SOBA Documentation

Release 1

Eduardo Merino

Jan 09, 2018

Contents

CHAPTER 1

SOBA Overview

SOBA. is a new system of Simulation of Occupancy Based on Agents implemented in Python., which has become an increasingly popular language for scientific computing.

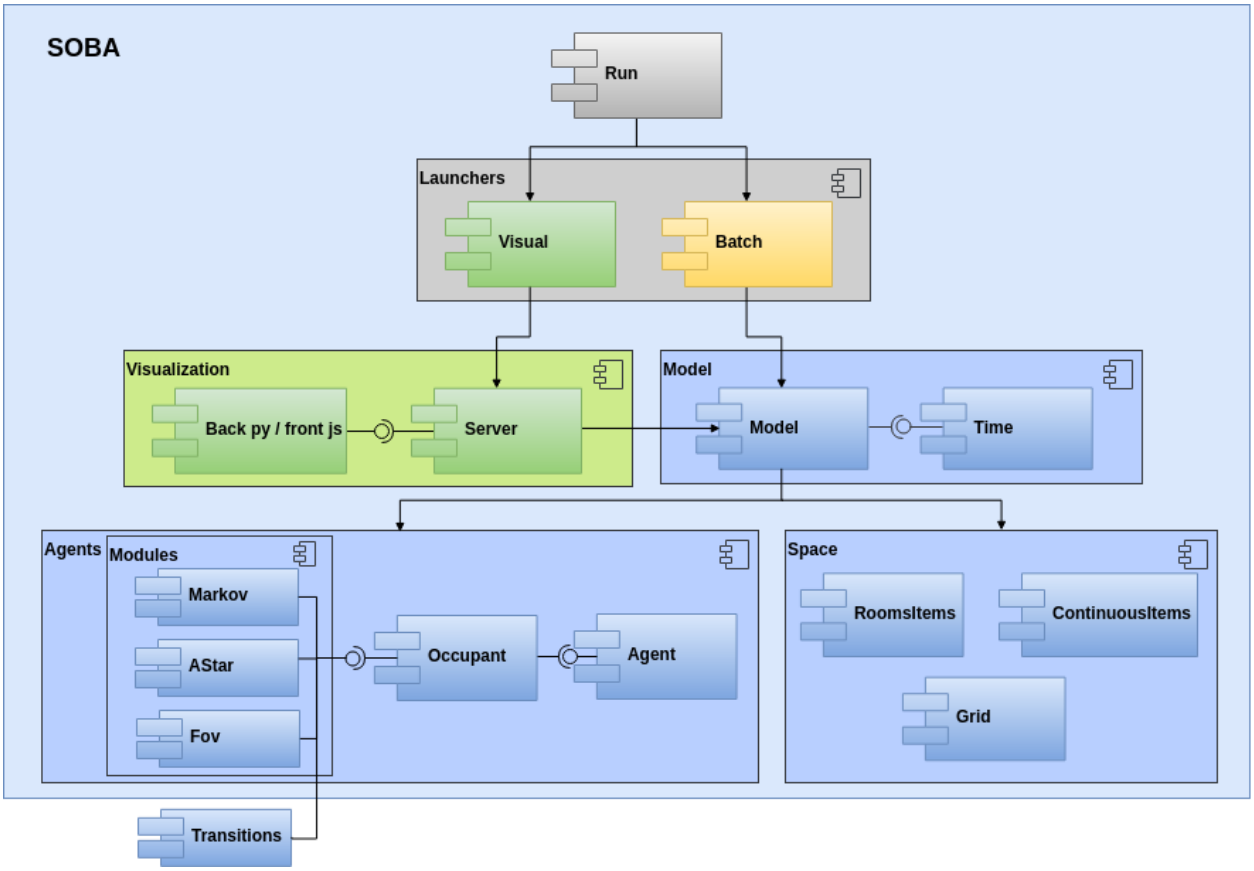
This software is useful for conducting studies based on occupancy simulations, mainly in buildings, such as drill simulation or energy studies.

The simulations are configured by declaring one or more types of occupants, with specific and definable behavior, a physical space (rooms of the building) and agents that are interconnected with each other and with the occupants. The simulation and results can be evaluated both in real time and post-simulation.

It is provided as open source software:

`Github repository`: https://github.com/gsi-upm/soba

1.1 Arquitecture Description



SOBA is implemented through 5 modules which group independent components with a related function.

- **Module Model.**
  - **Model**. This component is the core of the simulations. The model creates and manages space and agents, provides a scheduler that controls the agents activation regime, stores model-level parameters and serves as a container for the rest of components.
  - Time. Component of time management during the simulation in sexagesimal units and controller of the scheduler during the simulation.
- **Module Agents.**
  - **Agent**. Base class to define any type of agent, which performs actions and interactions within a model.
  - **Occupant**. An object of the Occupant class is a type of agent developed and characterized to simulate the behavior of crowds in buildings.
  - **Occupancy module.**
    - **Markov**.
    - AStar. Auxiliar class used by the occupants to move in the building.
    - FOV.This component is a permissive field of view, which is useful to define the occupant visibility.
  - Transitions. This external package is a lightweight, object-oriented state machine implementation in Python.

- **Module Space.**
  - **Grid**. The space where the agents are situated and where they perform their actions is defined by means of a grid with coordinates (x, y).
  - ContinuousItems / RoomsItems. Various classes that define the representation of physical space objects in the model.
- Module Visualization. Two components provide a simple mechanism to represent the model in a web interface, based on HTML rendering though a server interface, implemented with web sockets.
- Module Launchers. The simulation will be executed defining a type of perfomance: in bacth or with visual representation.

CHAPTER 2

How install

2.1 Pip instalation

```
$ pip install soba
```

2.2 Github repository

https://github.com/gsi-upm/soba

CHAPTER 3

Introductory Tutorial

3.1 Instalation

First of all, you need to install the package using pip.

```
$ pip install soba
```

In case of error, this other command should be used, ensuring to have installed python 3 and pip 3.

```
$ pip3 install soba
```

3.2 Tutorial

The SOBA tool can be provided to be used directly on two scenarios:

1. Generic case with a space defined as a grid of a given square size (by default, half a meter on each side).
2. Simplified case with a room defined by rooms, to perform simulations in simplified buildings that require less consumption of resources and specifications.

An introductory tutorial will be presented for each case, although most parameters are common or similar.

SOBA enables the performance of the simulations in two modes:

1. With visual representation.
2. In batch mode.

In the tutorials, the small modifications required to use each posibility are reflected.

IMPORTANT NOTE: The .py files described in this tutorial are available in the github repository https://github.com/gsi-upm/soba/tree/master/projects/basicExamples

3.2.1 Implementing a sample model with continuous space

Once soba is installed, the implementation can be started. First we define the generic parameters to both types of scenario.

1.- We define the characteristics of the occupants

```
from collections import OrderedDict
#JSON to store all the informacion.
jsonsOccupants = []

#Number of occupants
N = 12

#Definition of the states
states = OrderedDict([('Leaving','out'), ('Resting', 'sofa'), ('Working in my␣
↪laboratory', 'wp')])

#Definition of the schedule
schedule = {'t1': "09:00:00", 't2': "13:00:00", 't3': "14:10:00"}

#Possible Variation on the schedule
variation = {'t1': "00:10:00", 't2': "01:20:00", 't3': "00:20:00"}

#Probability of state change associated with the Markovian chain as a function of the␣
↪temporal period.
markovActivity = {
    '-t1': [[100, 0, 0], [0, 0, 0], [0, 0, 0]],
    't1-t2': [[30, 40, 30], [0, 50, 50], [0, 50, 50]],
    't2-t3': [[0, 0, 0], [50, 50, 0], [0, 0, 0]],
    't3-': [[0, 50, 50], [10, 90, 0], [0, 0, 0]]
}

#Time associated to each state (minutes)
timeActivity = {
    '-t1': [60, 0, 0], 't1-t2': [2, 60, 15], 't2-t3': [60, 10, 15], 't3-': [60, 20,␣
↪15]
}

#Store the information
jsonOccupant = {'type': 'example' , 'N': N, 'states': states , 'schedule': schedule,
↪'variation': variation, 'markovActivity': markovActivity, 'timeActivity':␣
↪timeActivity}
jsonsOccupants.append(jsonOccupant)
```

2.- We define the building plan or the distribution of the space.

```
import soba.visualization.ramen.mapGenerator as ramen

with open('labgsi.blueprint3d') as data_file:
    jsonMap = ramen.returnMap(data_file)
```

3.- We implement a Model inheriting a base class of SOBA.

```
from soba.model.model import ContinuousModel
import datetime as dt

class ModelExample(ContinuousModel):
```

```
    def __init__(self, width, height, jsonMap, jsonsOccupants, seed = dt.datetime.
↪now()):
        super().__init__(width, height, jsonMap, jsonsOccupants, seed = seed)

    def step(self):
        if self.clock.clock.day > 3:
            self.finishSimulation = True
        super().step()
```

4.- We call the execution methods.

```
4.1-With visual representation.
```

```
import soba.run

soba.run.run(ModelExample, [], cellW, cellH, jsonMap, jsonsOccupants)
```

```
4.1- Bacth mode.
```

```
#Fixed parameters during iterations
fixed_params = {"width": cellW, "height": cellH, "jsonMap": jsonMap, "jsonsOccupants
↪": jsonsOccupants}
#Variable parameters to each iteration
variable_params = {"seed": range(10, 500, 10)}

soba.run.run(ModelExample, fixed_params, variable_params)
```

3.2.2 Implementing a sample model with simplified space

Once soba is installed, the implementation can be started. First we define the generic parameters to both types of scenario.

1.- We define the characteristics of the occupants

```
from collections import OrderedDict
#JSON to store all the informacion.
jsonsOccupants = []

#Number of occupants
N = 3

#Definition of the states
states = OrderedDict([('out','Pos1'), ('Working in my laboratory', {'Pos2': 1, 'Pos3
↪': 2})])

#Definition of the schedule
schedule = {'t1': "09:00:00", 't2': "13:00:00", 't3': "14:10:00"}

#Possible Variation on the schedule
variation = {'t1': "00:10:00", 't2': "01:20:00", 't3': "00:20:00"}

#Probability of state change associated with the Markovian chain as a function of the␣
↪temporal period.
markovActivity = {
    '-t1': [[100, 0], [0, 0]],
```

```
    't1-t2': [[50, 50], [0, 0]],
    't2-t3': [[0, 0], [50, 0]],
    't3-': [[0, 50], [10, 90]]
}

#Time associated to each state (minutes)
timeActivity = {
    '-t1': [60, 0],
    't1-t2': [2, 60],
    't2-t3': [60, 10],
    't3-': [60, 20]
}

#Store the information
jsonOccupant = {'type': 'example' , 'N': N, 'states': states , 'schedule': schedule,
↪'variation': variation,
                'markovActivity': markovActivity, 'timeActivity': timeActivity}
jsonsOccupants.append(jsonOccupant)
```

2.- We define the building plan or the distribution of the space.

```
jsonMap = {
  'Pos1': {'entrance':'', 'conectedTo': {'U':'Pos2'}, 'measures': {'dx':2, 'dy':2}},
  'Pos2': {'measures': {'dx':3, 'dy':3.5}, 'conectedTo': {'R':'Pos3'}},
  'Pos3': {'measures': {'dx':3, 'dy':3.5}}
}
```

3.- We implement a Model inheriting a base class of SOBA.

```
from soba.model.model import ContinuousModel
import datetime as dt

class ModelExample(RoomsModel):

    def __init__(self, width, height, jsonMap, jsonsOccupants, seed = dt.datetime.
↪now()):
        super().__init__(width, height, jsonMap, jsonsOccupants, seed = seed)

    def step(self):
        if self.clock.clock.day > 3:
            self.finishSimulation = True
        super().step()
```

4.- We call the execution methods. 4.1- With visual representation.

```
cellW = 4
cellH = 4

soba.run.run(ModelExample, [], cellW, cellH, jsonMap, jsonsOccupants)
```

```
4.1- Bacth mode.
```

```
#Fixed parameters during iterations
fixed_params = {"width": cellW, "height": cellH, "jsonMap": jsonMap, "jsonsOccupants
↪": jsonsOccupants}
#Variable parameters to each iteration
variable_params = {"seed": range(10, 500, 10)}
```

```
soba.run.run(ModelExample, fixed_params, variable_params)
```

3.3 Running the simulation using the terminal

```
$ python exampleContinuous.py -v
```

Options:

```
-v,     Visual option on browser

-b,     Background option

-r,     Ramen option
```

CHAPTER 4

APIs

4.1 Model Module Documentation

4.1.1 Model

4.1.2 Time Control

4.2 Agents Module Documentation

4.2.1 Agent

class agent.**Agent**(*unique_id*, *model*)

Base class to create Agent objects. The agents are controlled by the scheduler of the Model associated.

Attributes: model: Model associated to the agent. unique_id: Unique id of the agent. color: Color with which the agent will be represented in the visualization.

Methods: step: Method invoked by the Model scheduler in each step. Step common to all Agents.

step()

Method invoked by the Model scheduler in each step. Step common to all Agents.

4.2.2 Occupant

4.2.3 Agent Modules

AStar

aStar.**canMovePos**(*model*, *cellPos*, *posAux*, *others=[]*)

Evaluate if a position is reachable in a continuous space.

Args: model: Model which invokes the algorithm. cellPos: a first one position given as (x, y). posAux: a second one position given as (x, y). others: List of auxiliary positions given to be considered impenetrable, that is, they will not be used by the AStar.

Return: List of positions (x, y).

aStar.**getConectedCellsContinuous**(*model*, *cell*, *others*)

Gets a list of connected cells in a continuous space.

Args: model: Model which invokes the algorithm. cell: cell object corresponding to the position. other: List of auxiliary positions given to be considered impenetrable, that is, they will not be used by the AStar.

Return: List of positions (x, y).

aStar.**getConectedCellsRooms**(*model*, *cell*)

Gets a list of connected cells in a space defined by rooms.

Args: model: Model which invokes the algorithm. cell: cell object corresponding to the room.

Return: List of positions (x, y).

aStar.**getPathContinuous**(*model*, *start*, *finish*, *other=[]*)

Calculate the optimal path in the models with the space continuous.

Args: model: Model which invokes the algorithm. start: Initial position. finish: Final position. other: List of auxiliary positions given to be considered impenetrable, that is, they will not be used by the AStar.

Return: List of positions (x, y).

aStar.**getPathRooms**(*model*, *start*, *finish*)

Calculate the optimal path in the models with the space defined by rooms.

Args: model: Model which invokes the algorithm. start: Initial position. finish: Final position.

Return: List of positions (x, y).

Markov

class behaviourMarkov.**Markov**(*agent_aux*)

Base class to models the activity of the agents by means of Markovian behavior.

Attributes: agent: Agent that is controlled by this models.

Methods: runStep: Execute a Markovian state change by evaluating the initial state and the probabilities associated with each possible state. getNextState: Evaluate a random change based on the probabilities corresponding to each state.

getNextState(*markov_matrix*, *NumberCurrentState*)

Evaluate a random change based on the probabilities corresponding to each state.

Args: markov_matrix: Markov matrix corresponding to a certain moment. NumberCurrentState: Unique id as number of the current state.

runStep(*markov_matrix*)

Execute a Markovian state change by evaluating the initial state and the probabilities associated with each possible s

Args: markov_matrix: Markov matrix corresponding to a certain moment.

FOV

fov.**FOV_RADIUS = 30000**
In the file aStar.py the filed of vision algorithm is implemented.

class fov.**Map**(*map*)
Class to calculate the field of vision (fov).

Attributes: data: Map to which to apply the algorithm.

Methods: do_fov: Calculate the field of view from a position (x, y).

More information: http://www.roguebasin.com/index.php?title=Python_shadowcasting_implementation

do_fov(*x*, *y*)

Calculate the field of view from a position (x, y).

Args: x, y: Observer's position

Return: Array of sight positions.

fov.**makeFOV**(*dungeon*, *pos*)

Create the invocation object of the fov algorithm and invoke it.

Args: dungeon: Array pos: Observer's position

Return: Array of sight positions.

4.3 Space Module Documentation

4.3.1 Grid

In the file grid.py it is defined the class Grid, which implements the space where take place the simulation as a grid (x, y).

class grid.**Grid**(*width*, *height*)
Class to implement the space where take place the simulation as a grid (x, y).

Attributes: height: Height in number of grid cells. width: Width in number of grid cells. grid: List of rows x, rows are lists of positions y. That is, a matrix of positions [x][y].

Methods: get_all_item: Get all the elements that have been placed in the grid. get_items_in_pos: Gets the elements located in a grid position. move_item: Change the position of a grid element. place_item: Place an element in a grid position. remove_item: Remove an item from the grid. is_cell_empty: Evaluate if a cell does not contain any item.

get_all_item()

Get all the elements that have been placed in the grid. Return: List of items.

get_items_in_pos(*pos*)

Gets the elements located in a grid position.

Args: pos: Position of the grid as (x, y).

Return: List of items.

is_cell_empty(*pos*)

Evaluate if a cell does not contain any item.

Args: pos: Position of the grid as (x, y).

Return: True (yes) or False (no).

move_item(*item*, *pos*)

Change the position of a grid element.

Args: item: Element in the grid. pos: New position of the item.

place_item(*item*, *pos*)

Place an element in a grid position.

Args: pos: Position of the grid as (x, y). item: Element outside the grid.

remove_item(*pos*, *item*)

Remove an item from the grid.

Args: pos: Position of the grid as (x, y). item: Element inside the grid.

4.3.2 Items in continuous modeling

In the file continuousItems.py four classes are defined to implement the elements of the physical space in a continuous model:

-GeneralItem: Class that implements generic elements positioned on the map with the effect of being impenetrable. -Door: Class that implements bulding plane doors. -Wall: Class that implements building walls. -Poi: Class that implements points of interest where Occupancy objects perform certain actions.

class continuousItems.**Door**(*model*, *pos1*, *pos2*, *rot*, *state=True*)

Class that implements bulding plane doors.

Attributes: state: Door status, open (True) or closed (False). pos1: First position to access to the door. pos2: Second position to access to the door. rot: Door orientation in the grid ('x' or 'y').

Methods: open: Change the status of the door to open. close: Change the status of the door to close.

close()
Change the status of the door to close (False)

open()
Change the status of the door to open (True)

class continuousItems.**GeneralItem**(*model*, *pos*, *color=None*)

Class that implements generic elements positioned on the map with the effect of being impenetrable.

Attributes: pos: Position where the object is located. color: Color with which the object will be represented in the visualization.

class continuousItems.**Poi**(*model*, *pos*, *ide*, *share=True*, *color=None*)

Class that implements relevant elements in the simulations: points of interest where Occupancy objects perform certain a

Attributes: pos: Position where the object is located. ide: Unique identifier associated with the point of interest. share: Define if the poi can be shared by more than one occupant. color: Color with which the object will be represented in the visualization.

class continuousItems.**Wall**(*block1*, *block2*, *block3*, *color=None*)

Class that implements building walls.

Attributes:

block1, block2, block3: lists of positions that contain positions between which an occupant can move obeying with the impenetrability of the wall.

color: Color with which the object will be represented in the visualization.

4.3.3 Items in simplified modeling

In the file continuousItems.py three classes are defined to implement the elements of the physical space in a simplified model based on a room distribution:

-Room: Class that implements the rooms through which the Agent/Ocupant objects are located, move and where activities are carried out. -Door: Class that implements bulding plane doors. -Wall: Class that implements building walls.

class roomsItems.**Door**(*room1=False*, *room2=False*, *state=False*)

Class that implements bulding plane doors.

Attributes: state: Door status, open (True) or closed (False). room1: First room to croos the door. room2: Second room to croos the door.

Methods: open: Change the status of the door to open. close: Change the status of the door to close.

close()
Change the status of the door to closed (False)

open()
Change the status of the door to open (True)

class roomsItems.**Room**(*name*, *conectedTo*, *dx*, *dy*, *pos=(0, 0)*)

Class that implements the rooms through which the Agent/Ocupant objects are located, move and where activities are car

Attributes: name: Unique name of the room. roomsConected: List of accessible rooms from this room. dx: Size in the ordinate x (meters). dy: Size in the ordinate y (meters). pos: Position of the room (x, y). agentsInRoom: List of agent objects in the room walls: List of Wall objects of the room. doors: List of Doors objects of the room.

class roomsItems.**Wall**(*room1=False*, *room2=False*)

Class that implements building walls.

Attributes: room1: First room to croos the door. room2: Second room to croos the door.

4.4 Launchers Module Documentation

4.4.1 Visual

4.4.2 Batch

Python Module Index

Index