Kami

Release main

James P. Howard, II <james.howard@jhu.edu>

Feb 02, 2023

CONTENTS

Kami is Agent-Based Modeling in Modern C++.

CHAPTER

ONE

OVERVIEW

Agent-based models (ABMs) are models for simulating the actions of individual actors within a provided environment to understand the behavior of the agents, most individually and collectively. ABMs are particularly suited for addressing problems governed by nonlinear processes or where there is a wide variety of potential responses an individual agent may provide depending on the environment and behavior of other agents. Because of this, ABMs have become powerful tools in both simulation and modeling, especially in public health and ecology, where they are also known as individual-based models. ABMs also provide support in economic, business, robotics, and many other fields.

1.1 Design Objectives

Kami provides agent-based modeling modern C++. The objectives in writing Kami are that it be lightweight, memory-efficient, and fast. It should be possible to develop a simple working model in under one hour of C++ development time. Accordingly, the platform is modeled on the Mesa library in Python, which itself was inspired by the MASON library in Java.

Many ABM platforms are designed around interaction and real time observation of the agent dynamics. Kami does not provide a visualization interface. Instead, Kami is meant to be used for ABMs requiring many runs with different starting conditions. Accordingly, Kami is single-threaded and multiple cores should be taken advantage of through multiple parallel runs of the supervising model.

1.1.1 Installation

Requirements

The core of Kami, `libkami`, has no requirements beyond a modern C++ compiler and `neargye-semver/0.3.0`. However, both the examples provided and the unit tests provided rely on three additional C++ packages. The full list is:

- cli11/1.9.1
- gtest/cci.20210126
- neargye-semver/0.3.0”
- spdlog/1.8.5
- fmt/7.1.3

`Google Test` provides a unit testing framework. `CLI11` provides a command line interface for each of the utilities that makeup the examples. `spdlog` provides a uniform output interface. Coupled with a command line option to set the output level, `spdlog` allows the unit tests and example programs to provide variable output levels depending on the users needs. Finally, `fmt` is required by `spdlog` for simple and easy string formatting.

Compiling

To compile and test locally in kami/build:

```
git clone https://github.com/k3jph/kami.git
cd kami
conan install -if build .
cmake -B build -DBUILD_SHARED_LIBS:BOOL=FALSE
cmake --build build
cmake --build build --target test
```

Conan Installation (Local)

To install via Conan:

```
conan create . kami/develop
```

1.1.2 About Agent-Based Models

Agent-based models (ABM) are a type of computational model used to simulate the behavior of autonomous agents within a system. These agents can be individuals, groups, organizations, or other entities that interact with one another and with their environment.

One of the key features of ABMs is that they focus on the micro-level interactions between individual agents, rather than aggregating data to study macro-level phenomena. This allows for the examination of complex behaviors that emerge from the interactions between agents, such as the spread of a disease or the formation of social networks.

ABMs are often used in fields such as economics, sociology, and biology to study the behavior of individuals and groups. They can also be used to simulate the effects of different policies or interventions on a system.

In order to create an ABM, the researcher must first define the agents and their characteristics, such as their behavior, beliefs, and goals. They must also define the rules of interaction between the agents and their environment. Once these parameters are set, the model can be run to simulate the behavior of the agents over time.

ABMs are a powerful tool for understanding complex systems, but they also have their limitations. Because they focus on micro-level interactions, they may not accurately capture macro-level phenomena. Additionally, they often require a significant amount of computational resources and can be difficult to validate.

Overall, agent-based models are a valuable tool for understanding the behavior of complex systems and the emergence of complex behaviors from the interactions between individuals. However, it is important to use them in conjunction with other methods to fully understand the system being studied.

1.1.3 Tutorial

Kami's interface is heavily influenced by Mesa's interface. However, by being written in C++, Kami runs substantially faster. This allows for faster runs and more runs within a fixed amount of time. The advantage here is that an agent-based model (ABM) built on the Kami platform is better suited for statistical and Monte Carlo approaches to modeling.

Model Form

Kami-based models have five key components:

1. Agents, which are objects representing the actors within the model
2. Populations, which are collections of Agents
3. Domains, which provide a representation of "physical" space the Agent inhabits
4. Schedulers, which provide a representation of "time" within the model
5. Model, which are objects connecting Populations, Domains, and Schedulers

In general, a model should have one scheduler, one domain, and some number of agents. However, it would not be impossible to have more than one scheduler or more than one domain. Because this is implemented in C++, your agents should subclass Agent and your model should subclass model. The schedulers and domains are sufficient as is for their purposes though custom schedulers and domains are not unreasonable.

A Minimal ABM

The minimal ABM starts with the simplest possible agent. Here, we create a class called `MinimalAgent`:

```
class MinimalAgent : public kami::Agent {
public:
  kami::AgentID step(std::shared_ptr<kami::Model> model) override {
    return this->get_agent_id();
  }
};
```

An `Agent`, and its subclasses, will automatically inherit an `AgentID`, which is the unique identifier for the session. The only explicit requirement on the `Agent` subclass is a *step()* method that accepts a `shared_ptr` to a `Model` and it must return the `Agent`'s `AgentID`. Obviously, an `Agent` should do something useful before returning.

The second component is `MinimalModel:`

```
class MinimalModel: public kami::Model {
public:
  MinimalModel() {
    auto sched = std::make_shared<kami::SequentialScheduler>();
    set_scheduler(sched);

    auto pop = std::make_shared<kami::Population>();
    set_population(pop);

    for (auto i = 0; i < 10; i++) {
      auto new_agent = std::make_shared<MinimalAgent>();
      pop->add_agent(new_agent);
    }
  }
};
```

The `MinimalModel` performs some important tasks that important to do during the setup or soon thereafter. In the constructor, first, a scheduler is created. The `SequentialScheduler` is the simplest scheduler and has no configuration needed. Using *set_scheduler()*, part of the Model class, the scheduler is associated with this model. Second, a *Population* is created and associated with this model with the *set_population()* method.

After this, the constructor initializes 10 `MinimalAgents` and adds them to the population.

```cpp
int main() {
  auto model = std::make_shared<MinimalModel>();

  for (int i = 0; i < 10; i++)
    model->step();

  return 0;
}
```

The last part is our *main()* function. It creates the *MinimalModel* then executes its *step()* method 10 times. The *step()* method, by default, calls the *step()* method of the scheduler. In the case of the `SequentialScheduler`, it loops over all the `Agent` instances in the `Population` and executes the associated *step()* method of each `Agent`.

That is it. It is the simplest minimal model that can be created using the Kami platform. However, for a basis, it is likely better to use the starter model, included in the examples directory.

1.1.4 Library API

Class Hierarchy

File Hierarchy

Full API

Namespaces

Namespace kami

Contents

Namespaces

- *Namespace kami::error*

Classes

- *Class Agent*
- *Class AgentID*
- *Class Constants*
- *Class Coord*
- *Class Domain*
- *Class Grid1D*
- *Class Grid2D*
- *Class GridCoord*
- *Class GridCoord1D*
- *Class GridCoord2D*
- *Class GridDomain*
- *Class Model*
- *Class MultiGrid1D*
- *Class MultiGrid2D*
- *Class Population*
- *Class RandomScheduler*
- *Class Reporter*
- *Class ReporterAgent*
- *Class ReporterModel*
- *Class Scheduler*
- *Class SequentialScheduler*
- *Class SoloGrid1D*
- *Class SoloGrid2D*
- *Class StagedAgent*
- *Class StagedScheduler*

Enums

- *Enum GridDistanceType*
- *Enum GridNeighborhoodType*

Functions

- *Function kami::get_version*

Typedefs

- *Typedef kami::Position*

Namespace kami::error

Contents

Classes

- *Class AgentNotFound*
- *Class LocationInvalid*
- *Class LocationUnavailable*
- *Class OptionInvalid*
- *Class ResourceNotAvailable*

Classes and Structs

Class Agent

- Defined in file_include_kami_agent.h

Inheritance Relationships

Derived Types

- `public kami::ReporterAgent` (*Class ReporterAgent*)
- `public kami::StagedAgent` (*Class StagedAgent*)

Class Documentation

class **Agent**

A superclass for all agents.

All agents should subclass the `Agent` class. At a minimum, subclasses must implement the `step()` function, to execute a single time step for each agent.

See also:

`ReporterAgent`, `StagedAgent`

Subclassed by *kami::ReporterAgent*, *kami::StagedAgent*

Public Functions

AgentID **get_agent_id**() const

Get the `Agent`'s `AgentID`.

Returns the `AgentID`

virtual *AgentID* **step**(std::shared_ptr<*Model*> model) = 0

Execute a time-step for the agent.

This function should step the agent instance. Any activities that the agent should perform as part of its time step should be in this function.

Parameters **model** – a reference copy of the model

Returns a copy of the *AgentID*

Friends

friend bool **operator==**(const *Agent* &lhs, const *Agent* &rhs)

Compare if two `Agent`s are the same `Agent`.

Subclasses of *Agent* may chose to extend this operator to tighten the restrictions on the comparison.

Note: This does not compare that two *Agent* instances are identical. Accordingly, this can be used to compare two instances of the same *Agent* at different points in its time stream.

Parameters

- **lhs** – is the left-hand side of the equality test.
- **rhs** – is the right-hand side of the equality test.

Returns true is they are equal and false if not.

friend bool **operator!=**(const *Agent* &lhs, const *Agent* &rhs)

Compare if two `Agent`s are not the same `Agent`.

Subclasses of `Agent` may chose to extend this operator to tighten the restrictions on the comparison.

Note: This does not compare that two *Agent* instances are identical. Accordingly, this can be used to compare two instances of the same `Agent` at different points in its time stream.

Parameters

- **lhs** – is the left-hand side of the equality test.
- **rhs** – is the right-hand side of the equality test.

Returns true is they are not equal and false if they are.

Class AgentID

- Defined in file_include_kami_agent.h

Class Documentation

class **AgentID**

A unique identifier for each `Agent`.

The unique identifier permits ordering to allow `AgentID`s to be used as keys for `std::map`. The unique identifier is unique for the session, however, `AgentID`s are not guaranteed to be unique from session-to-session.

See also:

Agent

Public Functions

AgentID()

Constructs a new unique identifier.

std::string **to_string**() const

Convert the identifier to a human-readable string.

Returns a human-readable form of the `AgentID` as `std::string`.

Friends

friend bool **operator==**(const *AgentID* &lhs, const *AgentID* &rhs)

Test if two `AgentID` instances are equal.

Parameters

- **lhs** – is the left-hand side of the equality test.
- **rhs** – is the right-hand side of the equality test.

Returns true is they are equal and false if not.

friend bool **operator!=**(const *AgentID* &lhs, const *AgentID* &rhs)

Test if two `AgentID` instances are not equal.

Parameters

- **lhs** – is the left-hand side of the equality test.
- **rhs** – is the right-hand side of the equality test.

Returns true is they are not equal and false if they are.

friend bool **operator<**(const *AgentID* &lhs, const *AgentID* &rhs)

Test if one *AgentID* is less than another.

Due to the way *AgentID* instances are used internally, the *AgentID* must be orderable. The < operator provides a basic ordering sufficient for `std::map`.

Parameters

- **lhs** – is the left-hand side of the ordering test.
- **rhs** – is the right-hand side of the ordering test.

Returns true if `lhs` is “less than” `rhs` as determined by the underlying implementation of the `AgentID`.

friend std::ostream &**operator<<**(std::ostream &lhs, const *AgentID* &rhs)

Output an *AgentID* to the specified output stream.

The form of the output will be the same as that produced by the `to_string()` member function.

Parameters

- **lhs** – is the stream to output the `AgentID` to
- **rhs** – is the `AgentID` to output

Returns the output stream for reuse

Class Constants

- Defined in file_include_kami_kami.h

Class Documentation

class **Constants**

A catalog of handy constants, mostly useful for seeding a random number generator.

Public Static Attributes

static constexpr auto **ADAMS_CONSTANT** = 42u

Life, the Universe, and Everything!

static constexpr auto **JENNYS_NUMBER** = 8675309u

Jenny, I've got your number.

static constexpr auto **JENNYS_CONSTANT** = 867.530901981

$(7^(e - 1/e) - 9) * pi^2$

Class Coord

- Defined in file_include_kami_domain.h

Inheritance Relationships

Derived Type

- `public kami::GridCoord` (*Class GridCoord*)

Class Documentation

class **Coord**

Provides a coordinate system for each *`Domain`*.

The coordinate system must be able to produce a human-readable version of the coordinates given. For instance, an integer grid in two dimensions would provide standard Descartes coordinates like (0, 0) for the origin, or (2, 3) for the position that is two units "up" and three units to the "right" of the origin. Implementation of a coordinate system is left up to the user, though there are several established systems provided.

See also:

GridCoord

Subclassed by *kami::GridCoord*

Public Functions

virtual std::string **`to_string`**() const = 0

Convert the coordinate to a human-readable string.

Returns a human-readable form of the *`Coord`* as `std::string`.

Friends

friend std::ostream &**`operator<<`**(std::ostream &lhs, const *Coord* &rhs)

Output a *`Coord`* to the specified output stream.

The form of the output will be the same as that produced by the *`to_string()`* member function.

Parameters

- **`lhs`** – is the stream to output the *`Coord`* to
- **`rhs`** – is the *`Coord`* to output

Returns the output stream for reuse

Class Domain

- Defined in file_include_kami_domain.h

Inheritance Relationships

Derived Type

- `public kami::GridDomain` (*Class GridDomain*)

Class Documentation

class **`Domain`**

Provides an environment for the agents to participate in.

Implementations of virtual environments are expected to subclass *`Domain`*.

Subclassed by *kami::GridDomain*

Protected Functions

`Domain`() = default

Constructor.

Making this constructor protected makes the class abstract without having to create any virtual functions.

Class AgentNotFound

- Defined in file_include_kami_error.h

Inheritance Relationships

Base Type

- `public logic_error`

Class Documentation

class **`AgentNotFound`** : public logic_error

Agent was not found.

Public Functions

inline explicit **`AgentNotFound`**(const char *s)

Constructor.

Parameters **`s`** – text description of the exception

inline explicit **`AgentNotFound`**(const std::string &s)

Constructor.

Parameters **`s`** – text description of the exception

Class LocationInvalid

- Defined in file_include_kami_error.h

Inheritance Relationships

Base Type

- `public domain_error`

Class Documentation

class **`LocationInvalid`** : public domain_error

Location specified is invalid.

See also:

`LocationUnavailable`

Public Functions

inline explicit **LocationInvalid**(const char *s)

Constructor.

Parameters **s** – text description of the exception

inline explicit **LocationInvalid**(const std::string &s)

Constructor.

Parameters **s** – text description of the exception

Class LocationUnavailable

- Defined in file_include_kami_error.h

Inheritance Relationships

Base Type

- `public domain_error`

Class Documentation

class **LocationUnavailable** : public domain_error

Location specified is unavailable.

See also:

LocationInvalid

Public Functions

inline explicit **LocationUnavailable**(const char *s)

Constructor.

Parameters **s** – text description of the exception

inline explicit **LocationUnavailable**(const std::string &s)

Constructor.

Parameters **s** – text description of the exception

Class OptionInvalid

- Defined in file_include_kami_error.h

Inheritance Relationships

Base Type

- `public invalid_argument`

Class Documentation

class **`OptionInvalid`** : public invalid_argument

The option given is not valid at this time.

Public Functions

inline explicit **`OptionInvalid`**(const char *s)

Constructor.

Parameters **`s`** – text description of the exception

inline explicit **`OptionInvalid`**(const std::string &s)

Constructor.

Parameters **`s`** – text description of the exception

Class ResourceNotAvailable

- Defined in file_include_kami_error.h

Inheritance Relationships

Base Type

- `public logic_error`

Class Documentation

class **`ResourceNotAvailable`** : public logic_error

The resource specified is not available at this time.

Public Functions

inline explicit **ResourceNotAvailable**(const char *s)

Constructor.

Parameters **s** – text description of the exception

inline explicit **ResourceNotAvailable**(const std::string &s)

Constructor.

Parameters **s** – text description of the exception

Class Grid1D

- Defined in file_include_kami_grid1d.h

Inheritance Relationships

Base Type

- `public kami::GridDomain` (*Class GridDomain*)

Derived Types

- `public kami::MultiGrid1D` (*Class MultiGrid1D*)
- `public kami::SoloGrid1D` (*Class SoloGrid1D*)

Class Documentation

class **Grid1D** : public kami::*GridDomain*

A one-dimensional grid where each cell may contain agents.

The grid is linear and may wrap around in its only dimension.

See also:

MultiGrid1D

See also:

SoloGrid1D

Subclassed by *kami::MultiGrid1D*, *kami::SoloGrid1D*

Public Functions

explicit **Grid1D**(unsigned int maximum_x, bool wrap_x = false)

Constructor.

Parameters

- **maximum_x** – **[in]** the length of the grid.
- **wrap_x** – **[in]** should the grid wrap around on itself.

virtual *AgentID* **add_agent**(*AgentID* agent_id, const *GridCoord1D* &coord) = 0

Place agent on the grid at the specified location.

Parameters

- **agent_id** – **[in]** the AgentID of the agent to add.
- **coord** – **[in]** the coordinates of the agent.

Returns false if the agent is not placed at the specified location, otherwise, true.

AgentID **delete_agent**(*AgentID* agent_id)

Remove agent from the grid.

Parameters **agent_id** – **[in]** the AgentID of the agent to remove.

Returns the AgentID of the Agent deleted

AgentID **delete_agent**(*AgentID* agent_id, const *GridCoord1D* &coord)

Remove agent from the grid at the specified location.

Parameters

- **agent_id** – **[in]** the AgentID of the agent to remove.
- **coord** – **[in]** the coordinates of the agent.

Returns the AgentID of the Agent deleted

AgentID **move_agent**(*AgentID* agent_id, const *GridCoord1D* &coord)

Move an agent to the specified location.

Parameters

- **agent_id** – **[in]** the AgentID of the agent to move.
- **coord** – **[in]** the coordinates of the agent.

bool **is_location_empty**(const *GridCoord1D* &coord) const

Inquire if the specified location is empty.

Parameters **coord** – **[in]** the coordinates of the query.

Returns true if the location has no Agents occupying it, false otherwise.

bool **is_location_valid**(const *GridCoord1D* &coord) const

Inquire if the specified location is valid within the grid.

Parameters **coord** – **[in]** the coordinates of the query.

Returns true if the location specified is valid, false otherwise.

GridCoord1D **get_location_by_agent**(const *AgentID* &agent_id) const

Get the location of the specified agent.

Parameters **agent_id** – **[in]** the *AgentID* of the agent in question.

Returns the location of the specified *Agent*

std::shared_ptr<std::set<*AgentID*>> **get_location_contents**(const *GridCoord1D* &coord) const

Get the contents of the specified location.

Parameters **coord** – **[in]** the coordinates of the query.

Returns a pointer to a `set` of *AgentID*s. The pointer is to the internal copy of the agent list at the location, therefore, any changes to that object will update the state of the gird. Further, the pointer should not be deleted when no longer used.

bool **get_wrap_x**() const

Inquire to whether the grid wraps in the `x` dimension.

Returns true if the grid wraps, and false otherwise

std::shared_ptr<std::unordered_set<*GridCoord1D*>> **get_neighborhood**(*AgentID* agent_id, bool include_center) const

Return the neighborhood of the specified *Agent*.

Parameters

- **agent_id** – **[in]** the *AgentID* of the agent in question
- **include_center** – **[in]** should the center-point, occupied by the agent, be in the list.

Returns an `unordered_set` of *GridCoord1D* that includes all of the coordinates for all adjacent points.

std::shared_ptr<std::unordered_set<*GridCoord1D*>> **get_neighborhood**(const *GridCoord1D* &coord, bool include_center) const

Return the neighborhood of the specified location.

Parameters

- **coord** – **[in]** the coordinates of the specified location.
- **include_center** – **[in]** should the center-point, occupied by the agent, be in the list.

Returns an `unordered_set` of *GridCoord1D* that includes all of the coordinates for all adjacent points.

unsigned int **get_maximum_x**() const

Get the size of the grid in the `x` dimension.

Returns the length of the grid in the `x` dimension

Protected Functions

GridCoord1D **coord_wrap**(const *GridCoord1D* &coord) const

Automatically adjust a coordinate location for wrapping.

Parameters **coord** – **[in]** the coordinates of the specified location.

Returns the adjusted coordinate wrapped if appropriate.

Protected Attributes

const std::vector<*GridCoord1D*> **directions** = {*GridCoord1D*(1), *GridCoord1D*(-1)}

Direction coordinates.

This can be used for addition to coordinates. Direction `0` is the first direction clockwise from "vertical." In this case, it can be on a vertically-oriented column, upwards, or to the right on a horizontally-oriented column. Then the additional directions are enumerated clockwise.

std::unique_ptr<std::unordered_multimap<*GridCoord1D*, *AgentID*>> **_agent_grid**

An `unordered_set` containing the `AgentIDs` of all agents assigned to this grid.

std::unique_ptr<std::map<*AgentID*, *GridCoord1D*>> **_agent_index**

A map containing the grid location of each agent.

Class Grid2D

- Defined in file_include_kami_grid2d.h

Inheritance Relationships

Base Type

- `public kami::GridDomain` (*Class GridDomain*)

Derived Types

- `public kami::MultiGrid2D` (*Class MultiGrid2D*)
- `public kami::SoloGrid2D` (*Class SoloGrid2D*)

Class Documentation

class **Grid2D** : public kami::*GridDomain*

A two-dimensional grid where each cell may contain agents.

The grid is linear and may wrap around in its only dimension.

See also:

MultiGrid2D

See also:

SoloGrid2D

Subclassed by *kami::MultiGrid2D*, *kami::SoloGrid2D*

Public Functions

explicit **Grid2D**(unsigned int maximum_x, unsigned int maximum_y, bool wrap_x = false, bool wrap_y = false)

Constructor.

Parameters

- **maximum_x** – **[in]** the length of the grid in the first dimension
- **maximum_y** – **[in]** the length of the grid in the second dimension
- **wrap_x** – **[in]** should the grid wrap around on itself in the first dimension
- **wrap_y** – **[in]** should the grid wrap around on itself in the second dimension

virtual *AgentID* **add_agent**(*AgentID* agent_id, const *GridCoord2D* &coord) = 0

Place agent on the grid at the specified location.

Parameters

- **agent_id** – **[in]** the *AgentID* of the agent to add.
- **coord** – **[in]** the coordinates of the agent.

Returns false if the agent is not placed at the specified location, otherwise, true.

AgentID **delete_agent**(*AgentID* agent_id)

Remove agent from the grid.

Parameters **agent_id** – **[in]** the *AgentID* of the agent to remove.

Returns false if the agent is not removed, otherwise, true.

AgentID **delete_agent**(*AgentID* agent_id, const *GridCoord2D* &coord)

Remove agent from the grid at the specified location.

Parameters

- **agent_id** – **[in]** the *AgentID* of the agent to remove.
- **coord** – **[in]** the coordinates of the agent.

Returns false if the agent is not removed, otherwise, true.

AgentID **move_agent**(*AgentID* agent_id, const *GridCoord2D* &coord)

Move an agent to the specified location.

Parameters

- **agent_id** – **[in]** the `AgentID` of the agent to move.
- **coord** – **[in]** the coordinates of the agent.

bool **is_location_empty**(const *GridCoord2D* &coord) const

Inquire if the specified location is empty.

Parameters **coord** – **[in]** the coordinates of the query.

Returns true if the location has no `Agent`s occupying it, false otherwise.

bool **is_location_valid**(const *GridCoord2D* &coord) const

Inquire if the specified location is valid within the grid.

Parameters **coord** – **[in]** the coordinates of the query.

Returns true if the location specified is valid, false otherwise.

virtual *GridCoord2D* **get_location_by_agent**(const *AgentID* &agent_id) const

Get the location of the specified agent.

Parameters **agent_id** – **[in]** the `AgentID` of the agent in question.

Returns the location of the specified `Agent`

std::shared_ptr<std::set<*AgentID*>> **get_location_contents**(const *GridCoord2D* &coord) const

Get the contents of the specified location.

Parameters **coord** – **[in]** the coordinates of the query.

Returns a pointer to a `set` of `AgentID`s. The pointer is to the internal copy of the agent list at the location, therefore, any changes to that object will update the state of the gird. Further, the pointer should not be deleted when no longer used.

bool **get_wrap_x**() const

Inquire to whether the grid wraps in the `x` dimension.

Returns true if the grid wraps, and false otherwise

bool **get_wrap_y**() const

Inquire to whether the grid wraps in the `y` dimension.

Returns true if the grid wraps, and false otherwise

virtual std::shared_ptr<std::unordered_set<*GridCoord2D*>> **get_neighborhood**(*AgentID* agent_id, bool include_center, *GridNeighborhoodType* neighborhood_type) const

Return the neighborhood of the specified *Agent*.

See also:

`NeighborhoodType`

Parameters

- **agent_id** – **[in]** the `AgentID` of the agent in question.

- **neighborhood_type** – **[in]** the neighborhood type.
- **include_center** – **[in]** should the center-point, occupied by the agent, be in the list.

Returns a set of *GridCoord2D* that includes all of the coordinates for all adjacent points.

std::shared_ptr<std::unordered_set<*GridCoord2D*>> **get_neighborhood**(const *GridCoord2D* &coord, bool include_center, *GridNeighborhoodType* neighborhood_type) const

Return the neighborhood of the specified location.

See also:

`NeighborhoodType`

Parameters

- **coord** – **[in]** the coordinates of the specified location.
- **neighborhood_type** – **[in]** the neighborhood type.
- **include_center** – **[in]** should the center-point, occupied by the agent, be in the list.

Returns a set of *GridCoord2D* that includes all of the coordinates for all adjacent points.

unsigned int **get_maximum_x**() const

Get the size of the grid in the `x` dimension.

Returns the length of the grid in the `x` dimension

unsigned int **get_maximum_y**() const

Get the size of the grid in the `y` dimension.

Returns the length of the grid in the `xy dimension

Protected Functions

GridCoord2D **coord_wrap**(const *GridCoord2D* &coord) const

Automatically adjust a coordinate location for wrapping.

Parameters **coord** – **[in]** the coordinates of the specified location.

Returns the adjusted coordinate wrapped if appropriate.

Protected Attributes

const std::vector<*GridCoord2D*> **directions_vonneumann** = {*GridCoord2D*(0, 1), *GridCoord2D*(1, 0), *GridCoord2D*(0, -1), *GridCoord2D*(-1, 0)}

von Neumann neighborhood coordinates

This can be used for addition to coordinates. Direction `0` is the first direction clockwise from "vertical." Then the additional directions are enumerated clockwise.

const std::vector<*GridCoord2D*> **directions_moore** = {*GridCoord2D*(0, 1), *GridCoord2D*(1, 1), *GridCoord2D*(1, 0), *GridCoord2D*(1, -1), *GridCoord2D*(0, -1), *GridCoord2D*(-1, -1), *GridCoord2D*(-1, 0), *GridCoord2D*(-1, 1)}

Moore neighborhood coordinates.

This can be used for addition to coordinates. Direction `0` is the first direction clockwise from "vertical." Then the additional directions are enumerated clockwise.

std::unique_ptr<std::unordered_multimap<*GridCoord2D*, *AgentID*>> **_agent_grid**

A map containing the `AgentID`s of all agents assigned to this grid.

std::unique_ptr<std::map<*AgentID*, *GridCoord2D*>> **_agent_index**

A map containing the grid location of each agent.

Class GridCoord

- Defined in file_include_kami_grid.h

Inheritance Relationships

Base Type

- `public kami::Coord` (*Class Coord*)

Derived Types

- `public kami::GridCoord1D` (*Class GridCoord1D*)
- `public kami::GridCoord2D` (*Class GridCoord2D*)

Class Documentation

class **GridCoord** : public kami::*Coord*

An abstract for gridded coordinates.

All gridded coordinates are expected to subclass `GridCoord`.

Subclassed by *kami::GridCoord1D*, *kami::GridCoord2D*

Public Functions

virtual double **distance**(std::shared_ptr<*Coord*> &p) const = 0

Find the distance between two points.

Find the distance between two points using the specified metric.

However, the coordinate class is not aware of the properties of the *GridDomain* it is operating on. Accordingly, if the direct path is measured, without accounting for and toroidal wrapping of the underlying *GridDomain*.

Parameters `p` – the point to measure the distance to

Returns the distance as a `double`

Class GridCoord1D

- Defined in file_include_kami_grid1d.h

Inheritance Relationships

Base Type

- `public kami::GridCoord` (*Class GridCoord*)

Class Documentation

class **GridCoord1D** : public kami::*GridCoord*

One-dimensional coordinates.

Public Functions

explicit **GridCoord1D**(int x_coord)

Constructor for one-dimensional coordinates.

int **x**() const

Return the `x` coordinate.

virtual std::string **to_string**() const override

Convert the coordinate to a human-readable string.

Returns a human-readable form of the *Coord* as `std::string`.

virtual double **distance**(std::shared_ptr<*Coord*> &p) const override

Find the distance between two points.

Find the distance between two points using the specified metric. There are three options provided by the `GridDistanceType` class. However, of the three distance types provided, all provide the same result so the value is ignored and the single result is returned.

However, the coordinate class is not aware of the properties of the *Grid1D* it is operating on. Accordingly, if the direct path is measured, without accounting for and toroidal wrapping of the underlying *Grid1D*.

Parameters **p** – the point to measure the distance to

Returns the distance as a `double`

Friends

friend bool **operator==**(const *GridCoord1D* &lhs, const *GridCoord1D* &rhs)

Test if two coordinates are equal.

friend bool **operator!=**(const *GridCoord1D* &lhs, const *GridCoord1D* &rhs)

Test if two coordinates are not equal.

friend std::ostream &**operator<<**(std::ostream &lhs, const *GridCoord1D* &rhs)

Output a given coordinate to the specified stream.

inline friend *GridCoord1D* **operator+**(const *GridCoord1D* &lhs, const *GridCoord1D* &rhs)

Add two coordinates together.

inline friend *GridCoord1D* **operator-**(const *GridCoord1D* &lhs, const *GridCoord1D* &rhs)

Subtract one coordinate from another.

inline friend *GridCoord1D* **operator***(const *GridCoord1D* &lhs, double rhs)

Multiply a coordinate by a scalar.

If any component of the resulting value is not a whole number, it is truncated following the same rules as `int`.

inline friend *GridCoord1D* **operator***(double lhs, const *GridCoord1D* &rhs)

Multiply a coordinate by a scalar.

If any component of the resulting value is not a whole number, it is truncated following the same rules as `int`.

Class GridCoord2D

- Defined in file_include_kami_grid2d.h

Inheritance Relationships

Base Type

- `public kami::GridCoord` (*Class GridCoord*)

Class Documentation

class **GridCoord2D** : public kami::*GridCoord*

Two-dimensional coordinates.

Public Functions

GridCoord2D(int x_coord, int y_coord)

Constructor for two-dimensional coordinates.

int **x**() const

Get the coordinate in the first dimension or `x`.

int **y**() const

Get the coordinate in the second dimension or `y`.

virtual std::string **to_string**() const override

Convert the coordinate to a human-readable string.

Returns a human-readable form of the *Coord* as `std::string`.

virtual double **distance**(std::shared_ptr<*Coord*> &p) const override

Find the distance between two points.

Find the distance between two points using the specified metric.

However, the coordinate class is not aware of the properties of the *Grid2D* it is operating on. Accordingly, if the direct path is measured, without accounting for and toroidal wrapping of the underlying *Grid2D*.

Parameters **p** – the point to measure the distance to

Returns the distance as a `double`

double **distance**(std::shared_ptr<*GridCoord2D*> &p, *GridDistanceType* distance_type = *GridDistanceType*::*Euclidean*) const

Find the distance between two points.

Find the distance between two points using the specified metric. There are three options provided by the `GridDistanceType` class.

However, the coordinate class is not aware of the properties of the *Grid2D* it is operating on. Accordingly, if the direct path is measured, without accounting for and toroidal wrapping of the underlying *Grid2D*.

Parameters

- **p** – the point to measure the distance to
- **distance_type** – specify the distance type

Returns the distance as a `double`

Protected Functions

inline double **distance_chebyshev**(std::shared_ptr<*GridCoord2D*> &p) const

Find the distance between two points using the Chebyshev metric.

Parameters **p** – the point to measure the distance to

Returns the distance as a `double`

inline double **distance_euclidean**(std::shared_ptr<*GridCoord2D*> &p) const

Find the distance between two points using the Euclidean metric.

Parameters **p** – the point to measure the distance to

Returns the distance as a `double`

inline double **distance_manhattan**(std::shared_ptr<*GridCoord2D*> &p) const

Find the distance between two points using the Manhattan metric.

Parameters **p** – the point to measure the distance to

Returns the distance as a `double`

Friends

friend bool **operator==**(const *GridCoord2D*&, const *GridCoord2D*&)

Test if two coordinates are equal.

friend bool **operator!=**(const *GridCoord2D*&, const *GridCoord2D*&)

Test if two coordinates are not equal.

friend std::ostream &**operator<<**(std::ostream&, const *GridCoord2D*&)

Output a given coordinate to the specified stream.

inline friend *GridCoord2D* **operator+**(const *GridCoord2D* &lhs, const *GridCoord2D* &rhs)

Add two coordinates together.

inline friend *GridCoord2D* **operator-**(const *GridCoord2D* &lhs, const *GridCoord2D* &rhs)

Subtract one coordinate from another.

inline friend *GridCoord2D* **operator***(const *GridCoord2D* &lhs, const double rhs)

Multiply a coordinate by a scalar.

If any component of the resulting value is not a whole number, it is truncated following the same rules as `int`.

inline friend *GridCoord2D* **operator***(const double lhs, const *GridCoord2D* &rhs)

Multiply a coordinate by a scalar.

If any component of the resulting value is not a whole number, it is truncated following the same rules as `int`.

Class GridDomain

- Defined in file_include_kami_grid.h

Inheritance Relationships

Base Type

- `public kami::Domain` (*Class Domain*)

Derived Types

- `public kami::Grid1D` (*Class Grid1D*)
- `public kami::Grid2D` (*Class Grid2D*)

Class Documentation

class **`GridDomain`** : public kami::*Domain*

An abstract domain based on a gridded environment.

All gridded domains are expected to consist of cells in a rectilinear grid where the cells are equal size and laid out in an ordered fashion.

Subclassed by *kami::Grid1D*, *kami::Grid2D*

Class Model

- Defined in file_include_kami_model.h

Inheritance Relationships

Base Type

- `public std::enable_shared_from_this< Model >`

Derived Type

- `public kami::ReporterModel` (*Class ReporterModel*)

Class Documentation

class **`Model`** : public std::enable_shared_from_this<*Model*>

An abstract for generic models.

See also:

`ReporterModel`

Subclassed by *kami::ReporterModel*

Public Functions

std::shared_ptr<*Domain*> **get_domain**()

Get the `Domain` associated with this model.

Returns a shared pointer to the `Domain`

std::shared_ptr<*Domain*> **set_domain**(std::shared_ptr<*Domain*> domain)

Add a `Domain` to this scheduler.

This method will associate a model with the scheduler.

Returns a shared pointer to the `Domain`

std::shared_ptr<*Population*> **get_population**()

Get the `Population` associated with this model.

Returns a shared pointer to the `Population`

std::shared_ptr<*Population*> **set_population**(std::shared_ptr<*Population*> population)

Add a `Model` to this scheduler.

This method will associate a model with the scheduler.

Returns a shared pointer to the `Population`

std::shared_ptr<*Scheduler*> **get_scheduler**()

Get the `Scheduler` associated with this model.

Returns a shared pointer to the `Scheduler`

std::shared_ptr<*Scheduler*> **set_scheduler**(std::shared_ptr<*Scheduler*> scheduler)

Add a `Model` to this scheduler.

This method will associate a model with the scheduler.

Returns a shared pointer to the `Scheduler`

virtual std::shared_ptr<*Model*> **step**()

Execute a single time step of the model.

This method will collect all the `Agent`s in the `Population` associated with model and pass them to the associated `Scheduler` for stepping.

Returns a shared pointer to the model instance

Protected Attributes

std::shared_ptr<*Domain*> **_domain** = nullptr

Reference copy of the `Domain`

std::shared_ptr<*Population*> **_pop** = nullptr

Reference copy of the `Population`

std::shared_ptr<*Scheduler*> **_sched** = nullptr

Reference copy of the `Scheduler`

Class MultiGrid1D

- Defined in file_include_kami_multigrid1d.h

Inheritance Relationships

Base Type

- `public kami::Grid1D` (*Class Grid1D*)

Class Documentation

class **`MultiGrid1D`** : public kami::*Grid1D*

A one-dimensional grid where each cell may contain multiple agents.

The grid is linear and may wrap around in its only dimension.

See also:

`Grid1D`

See also:

`SoloGrid1D`

Public Functions

`MultiGrid1D`(unsigned int maximum_x, bool wrap_x)

Constructor.

Parameters

- **`maximum_x`** – **[in]** the length of the grid.
- **`wrap_x`** – **[in]** should the grid wrap around on itself.

virtual *AgentID* **`add_agent`**(*AgentID* agent_id, const *GridCoord1D* &coord) override

Place agent on the grid at the specified location.

Parameters

- **`agent_id`** – **[in]** the *`AgentID`* of the agent to add.
- **`coord`** – **[in]** the coordinates of the agent.

Returns false if the agent is not placed at the specified location, otherwise, true

Class MultiGrid2D

- Defined in file_include_kami_multigrid2d.h

Inheritance Relationships

Base Type

- `public kami::Grid2D` (*Class Grid2D*)

Class Documentation

class **`MultiGrid2D`** : public kami::*Grid2D*

A two-dimensional grid where each cell may contain multiple agents.

The grid is linear and may wrap around in either dimension.

See also:

`Grid2D`

See also:

`SoloGrid2D`

Public Functions

`MultiGrid2D``(`unsigned int maximum_x, unsigned int maximum_y, bool wrap_x, bool wrap_y`)`

Constructor.

Parameters

- **`maximum_x`** – **[in]** the length of the grid in the first dimension
- **`maximum_y`** – **[in]** the length of the grid in the second dimension
- **`wrap_x`** – **[in]** should the grid wrap around on itself in the first dimension
- **`wrap_y`** – **[in]** should the grid wrap around on itself in the second dimension

virtual *AgentID* **`add_agent`**`(`*AgentID* agent_id, const *GridCoord2D* &coord`)` override

Place agent on the grid at the specified location.

Parameters

- **`agent_id`** – **[in]** the *`AgentID`* of the agent to add.
- **`coord`** – **[in]** the coordinates of the agent.

Returns the *`AgentID`* of the agent added

Class Population

- Defined in file_include_kami_population.h

Class Documentation

class **Population**

An abstract for generic models.

Public Functions

std::shared_ptr<*Agent*> **get_agent_by_id**(*AgentID* agent_id) const

Get a reference to an `Agent` by `AgentID`

Parameters **agent_id** – **[in]** the `AgentID` to search for.

Returns a reference to the desired `Agent` or nothing is not found

AgentID **add_agent**(const std::shared_ptr<*Agent*> &agent) noexcept

Add an *Agent* to the *Population*.

Parameters **agent** – The *Agent* to add.

Returns the ID of the agent added

std::shared_ptr<*Agent*> **delete_agent**(*AgentID* agent_id)

Remove an *Agent* from the *Population*.

Parameters **agent_id** – The *AgentID* of the agent to remove.

Returns a shared pointer to the *Agent* deleted

std::unique_ptr<std::vector<*AgentID*>> **get_agent_list**() const

Returns the agent list.

Returns a `std::vector` of all the `AgentID`'s in the `Population`

Protected Attributes

std::map<kami::*AgentID*, std::shared_ptr<*Agent*>> **_agent_map**

A mapping of `AgentID` to `Agent` pointers.

This is the mapping of all `AgentID`s to pointers to the corresponding `Agent` in this population. This is left exposed as protected should any subclass wish to manipulate this mapping directly.

Class RandomScheduler

- Defined in file_include_kami_random.h

Inheritance Relationships

Base Types

- `public kami::SequentialScheduler` (*Class SequentialScheduler*)
- `private std::enable_shared_from_this< RandomScheduler >`

Class Documentation

class **RandomScheduler** : public kami::*SequentialScheduler*, private std::enable_shared_from_this<*RandomScheduler*>

Will execute all agent steps in a random order.

A random scheduler will iterate over the agents assigned to the scheduler and call their *step()* function in a random order. That order should be different for each subsequent call to *step()*, but is not guaranteed not to repeat.

Public Functions

RandomScheduler() = default

Constructor.

explicit **RandomScheduler**(std::shared_ptr<std::mt19937> rng)

Constructor.

Parameters **rng** – [in] A uniform random number generator of type `std::mt19937`, used as the source of randomness.

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*Model*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) override

Execute a single time step.

This method will randomize the list of Agents provided then execute the *Agent::step()* method for every *Agent* listed.

Parameters

- **model** – a reference copy of the model
- **agent_list** – list of agents to execute the step

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*ReporterModel*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) override

Execute a single time step for a *ReporterModel*

This method will randomize the list of Agents provided then execute the *Agent::step()* method for every *Agent* listed.

Parameters

- **model** – a reference copy of the *ReporterModel*
- **agent_list** – list of agents to execute the step

Returns returns vector of agents successfully stepped

std::shared_ptr<std::mt19937> **set_rng**(std::shared_ptr<std::mt19937> rng)

Set the RNG.

Set the random number generator used to randomize the order of agent stepping.

Parameters **rng** – [in] A uniform random number generator of type `std::mt19937`, used as the source of randomness.

Returns a shared pointer to the random number generator

std::shared_ptr<std::mt19937> **get_rng**()

Get the RNG.

Get a reference to the random number generator used to randomize the order of agent stepping.

Class Reporter

- Defined in file_include_kami_reporter.h

Inheritance Relationships

Base Type

- `public std::enable_shared_from_this< Reporter >`

Class Documentation

class **Reporter** : public std::enable_shared_from_this<*Reporter*>

A *Reporter* is a module that works with *ReporterAgent* and *ReporterModel* to collect information about the state of the model for later analysis.

Public Functions

Reporter()

Constructor.

std::shared_ptr<*Reporter*> **clear**()

Empty the report.

Clear all entries from the report; new collection operations begin with a blank slate.

Returns a reference copy of the `Reporter`

std::unique_ptr<nlohmann::json> **collect**(const std::shared_ptr<*ReporterModel*> &model)

Collect the current state of the model.

This will collect the current state of each agent associated with the population returned by the `Model`'s `get_population()`.

Parameters **model** – reference copy of the model

Returns a copy of the current report

std::unique_ptr<nlohmann::json> **collect**(const std::shared_ptr<*ReporterModel*> &model, const std::shared_ptr<*Population*> &pop)

Collect the current state of the model.

This will collect the current state of each agent associated with the `Population`.

Parameters

- **model** – reference copy of the model
- **pop** – *Population* to collect on

Returns a copy of the current report

std::unique_ptr<nlohmann::json> **collect**(const std::shared_ptr<*ReporterModel*> &model, const std::unique_ptr<std::vector<*AgentID*>> &agent_list)

Collect the current state of the model.

This will collect the current state of each agent given

Parameters

- **model** – reference copy of the model
- **agent_list** – a vector agents to report on

Returns a copy of the current report

std::unique_ptr<nlohmann::json> **report**(const std::shared_ptr<*ReporterModel*> &model)

Collect the report.

This will return the aggregate report of all the data collected by this `Reporter`.

Parameters **model** – reference copy of the model

Returns a copy of the current report

Protected Attributes

std::unique_ptr<std::vector<nlohmann::json>> **_report_data** = nullptr

A vector of the the report collected so far.

Class ReporterAgent

- Defined in file_include_kami_reporter.h

Inheritance Relationships

Base Type

- `public kami::Agent` (*Class Agent*)

Class Documentation

class **ReporterAgent** : public kami::*Agent*

A superclass for all reporting agents.

All reporting agents should subclass the `ReportingAgent` class. At a minimum, subclasses must implement the `step()` function, to execute a single time step for each agent.

See also:

`Agent`, `StagedAgent`

Public Functions

virtual std::unique_ptr<nlohmann::json> **collect**() = 0

Collect the current state of the agent.

This function should collect the agent's current state. The agent, notably, does not need to collect historical state, as the historical state is retained by the `Reporter` instance until such time as `Reporter::report()` is called. However, the implementation of the *collect()* function is up to the end user who can, ultimately, return whatever they want.

The only restriction on collect is that it must return its data as a nlohmann::json JSON object. See `Reporter` for additional details.

virtual *AgentID* **step**(std::shared_ptr<*ReporterModel*> model) = 0

Execute a time-step for the agent.

This function should step the agent instance. Any activities that the agent should perform as part of its time step should be in this function.

Parameters **model** – a reference copy of the model

Returns a copy of the *AgentID*

Class ReporterModel

- Defined in file_include_kami_reporter.h

Inheritance Relationships

Base Type

- `public kami::Model` (*Class Model*)

Class Documentation

class **ReporterModel** : public kami::*Model*

An abstract for generic models with a reporting capability.

See also:

Model

Public Functions

ReporterModel()

Constructor.

virtual std::unique_ptr<nlohmann::json> **collect**() = 0

Collect the current state of the model.

This function should collect the model's current state. The model, notably, does not need to collect historical state, as the historical state is retained by the *Reporter* instance until such time as *Reporter::report()* is called. However, the implementation of the *collect()* function is up to the end user who can, ultimately, return whatever they want.

This is not expected to return agent data collection, as the agents' information is collected separately.

virtual unsigned int **get_step_id**()

Get the step id of the model.

The step_id should probably be a monotonically incrementing integer.

virtual std::shared_ptr<*Model*> **step**() override

Execute a single time step of the model.

This method will collect all the *Agent*s in the *Population* associated with the model and pass them to the associated *Scheduler* for stepping. After scheduling, this method will run the *collect()* for the *Reporter* associated with this model.

Returns a shared pointer to the model instance

std::unique_ptr<nlohmann::json> **report**()

Get the current report.

This method will return an object containg the data collected to that point in the simulation.

Returns a unique pointer to a `nlohmann::json` object representing the current report

Protected Attributes

std::shared_ptr<*Reporter*> **`_rpt`**

The current report.

unsigned int **`_step_count`** = { }

The model's current step count.

Class Scheduler

- Defined in file_include_kami_scheduler.h

Inheritance Relationships

Derived Type

- `public kami::SequentialScheduler` (*Class SequentialScheduler*)

Class Documentation

class **`Scheduler`**

Create a Kami scheduler.

Schedulers are responsible for executing each time step in the model. A scheduler will have a collection of agents assigned to it and will execute the step function for each agent based on the type of scheduling implemented.

Subclassed by *kami::SequentialScheduler*

Public Functions

virtual std::unique_ptr<std::vector<*AgentID*>> **`step`**(std::shared_ptr<*Model*> model) = 0

Execute a single time step.

This method will step through the list of Agents in the *`Population`* associated with `model` and then execute the *`Agent::step()`* method for every *Agent* assigned to this scheduler in the order assigned.

Parameters **`model`** – a reference copy of the model

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **`step`**(std::shared_ptr<*ReporterModel*> model) = 0

Execute a single time step for a *`ReporterModel`*

This method will step through the list of Agents in the scheduler's internal queue and then execute the *`Agent::step()`* method for every *Agent* assigned to this scheduler in the order assigned.

Parameters **`model`** – a reference copy of the *`ReporterModel`*

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*Model*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) = 0

Execute a single time step.

This method will step through the list of Agents provided and then execute the `Agent::step()` method for every *Agent* assigned to this scheduler in the order assigned.

Parameters

- **model** – a reference copy of the model
- **agent_list** – list of agents to execute the step

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*ReporterModel*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) = 0

Execute a single time step for a `ReporterModel`

This method will step through the list of Agents in the scheduler's internal queue and then execute the `Agent::step()` method for every *Agent* assigned to this scheduler in the order assigned.

Parameters

- **model** – a reference copy of the `ReporterModel`
- **agent_list** – list of agents to execute the step

Returns returns vector of agents successfully stepped

Protected Attributes

int **_step_counter** = 0

Counter to increment on each step

Class SequentialScheduler

- Defined in file_include_kami_sequential.h

Inheritance Relationships

Base Type

- `public kami::Scheduler` (*Class Scheduler*)

Derived Types

- public kami::RandomScheduler (*Class RandomScheduler*)
- public kami::StagedScheduler (*Class StagedScheduler*)

Class Documentation

class **SequentialScheduler** : public kami::*Scheduler*

Will execute all agent steps in a sequential order.

A sequential scheduler will iterate over the agents assigned to the scheduler and call their *step()* function in a sequential order. That order is preserved between calls to *step()* but may be modified by addAgent() or deleteAgent().

Subclassed by *kami::RandomScheduler*, *kami::StagedScheduler*

Public Functions

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*Model*> model) override

Execute a single time step.

This method will step through the list of Agents in the scheduler's internal queue and then execute the *Agent::step()* method for every *Agent* assigned to this scheduler in the order assigned.

Parameters **model** – a reference copy of the model

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*ReporterModel*> model) override

Execute a single time step for a *ReporterModel*

This method will step through the list of Agents in the scheduler's internal queue and then execute the *Agent::step()* method for every *Agent* assigned to this scheduler in the order assigned.

Parameters **model** – a reference copy of the *ReporterModel*

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*Model*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) override

Execute a single time step.

This method will step through the list of Agents provided and then execute the *Agent::step()* method for every *Agent* assigned to this scheduler in the order assigned.

Parameters

- **model** – a reference copy of the model
- **agent_list** – list of agents to execute the step

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*ReporterModel*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) override

Execute a single time step for a `ReporterModel`

This method will step through the list of Agents in the scheduler's internal queue and then execute the `Agent::step()` method for every *Agent* assigned to this scheduler in the order assigned.

Parameters

- **model** – a reference copy of the `ReporterModel`
- **agent_list** – list of agents to execute the step

Returns returns vector of agents successfully stepped

Class SoloGrid1D

- Defined in file_include_kami_sologrid1d.h

Inheritance Relationships

Base Type

- `public kami::Grid1D` (*Class Grid1D*)

Class Documentation

class **SoloGrid1D** : public kami::*Grid1D*

A one-dimensional grid where each cell may contain one agents.

The grid is linear and may wrap around in its only dimension.

See also:

`Grid1D`

See also:

`MultiGrid1D`

Public Functions

SoloGrid1D(unsigned int maximum_x, bool wrap_x)

Constructor.

Parameters

- **maximum_x** – **[in]** the length of the grid.
- **wrap_x** – **[in]** should the grid wrap around on itself.

virtual *AgentID* **add_agent**(*AgentID* agent_id, const *GridCoord1D* &coord) override

Place agent on the grid at the specified location.

Parameters

- **agent_id** – **[in]** the *AgentID* of the agent to add.
- **coord** – **[in]** the coordinates of the agent.

Returns false if the agent is not placed at the specified location, otherwise, true

Class SoloGrid2D

- Defined in file_include_kami_sologrid2d.h

Inheritance Relationships

Base Type

- `public kami::Grid2D` (*Class Grid2D*)

Class Documentation

class **SoloGrid2D** : public kami::*Grid2D*

A two-dimensional grid where each cell may contain multiple agents.

The grid is linear and may wrap around in either dimension.

See also:

Grid2D

See also:

MultiGrid2D

Public Functions

SoloGrid2D(unsigned int maximum_x, unsigned int maximum_y, bool wrap_x, bool wrap_y)

Constructor

Parameters

- **maximum_x** – **[in]** the length of the grid in the first dimension
- **maximum_y** – **[in]** the length of the grid in the second dimension
- **wrap_x** – **[in]** should the grid wrap around on itself in the first dimension
- **wrap_y** – **[in]** should the grid wrap around on itself in the second dimension

virtual *AgentID* **add_agent**(*AgentID* agent_id, const *GridCoord2D* &coord) override

Place agent on the grid at the specified location.

Parameters

- **agent_id** – **[in]** the `AgentID` of the agent to add.
- **coord** – **[in]** the coordinates of the agent.

Returns false if the agent is not placed at the specified location, otherwise, true

Class StagedAgent

- Defined in file_include_kami_agent.h

Inheritance Relationships

Base Type

- `public kami::Agent` (*Class Agent*)

Class Documentation

class **StagedAgent** : public kami::*Agent*

A superclass for all staged agents.

Staged agents use a two-phase step to allow agents to take actions without updating the state of the model before all agents have been allowed to update. All work necessary to advance the `StagedAgent` state should take place in the `step()` function. However, the `StagedAgent` should not actually update the state, and instead save the results for later use. Finally, during the `advance()` stage, the *StagedAgent* state should update.

`StagedAgents` must implement both the `step()` and `advance()` functions.

Public Functions

virtual *AgentID* **advance**(std::shared_ptr<*Model*> model) = 0

Post-step advance the agent.

This method should be called after `step()`. Any updates or cleanups to the agent that must happen for the `StagedAgent` to complete its step must happen here.

Parameters **model** – a reference copy of the model

Class StagedScheduler

- Defined in file_include_kami_staged.h

Inheritance Relationships

Base Type

- `public kami::SequentialScheduler` (*Class SequentialScheduler*)

Class Documentation

class **StagedScheduler** : public kami::*SequentialScheduler*

Will execute all agent steps in a sequential order.

A sequential scheduler will iterate over the agents assigned to the scheduler and call their `step()` function in a sequential order. That order is preserved between calls to `step()` but may be modified by `add_agent()` or `delete_agent()`.

Public Functions

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*Model*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) override

Execute a single time step.

This method will step through the list of Agents in the scheduler's internal queue and execute the `Agent::step()` method for each `Agent` in the same order. Finally, it will execute the `Agent::advance()` method for each *Agent* in the same order.

Parameters

- **`model`** – a reference copy of the model
- **`agent_list`** – list of agents to execute the step

Returns returns vector of agents successfully stepped

virtual std::unique_ptr<std::vector<*AgentID*>> **step**(std::shared_ptr<*ReporterModel*> model, std::unique_ptr<std::vector<*AgentID*>> agent_list) override

Execute a single time step for a `ReporterModel`

This method will step through the list of Agents in the scheduler's internal queue and then execute the `Agent::step()` method for every *Agent* assigned to this scheduler in the order assigned.

Parameters

- **`model`** – a reference copy of the `ReporterModel`
- **`agent_list`** – list of agents to execute the step

Returns returns vector of agents successfully stepped

Enums

Enum GridDistanceType

- Defined in file_include_kami_grid.h

Enum Documentation

enum `kami::`**`GridDistanceType`**

Distance types for orthogonal grid domains.

Values:

enumerator **`Euclidean`**

Euclidean distance.

The Euclidean distance is the length of the line segment connecting two points. This is commonly called a "beeline" or "as the crow flies."

enumerator **`Manhattan`**

Manhattan distance.

The Manhattan distance is the sum of the absolute value of the differences of the elements. This is commonly called the "taxicab distance," "rectilinear distance," or many other formal names.

enumerator **`Chebyshev`**

Chebyshev distance.

The Chebyshev distance, also called the "chessboard" distance is the number of single point jumps necessary to move from one point to the next. This can be likened to a king on a chessboard and the number of moves necessary to move from a given point to any other given point.

Enum GridNeighborhoodType

- Defined in file_include_kami_grid.h

Enum Documentation

enum `kami::`**`GridNeighborhoodType`**

Neighborhood types for orthogonal grid domains of cells.

Orthogonal grid domains are those that provide cells equidistant along a standard Cartesian grid. `GridNeighborhoodType` allows for the distinction between those neighborhoods that include those cells touching on the corners or diagonally and those neighborhoods that do not.

Values:

enumerator **`Moore`**

Moore neighborhood.

Moore neighborhood types include diagonally-adjacent cells as neighbors.

enumerator **`VonNeumann`**

Von Neumann neighborhood.

Von Neumann neighborhood types do not include diagonally-adjacent cells as neighbors.

Functions

Function kami::get_version

- Defined in file_include_kami_kami.h

Function Documentation

inline semver::version `kami::`**`get_version`**`()`

Get the current version of Kami.

Returns a `semver::version` object containing version information

Typedefs

Typedef kami::Position

- Defined in file_include_kami_position.h

Typedef Documentation

typedef std::variant<*GridCoord1D*, *GridCoord2D*> `kami::`**`Position`**

1.1.5 Examples

- bankreserves
- boltzmann1d
- boltzmann2d
- starter

bankreserves

This example provides a two-dimensional bank reserves model (BSM) as an example of a simple application of the reporter classes for monitoring the internal functioning of the model.

The BSM is a type of computational model that simulates the behavior of customers and their interactions with a bank. It is used to study the dynamics of the money supply and the management of reserves by the bank.

In a BSM, individuals are represented as autonomous agents that make decisions about saving, borrowing, and repaying loans based on their individual objectives and constraints. The bank is also represented as an agent that maintains accounts for each individual. The interactions between individuals and the bank are simulated over time, and the model can be used to study the effects of different reserve requirements policies on the creation of money, borrowing, lending, and savings.

One of the main advantages of a BSM is that it allows for the examination of the micro-level interactions between individuals and the bank, which can provide a more detailed understanding of the dynamics of the monetary system.

It is important to note that BSMs are a simplified representation of the real world and may not capture all the nuances of the monetary system being studied. It's also important to use this model in conjunction with other methods to fully understand the monetary system.

Option	Description
-c *agent_count*	Set the number of agents
-f *output_file_name*	Set the JSON report file
-l *log_level_option*	Set the logging level
-n *max_steps*	Set the number of steps to run the model
-s *initial_seed*	Set the initial seed
-x *x_size*	Set the number of columns
-y *y_size*	Set the number of rows
-w *max_initial_wealth*	Set the maximum initial agent wealth

boltzmann1d

This example provides a one-dimensional Boltzmann wealth model (BWM) as an example of a simple application of the one-dimensional gridded system.

The BWM is a type of agent-based model used to study the distribution of wealth among individuals or agents within a population. The model is named after the physicist Ludwig Boltzmann, who first proposed a similar model to study the distribution of energy among particles in a gas.

In a BWM, agents are assigned a certain amount of wealth, and the model simulates their interactions over time. These interactions can include buying and selling goods and services, lending and borrowing money, and inheriting wealth from other agents.

The key feature of the BWM is that it incorporates a "wealth-exchange mechanism" which determines the probability of agents making a wealth exchange with each other. This mechanism is often based on the difference in wealth between agents, with wealthier agents more likely to make exchanges with other wealthy agents.

The model can be run for a specified number of time steps, and the resulting wealth distribution can be analyzed to study the emergence of wealth inequality and the factors that contribute to it. The model can also be used to study the effects of different policies or interventions on the wealth distribution.

The BWM has been used to study a variety of different economic systems, including capitalist, socialist, and feudal systems. However, it is important to note that like other agent-based models, the BWM is a simplified representation of the real world and may not capture all the nuances of the economic system being studied.

Overall, the BWM is a useful tool for studying the distribution of wealth and the emergence of wealth inequality in a population. It can provide insight into the factors that contribute to wealth inequality and the effects of different policies on the distribution of wealth.

Option	Description
-c *agent_count*	Set the number of agents
-l *log_level_option*	Set the logging level
-n *max_steps*	Set the number of steps to run the model
-s *initial_seed*	Set the initial seed
-x *x_size*	Set the number of columns

boltzmann2d

This example provides a two-dimensional Boltzmann wealth model (BWM) as an example of a simple application of the two-dimensional gridded system.

The BWM is a type of agent-based model used to study the distribution of wealth among individuals within a population. The model simulates agents' interactions over time, such as buying and selling goods, lending and borrowing money, and inheriting wealth. The model is based on a "wealth-exchange mechanism" which determines the probability of agents making a wealth exchange with each other, it is often based on the difference in wealth between agents. The model can be run for a specified number of time steps, and the resulting wealth distribution can be analyzed to study the emergence of wealth inequality and the factors that contribute to it.

For more information on BWMs, please see the boltzmann1d example documentation.

Option	Description
-c *agent_count*	Set the number of agents
-l *log_level_option*	Set the logging level
-n *max_steps*	Set the number of steps to run the model
-s *initial_seed*	Set the initial seed
-x *x_size*	Set the number of columns
-y *y_size*	Set the number of rows

starter

This example provides a starter scaffold for beginning a new agent-based model (ABM). The agents and models perform no real functions in the starter and is likely to be the most minimum functioning model.

Option	Description
-c *agent_count*	Set the number of agents
-l *log_level_option*	Set the logging level
-n *max_steps*	Set the number of steps to run the model
-s *initial_seed*	Set the initial seed

1.1.6 Changelog

Below is the consolidated changelog for Kami.

- : Added baseline for continuous domains
- : Started using Conventional Commits
- : Streamlined documentation builds
- : Corrected bug in documentation build
- : Added distance measures to grid coordinate objects
- : Switched from ranlux24 to mt19937 due to speed
- : Added some useful constants, for use as random seeds
- : Added data collecting and reporting modules
- : Added Bank Reserves model to demonstrate reporting
- : Moved to exception-based error handling
- : Readded step() to the Model interface
- : Added documentation for each example
- : Added a minimal example and tutorial
- : Updated all support packages to most current versions
- : Revised interfaces to the grids
- : Removed step()/run() from Model interface
- : Completed basic unit tests
- : Added a to do list to the documentation
- : Added background info to READ ME
- : Completed initial unit tests
- : Added support semver versioning via neargye-semver
- : Restructured the entire interface
- : Added a barebones "starter" model to build from
- : Corrected the badge link in the README file
- : Make library static by default
- : Numerous build process cleanups
- : Numerous code cleanups to remove void returns and eliminate stored Model references
- : Numerous documentation cleanups
- : Fixed Changelog with current releases added
- : Fixed Changelog to move docs to support
- : Fixed README file links
- : Retagged previous versions using pure Semantic Versioning
- : Cleaned up numerous issues found by CLion's linter
- : Added a changelog!

- : Documentation for *kami::RandomScheduler*
- : Added basic installation instructions
- : Added a new overview to the documents
- : Initial public release.

1.1.7 To Do List

Must Dos

The list below is a list of things considered necessary before a 1.0 release. This list is *not* static.

- Network domain
- Hexgrid domain
- Continuous grid domain

Wishlist

The list below is a list of things considered nice to have at any point.

- Revise unit tests to take advantage of fixtures
- Network Boltzmann model example
- Additional examples as appropriate
- Globe domain, on sphere, with latitude and longitude

1.1.8 License

INDEX

K