrate on how one simulation is created. Let us think of one simulation as an experiment. The experiment is we generate a bunch of neanderthals and then observe what they do.
The main function that is executed when we run the simulation is our experiment: our main iterates over time and makes the neanderthalers do stuff. What they can do is defined in seperate files in the folder: **simulation_modules**/.
If we now want to design a new experiment, where the neanderthals do something different or do something they have not done so far, we thus need a new *main* function. A good starting point to implement such a main function is to just copy the mainfunction of an experiment that we know already works well.
Each main function / experiment will depend on the other parts of the source code. If we only want to change the behaviour of the neanderthals and do not want to expand the programs functionality we will only have to work with the greendependencies, as illustrated by the following picture:

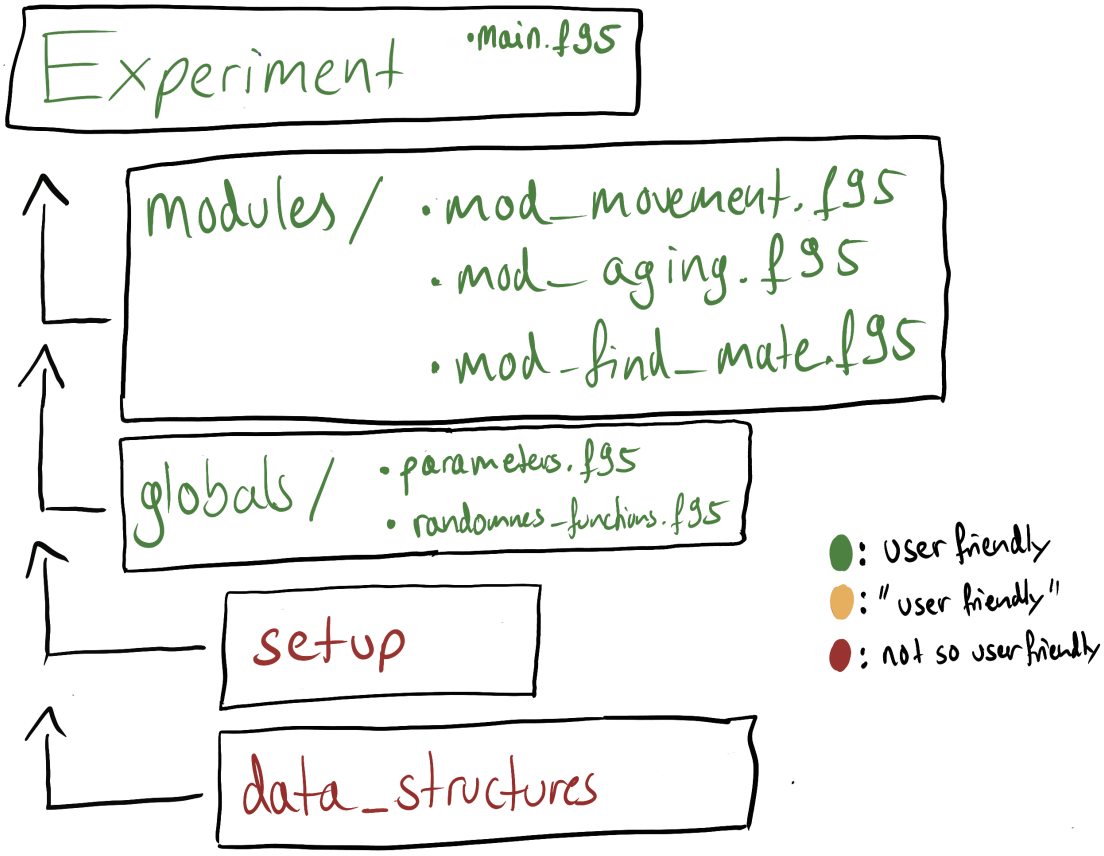


Figure 2: Simplified dependencies of one experiment.

4.3 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.

4.4 Structure of the main

The apply modules to cells function doesnt exist yet. The grid data structure still needs some finetuning.

The actual simulation is implemented in `main`. It consists of a large loop over the simulation time steps (`t`). For each time step then the modules are applied:

```
! Main:
        ! Setup

        call setup_hep()

        call setup_agents()

        ! Test for correctness:

        call run_tests()

        ! Beginning of Simulation
```

```
do t = 1, TN

    ! apply modules to agents

    call apply_module_to_agents(agent_move)

    call apply_modules_to_agents(agents_reproduce)
        ...

    ! apply modules to grid

    call apply_modules_to_cells(death_overpopulation)
        ...

    ! test during simulation

    call run_control_tests()

enddo

! save the data:

call save_data()
```

There are and there will be simulation modules that are programmed more efficiently when they access the agents and others when the access the grid. For example, the movement of agents is handled individually: we loop over all agents to update their positions. Thus the simulation module for the movement takes a agent as 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
```

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. Thus the current death module requires a cell as input:

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

    integer :: excess_population

    excess_population = cell_ptr%number_of_agents -
       max_agents_per_cell
```

```
        if (excess_population <= 0) then
            ! Nothing to be done
            return
        endif

        call kill_n_agents_in_cell(cell_ptr)

    end subroutine agent_move
```

4.5 Overall structure of the program

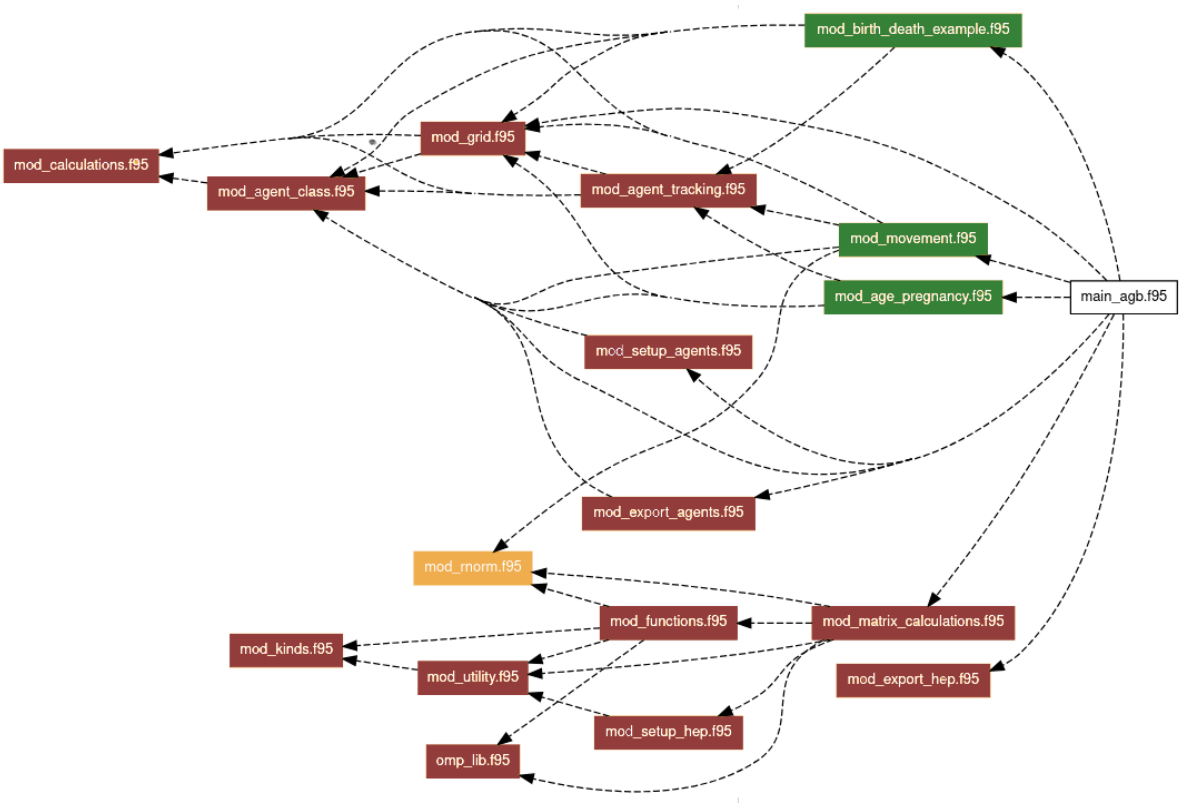


Figure 3: Dependencys of main.f95

The structure of the dependencies in the `main` program reflects how this project was developed as an extension of Konstantin Klein's original program. If you look at figure 4 you can devide the dependencies of *main_ agb.f95* roughly in two groups:

- one group in the top where all dependencies end in *mod_ calculations*
- and one group in the bottom where all dependencies end in *mod_ kinds.f95*.

The group in the bottom corresponds to Konstantin Klein's original program. Let us group the code into even more groups. This time not by who coded them but by their function:

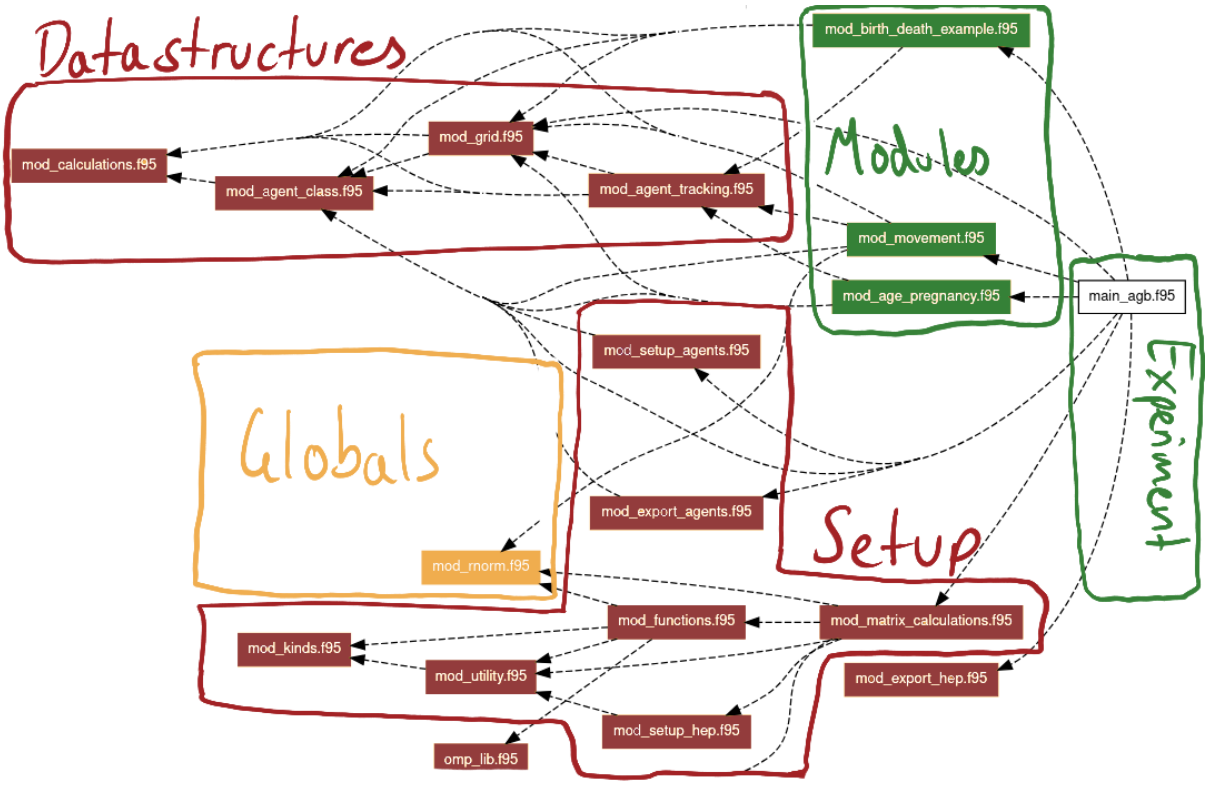


Figure 4: Dependencys of main.f95 annotated.

4.5.1 Generating Documentation with `ford`

The illustrations of the file dependencies in this chapter where generated using `ford`. This is a python tool that automatically generates fortran documentation. In this manual you only find the generated images. If you create the documentation yourself in your local branch you can extract even more information from it by browsing the html files created by ford. To create these do the following:

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:

5 Usage - new simulation modules

5.1 Command-line Interface

The main program that is compiled using *make all* can be run using

```
./bin/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 or with the grid
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 or the grid?

As explained in section 4.4 there are two types of modules and you should think about which one you are going to develop before you start coding. You have to decide if:

- You want to do something for each agent.
- You want to do something for each gridcell.

Once you have made that decision you start by writing a header for your simulation module. We assume for now that you have decided to *"Do somethin for each agent."* because it is important to understand the agent structre before you can fully understand the grid structure. The header of your function should look something like this:

```
subroutine my_module(agent_ptr)
    implicit none
    type(Node), pointer, intent(inout) :: agent_ptr

    ...

end subroutine my_module
```

As you can see the your function gets a pointer to a object of type Node which is called agent. If you are familiar with pointers then you can skip the next section.

5.3.1 Pointers and Data Structures

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.

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.