Package 'villager'

May 8, 2026

Title A Framework for Designing and Running Agent Based Models

Version 2.0.0

Description This is a package for creating and running Agent Based Models (ABM). It provides a set of base classes with core functionality to allow bootstrapped models. For more intensive modeling, the supplied classes can be extended to fit researcher needs.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.1

Depends R (>= 3.5.0)

Imports readr, R6, uuid

Suggests covr, dplyr, knitr, leaflet, plotly, remotes, rmarkdown, testthat, roxygen2, pandoc,

URL https://github.com/zizroc/villager/

BugReports https://github.com/zizroc/villager/issues/

VignetteBuilder knitr

NeedsCompilation no

Author Thomas Thelen [aut, cre],
Gerardo Aldana [aut],
Marcus Thomson [aut],
Toni Gonzalez [aut]

Maintainer Thomas Thelen <tommythelen@gmail.com>

Repository CRAN

Date/Publication 2024-05-25 15:20:03 UTC

Contents

`agent` *agent*

Description

This is an object that represents a villager (agent).

Details

This class acts as an abstraction for handling villager-level logic. It can take a number of functions that run at each timestep. It also has an associated

Methods

`as_table()` Represents the current state of the agent as a tibble

`get_age()` Returns age in terms of years

`get_gender()`

`get_days_sincelast_birth()` Get the number of days since the agent last gave birth

`initialize()` Create a new agent

`propagate()` Runs every day

Create a new agent

Public fields

`identifier` A unique identifier that can be used to identify and find the agent

`first_name` The agent's first name

`last_name` The agent's last name

`age` The agent's age

`mother_id` The identifier of the agent's mother

`father_id` The identifier of the agent's father

`profession` The agent's profession

`partner` The identifier of the agent's partner

`gender` The agent's gender

`alive` A boolean flag that represents whether the villager is alive or dead

`children` A list of children identifiers

`health` A percentage value of the agent's current health

Methods

Public methods:

- agent$new()
- agent$is_alive()
- agent$get_days_since_last_birth()
- agent$add_child()
- agent$as_table()
- agent$clone()

Method `new()`**:** Used to created new agent objects.

Usage:

```
agent$new(
  identifier = NA,
  first_name = NA,
  last_name = NA,
  age = 0,
  mother_id = NA,
  father_id = NA,
  partner = NA,
  children = vector(mode = "character"),
  gender = NA,
  profession = NA,
  alive = TRUE,
  health = 100
)
```

Arguments:

`identifier` The agent's identifier

`first_name` The agent's first name

`last_name` The agent's last name

`age` The age of the agent

`mother_id` The identifier of the agent's mother

`father_id` The identifier of the agent' father

`partner` The identifier of the agent's partner

`children` An ordered list of of the children from this agent

`gender` The gender of the agent

`profession` The agent's profession

`alive` Boolean whether the agent is alive or not

`health` A percentage value of the agent's current health

Returns: A new agent object A function that returns true or false whether the villager dies This is run each day

Method `is_alive()`**:**

Usage:

```
agent$is_alive()
```

Returns: A boolean whether the agent is alive (true for yes) Gets the number of days from the last birth. This is also the age of the most recently born agent

Method `get_days_since_last_birth()`**:**

Usage:

```
agent$get_days_since_last_birth()
```

Returns: The number of days since last birth Connects a child to the agent. This method ensures that the 'children' vector is ordered.

Method `add_child()`**:**

Usage:

```
agent$add_child(child)
```

Arguments:

`child` The agent object representing the child

Returns: None Returns a data.frame representation of the agent

Method `as_table()`**:** I hope there's a more scalable way to do this in R; Adding every new attribute to this function isn't practical

Usage:

```
agent$as_table()
```

Details: The village_state holds a copy of all of the villagers at each timestep; this method is used to turn the agent properties into the object inserted in the village_state.

Returns: A data.frame representation of the agent

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
agent$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`agent_manager` *agent Manager*

Description

A class that abstracts the management of aggregations of agent classes. Each village should have an instance of a agent_manager to interface the agents inside.

Methods

`add_agent()` Adds a single agent to the manager.

`get_average_age()` Returns the average age, in years, of all the agents.

`get_living_agents()` Gets a list of all the agents that are currently alive.

`get_states()` Returns a data.frame consisting of all of the managed agents.

`get_agent()` Retrieves a particular agent from the manager.

`get_agent_index()` Retrieves the index of a agent.

`initialize()` Creates a new manager instance.

`load()` Loads a csv file defining a population of agents and places them in the manager.

`remove_agent()` Removes a agent from the manager

Creates a new agent manager instance.

Public fields

`agents` A list of agents objects that the agent manager manages.

`agent_class` A class describing agents. This is usually the default villager supplied 'agent' class

Methods

Public methods:

- `agent_manager$new()`
- `agent_manager$get_agent()`
- `agent_manager$get_living_agents()`
- `agent_manager$add_agent()`
- `agent_manager$remove_agent()`
- `agent_manager$get_states()`
- `agent_manager$get_agent_index()`
- `agent_manager$connect_agents()`
- `agent_manager$get_living_population()`
- `agent_manager$get_average_age()`
- `agent_manager$add_children()`
- `agent_manager$load()`
- `agent_manager$clone()`

Method `new()`:

Usage:

```
agent_manager$new(agent_class = villager::agent)
```

Arguments:

`agent_class` The class that's being used to represent agents being managed Given the identifier of a agent, sort through all of the managed agents and return it if it exists.

Method `get_agent()`**:** Return the R6 instance of a agent with identifier 'agent_identifier'.

Usage:

```
agent_manager$get_agent(agent_identifier)
```

Arguments:

`agent_identifier` The identifier of the requested agent.

Returns: An R6 agent object Returns a list of all the agents that are currently alive.

Method `get_living_agents()`**:**

Usage:

```
agent_manager$get_living_agents()
```

Returns: A list of living agents Adds a agent to the manager.

Method `add_agent()`**:**

Usage:

```
agent_manager$add_agent(...)
```

Arguments:

`...` One or more agents

Returns: None Removes a agent from the manager

Method `remove_agent()`**:**

Usage:

```
agent_manager$remove_agent(agent_identifier)
```

Arguments:

`agent_identifier` The identifier of the agent being removed

Returns: None Returns a data.frame of agents

Method `get_states()`**:**

Usage:

```
agent_manager$get_states()
```

Details: Each row of the data.frame represents a agent object

Returns: A single data.frame of all agents Returns the index of a agent in the internal agent list

Method `get_agent_index()`**:**

Usage:

```
agent_manager$get_agent_index(agent_identifier)
```

Arguments:

`agent_identifier` The identifier of the agent being located

Returns: The index in the list, or R's default return value Connects two agents together as mates

Method `connect_agents()`**:**

Usage:

```
agent_manager$connect_agents(agent_a, agent_b)
```

Arguments:

`agent_a` A agent that will be connected to agent_b

`agent_b` A agent that will be connected to agent_a Returns the total number of agents that are alive

Method `get_living_population()`**:**

Usage:

```
agent_manager$get_living_population()
```

Returns: The number of living agents Returns the average age, in years, of all of the agents

Method `get_average_age()`**:**

Usage:

```
agent_manager$get_average_age()
```

Details: This is an *example* of the kind of logic that the manager might handle. In this case, the manager is performing calculations about its aggregation (agents). Note that the 364 days needs to work better

Returns: The average age in years Takes all of the agents in the manager and reconstructs the children

Method `add_children()`**:**

Usage:

```
agent_manager$add_children()
```

Details: This is typically called when loading agents from disk for the first time. When children are created during the simulation, the family connections are made through the agent class and added to the manager via add_agent.

Returns: None Loads agents from disk.

Method `load()`**:**

Usage:

```
agent_manager$load(file_name)
```

Arguments:

`file_name` The location of the file holding the agents.

Details: Populates the agent manager with a set of agents defined in a csv file.

Returns: None

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
agent_manager$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`data_writer` *Data Writer*

Description

A class responsible for the simulation data to disk.

Details

This class can be subclasses to provide advanced data writing to other data sources. This should also be subclassed if the agent and resource classes are subclasses, to write any additional fields to the data source.

Methods

`write()` Writes the agent and resources to disk.

Create a new data writer.

Public fields

`results_directory` The folder where the simulation results are written to

`agent_filename` The location where the agents are written to

`resource_filename` The location where the resources are written to

Methods

Public methods:

- `data_writer$new()`
- `data_writer$write()`
- `data_writer$clone()`

Method `new()`**:** Creates a new data writer object that has optional paths for data files.

Usage:

```
data_writer$new(
  results_directory = "results",
  agent_filename = "agents.csv",
  resource_filename = "resources.csv"
)
```

Arguments:

`results_directory` The directory where the results file is written to

`agent_filename` The name of the file for the agent data

`resource_filename` The name of the file for the resource data

Returns: A new agent object Writes a village's state to disk.

Method `write()`**:** Takes a state an the name of a village and writes the agents and resources to disk

Usage:

```
data_writer$write(state, village_name)
```

Arguments:

`state` The village's village_state that's being written

`village_name` The name of the village. This is used to create the data directory

Returns: None

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
data_writer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`model_data` *Model Data*

Description

R6 Class representing data that's external from resources and agents

It contains a single variable, 'events' for when the data holds a list of events

Public fields

`events` Any events that need to be tracked

Methods

Public methods:

- `model_data$new()`
- `model_data$clone()`

Method `new()`**:** Creates a new model_data object

Usage:

```
model_data$new()
```

Returns: A new model data object

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
model_data$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`resource` *Resource*

Description

This is an object that represents a single resource.

Methods

`initialize()` Create a new resource

`as_table()` Represents the current state of the resource as a tibble

Creates a new resource.

Public fields

`name` The name of the resource

`quantity` The quantity of the resource that exists

Methods

Public methods:

- `resource$new()`
- `resource$as_table()`
- `resource$clone()`

Method `new()`**:** Creates a new resource object

Usage:

```
resource$new(name = NA, quantity = 0)
```

Arguments:

`name` The name of the resource

`quantity` The quantity present Returns a data.frame representation of the resource

Method `as_table()`**:**

Usage:

```
resource$as_table()
```

Returns: A data.frame of resources

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
resource$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

resource_manager *Resource Manager*

Description

This object manages all of the resources in a village.

Methods

`initialize()` Creates a new manager

`get_resources()` Gets all of the resources that the manager has

`get_resource()` Retrieves a resource from the manager

`add_resource()` Adds a resource to the manager

`remove_resource()` Removes a resource from the manager

`get_resource_index()` Retrieves the index of the resource

`get_states()` Returns a list of states

`load()` Loads a csv file of resources and adds them to the manager.

Public fields

`resources` A list of resource objects

`resource_class` The class used to represent resources Creates a new , empty, resource manager for a village.

Methods

Public methods:

- `resource_manager$new()`
- `resource_manager$get_resources()`
- `resource_manager$get_resource()`
- `resource_manager$add_resource()`
- `resource_manager$remove_resource()`
- `resource_manager$get_resource_index()`
- `resource_manager$get_states()`
- `resource_manager$load()`
- `resource_manager$clone()`

Method `new()`**:** Get a new instance of a resource_manager

Usage:

```
resource_manager$new(resource_class = villager::resource)
```

Arguments:

`resource_class` The class being used to describe the resources being managed Gets all of the managed resources

Method `get_resources()`**:**

Usage:

```
resource_manager$get_resources()
```

Returns: A list of resources Gets a resource given a resource name

Method `get_resource()`**:**

Usage:

```
resource_manager$get_resource(name)
```

Arguments:

`name` The name of the requested resource

Returns: A resource object Adds a resource to the manager.

Method `add_resource()`**:**

Usage:

```
resource_manager$add_resource(...)
```

Arguments:

`...` The resources to add

Returns: None Removes a resource from the manager

Method `remove_resource()`**:**

Usage:

```
resource_manager$remove_resource(name)
```

Arguments:

`name` The name of the resource being removed

Returns: None Returns the index of a resource in the internal resource list

Method `get_resource_index()`**:**

Usage:

```
resource_manager$get_resource_index(name)
```

Arguments:

`name` The name of the resource being located

Returns: The index in the list, or R's default return value Returns a data.frame where each row is a resource.

Method `get_states()`**:**

Usage:

```
resource_manager$get_states()
```

Details: Subclasses should not have to override this method because it takes all member variables into account

Returns: A single data.frame Loads a csv file of resources into the manager

Method `load()`**:**

Usage:

```
resource_manager$load(file_name)
```

Arguments:

`file_name` The path to the csv file

Returns: None

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
resource_manager$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`simulation` *simulation*

Description

Advances one or more villages through time

Methods

`run_model()` Runs the simulation

Creates a new Simulation instance

Public fields

`length` The total number of time steps that the simulation runs for

`villages` A list of villages that the simulator will run

`writer` An instance of a data_writer class for writing village data to disk

Methods

Public methods:

- `simulation$new()`
- `simulation$run_model()`
- `simulation$clone()`

Method `new()`**:** Creates a new simulation object to control the experiment

Usage:

```
simulation$new(length, villages, writer = villager::data_writer$new())
```

Arguments:

`length` The number of steps the simulation takes

`villages` A list of villages that will be simulated

`writer` The data writer to be used with the villages Runs the simulation

Method `run_model()`**:**

Usage:

```
simulation$run_model()
```

Returns: None

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
simulation$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`village` *Village*

Description

This is an object that represents the state of a village at a particular time.

Details

This class acts as a type of record that holds the values of the different village variables. This class can be subclassed to include more variables that aren't present.

Methods

`initialize()` Creates a new village

`propagate()` Advances the village a single time step

`set_initial_state()` Initializes the initial state of the village

Public fields

`name` An optional name for the village

`initial_condition` A function that sets the initial state of the village

`current_state` The village's current state

`previous_state` The village's previous state

`models` A list of functions or a single function that should be run at each timestep

`model_data` Optional data that models may need

`agent_mgr` The manager that handles all of the agents

`resource_mgr` The manager that handles all of the resources Initializes a village

Methods

Public methods:

- village$new()
- village$propagate()
- village$set_initial_state()
- village$clone()

Method `new()`**:** This method is meant to set the variables that are needed for a village to propagate through time.

Usage:

```
village$new(
  name,
  initial_condition,
  models = list(),
  agent_class = villager::agent,
  resource_class = villager::resource
)
```

Arguments:

`name` An optional name for the village

`initial_condition` A function that gets called on the first time step

`models` A list of functions or a single function that should be run at each time step

`agent_class` The class that's being used to represent agents

`resource_class` The class being used to describe the resources Propagates the village a single time step

Method `propagate()`**:**

Usage:

```
village$propagate(current_step)
```

Arguments:

`current_step` The current time step

Details: This method is used to advance the village a single time step. It should NOT be used to set initial conditions. See the set_initial_state method.

Returns: None Runs the user defined function that sets the initial state of the village

Method `set_initial_state()`**:** Runs the initial condition model

Usage:

```
village$set_initial_state()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
village$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`village_state` *village_state*

Description

This is an object that represents the state of a village at a particular time.

Details

This class acts as a type of record that holds the values of the different village variables. This class can be subclassed to include more variables that aren't present.

Methods

Creates a new State

Public fields

`step` The time step that the state is relevant to

`agent_states` A list of agent states

`resource_states` A list of resource states

Methods

Public methods:

- `village_state$new()`
- `village_state$clone()`

Method `new()`**:** Initializes all of the properties in the state to the ones passed in. This should be called by subclasses during initialization.

Usage:

```
village_state$new(
  step = 0,
  agent_states = vector(),
  resource_states = vector()
)
```

Arguments:

`step` The time step that the state is relevant to

`agent_states` A vector of tibbles representing the states of the agents

`resource_states` A vector of tibbles representing the states of the resources

Details: When adding a new property, make sure to add it to the tibble representation.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
village_state$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Index