Sensim: Sensor Extremelly Naturally Simulated In our Model (or just Sensor Simulation)

Simon Fernandez, Mathieu Vavrille

December 2018

Abstract

This project is a helper to create simulation of sensors. The goal is to model a place composed of many sensors (smart city, smart campus), and simulate the behavior of all the sensors. The implementation is done using Python (3), and the created language is an embedded language.

1 General model

The general model is divided into three main components:

- Simulation
- Display
- Sensor

There are also secondary components used to encapsulate data and time through the simulation : *Data* and *Timestamp*. These two classes will be used by the other components.

Figure 1 shows the links between the different components of the general model.

1.1 Simulation

The simulation is the core class that will run the simulation. In this class, the user should be able to define the speed at with he wants to run the simulation (for example run a simulation twice as fast as it should be). He will also define the sensors that will be used, and the display where whe wants to output the data.

Once the different components of the model are given, the user may start the simulation. The *Simulation* class will then handle time, ask the sensors for data, and send it to the given display.

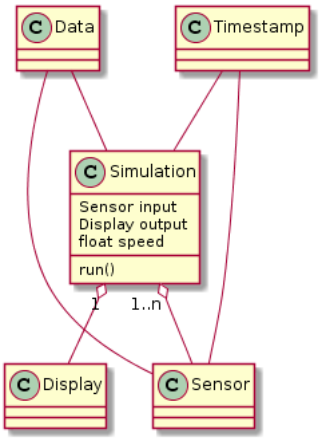

Figure 1: The general model

1.2 Display

This class will deal with the output of the simulation. Many different outputs can be defined: CSV, JSON formats, or outputing in an influxDB database (that can be then visualised with Grafana).

The Display is plugged to the simulation and will receive the generated data as soon as they are generated during the simulation.

1.3 Sensor

This is the most base class that will allow the user to define sensors. The model provides different kinds of predefined Sensors in order to help the user to easily build their simulation.

Even if different kinds of sensors exist, they must all behave the same way. Because of this, the Simulation does not have to know what kind of sensors it is handling.

1.4 Generator

Some sensors may be complex to build. If they are an aggregation of many sensors, or implement complex functions. In order to provide an easy way to build complex sensors, we made this Generator class. It can be seen as a Sensor in construction. The user will progressively give data to the Constructor, give it all the information it needs, and when all options are given, the Generator will output a Sensor behaving like the user wants.

This is why it is important for all Sensors to behave the same way : the user does not care how the Sensor behaves internaly, the important information is the data output. Generators are here to help build Sensors. But they are not mandatory, the user can build the sensors by hand.

1.5 Timestamp

The Timestamp class is here to provide a more friendly interface for time. There are different kinds of times : absolute dates, relative dates, durations, ... In order to provide a clean access to all these possibilities, the Timestamp class tries to overload all basic operations so that time can be smoothly used in user-defined functions. For example, when one wants a "function that is a second order polynomial in time", is time an absolute time (total seconds since Epoch) ? Is it time elapsed since the begining of the simulation ? Is it a function of the "hour" field of a date ?

The Timestamp allows to access all these informations and use them smoothly.

1.6 Data

The Data class will represent a point of data generated by a sensor. It encapsulates all informations that are needed to describe a piece of information. It contains the Timestamp when this data was generated, the name of the sensor that generated it, the value of the data, and all other field that may be used to describe the data.

This class will not be used by the user but it will be necessary for the simulation.

2 How to use the language

2.1 Creating the model

First, the user needs to create the set of sensors that are in their model. Each sensor has type `Sensor()`. This class is an abstract class. The user must use sub-classes that define different behaviours.

Each Sensor have a given name. If the name is not defined when building the sensor, a pre-generated unique name will be used.

There are three kinds of `Sensor(name)`.

2.1.1 `Importer(name)`

They are used to get data already generated and stored in different formats

- `JSONImporter(name, filename)` : Reads the JSON file named *filename* and outputs the data contained in it. It must contain the following fields:
  - "bn" : Name of the sensor
  - "e" : list of data : each piece of data contains two fields:
  - "t" : Timestamp when the data was generated
  - "v" : Value of the sensor at time "t"
- `csvImporter(name, filename)` : Reads the CSV file named *filename* and outputs the data contained in it. The first column must be the timestamp. The rest of the columns will be added in the custom fields of the *Data* object.

2.1.2 `ModelSensor(name, step)`

Generates data following a given behaviour. Data is generated on-the-fly. A value is generated after each *step* amount of time.

There are some pre-defined `ModelSensor()`.

- `FunctionSensor(name, step, function)` : $function$ is a python function that will be called on each time step. This allows to use Python functions, and external modules (for example, call the *np.sin* function from the numpy module).
- `PolynomialSensor(name, step, polynomial, coefs, points)` : a special case of `FunctionSensor`. It implements a polynomial function. There are three ways to define a polynomial sensor :
  - *polynomial*: give the polynomial function
  - $coefs$: give the array of the coefficients of the polynomial
  - *points*: give a list of (x, y) values, interpolates the polynomial fitting these points
- `MarkovChain()`: A Markov chain : use the `addState`, `addTransition` and `setStartNode` to define a custom chain. At each time step, it will compute a transition following the given probabilities, and send the new state.

2.1.3 Wrappers

Modify the behaviour of *sensor* it contains. It is a wrapper that can catch the returned data and apply operations on it.

There are three kinds of Wrappers for now:

- `MaskerSensor(name, sensor, start, stop)`: forwards the data of *sensor* only if the timestamp is between *start* and *stop*. This allows to "hide" the sensor during some time, or activate it only on a given period.
- `SpeedSensor(name, sensor, factor)`: scales the speed of *sensor* by factor $factor$. This allows to set a sensor to run twice as fast, or twice as slow.
- `AggregatedSensor(name, sensors)`: encapsulates a set of *sensors*. It gathers the data from all *sensors* and sends them, like a funnel.

2.2 User-friendly ways to define sensors

Some functions are added to help the user build Sensors. Each function returns the builded sensor, so they can be stacked. If s and s' are sensors,

- `s.named(name)` : changes the name of the sensor
- `s.turnedOnAt(time)`: disables the sensor before *time*

- `s.turnedOffAt(time)`: disables the sensor after *time*. *time* can be an absolute time, or a relative time (relative to the start time of the sensor)
- `s.turnedOnBetween(start, stop))`: both at the same time
- $s + s'$: builds a sensor sending the data of both sensors
- $s * n$ with n an integer : builds a sensor containing n copies of s

So, if s models the sensor of a parking slot, $(s * 30).named(\text{"}parking\text{"})$ is a sensor containing 30 independent parking slots, and returning the data of each slot. This multiple sensor is called "parking"

2.3 Preparing the display devices

Displays the simulated data to a format where it will be visualised.

- `InfluxDBDisplay(url)` : outputs the data to an influxDB data base running at url *url*
- `JSONDisplay(filename)` : outputs the data in a JSON file called $filename$

2.4 Creating the simulation

The `Simulation` object is built with all the previous objects: `Simulation(name, display, sensors, spee`

- *name* : name of the simulation
- *display* : the display where to export the data
- *sensors* : list of the sensors to simulate
- *speed* : speed factor to speed up or slow down the whole simulation
- *start* : timestamp when the simulation starts
- *stop* : timestamp when the simulation ends
- *realtime* : if $True$, the simulation will wait and make sure the data is generated at the good timestamp, nothing will be pre computed.

Once the simulation *simu* is defined, a simple $simu.run()$ will start the simulation.

2.5 Examples

The easiest way to discover how to build the simulation is to look at the examples given in the folder, they depict most of the functionnalities.

3 Concrete implementation

In this section, we will present how we implemented everything. The presentation will be done class by class. All the functions that are prefixed by an underscore should never be used by the user (for example `_popNext` or `_setup`). These functions are auxilliary functions that are used by the simulation to generate the data. This convention follows the PEP8 convention of python, as python does not give a way to protect methods.

3.1 Timestamp

The Timestamp object has many features. We based our implementation on the `datetime` and `dateutil` Python packages. They provide base operations and functions on time and dates. The Timestamp class is a simple wrapper around these packages. The major modification is the overloading of the different operators. In order to provide a user-friendly interface, we overloaded the operators to allow comparison, addition, substraction with strings. For example, one can write `t2 = t1 + "2h45m"`. We found a simple parser to use this kind of notations.

3.2 Data

The Data object is a simple dictionnary. It must have three informations : `timestamp`, `sensor`, `value`. Then, all additionnal field is added to the `__dict__` attribute. This way, the sensors can put all kind of information in the Data object.

3.3 Simulation

The Simulation works in three phases :

- Setup : Sets all internal parameters, calls `_setup()` on all sensors and displays, sets the start and stop time of the simulation
- Run : A simple loop : while there is still data to get and we are before the stop time, pop the next Data.
- Stop : When the simulation ends, close the displays.

3.4 Sensors

The main class that need to be understood first is the class of the sensors. The idea is to have sensors defined by different manners that will behave the same way. For example, from the simulation class, it will be impossible to know if a data comes from data imported from a file, or from a Markov Chain. The Sensor class is an abstract class that will define the functions that need to be implemented in order to get this behaviour.

3.4.1 The Sensor abstract class

The main implementation choice that was made was the one to know how we would generate the data, and send it to the simulation. This is done using a method `_popNext` that will return the next data generated by the sensor. The associated function is `_getNext` that will return the next data that will be generated, but this data is not meant to be used right now (the data can be used to know beforehand when will be the next data generated).

Each sensor must also implement a `_advanceTime` function, called after a `_popNext`, this moves the sensor inner clock to the next data that will be returned

Above that, a sensor has a `name` attribute, with the associated setter (`setName`). If the name is not defined by the user, it will be generated automatically as "Sensor_i" where i will be incremented each time a new name is defined (to prevent from two sensors with the same name).

Now we should look deeper into the different classes that will inherit from the Sensor class

3.4.2 The modelSensor abstract class

This class is an abstract class that represents a sensor modelled by some law. For example, the user may want to define a sensor that will behave like a Markov Chain, or a like a given function. The inherited class will represent a different model. There is an attribute defined by the modelSensor class that is the step of the created sensor. The step attribute will define at which speed the sensor will send data (for example, a computer energy consumption can be received each millisecond, whereas sensor from a parking lot will only send data every second, or minute).

Markov Chains One model that is implemented is the one of Markov Chains. The state returned by the sensor is following the low of a Markov Chain. To define a Markov Chain, the user need to define the nodes, the transitions and the starting node. The nodes can be named with integers or strings or any type, and it is these names that will be returned as the value of the sensor. Setting the transition is harder: the implementation choosed to give a method `addTransition(node1, node2, proba)`. The user has to define all the values of the transitions (the initial probability is 0).

We are working with probabilities, and then the sum of the probabilities of transition should sum up to one. We enforce this fact by not looking at the last transition for each node, because we know that it is one minus the sum of the other probabilities. This way, we prevent the user from defining bad transitions, and locking the simulation.

FunctionSensor class This class will also behave like a sensor, but will generate data from a given function. The function is an attribute of the instance of the class, and there is the setter associated to it.

Polynomial sensor A special type of FunctionSensor. It implements a polynomial function of the time. It can be built from a polynomial, a set of coefficients or a set of points from which we interpolate the polynomial. See previous section for useage.

Masked Sensor A Wrapper around a sensor. Calls the sub-sensor, but forwards its data only if the timestamp is in in a valid segment.

Speed Sensor A Wrapper around a sensor. Calls the sub-sensor, and scales the timestamp of the returned data. This way, it acts as if the sub-sensor runds at a different speed.

3.4.3 The Importer abstract class

The importer class is also an abstract class that implements the function to read text from an input file, and the functions to get the next data (`_popNext` and `_getNext`). The user have to define a filename at the creation of the class. The real importation of the data (depending on the type of storage of the data) is done in inherited classes, during the setup phase. During this phase, the inherited classes will have to store the data in the attribute `data`, in a `deque` (which is a list where we can pop the first element in constant time), where the timestamps are sorted in increasing order. Then the methods `_popNext` and `_getNext` will do the interface with the simulation (by popping data from the `deque`).

The JsonImporter class To import a JSON file, we need to know what will be the attributes of the main dictionnary. I use this format, as it is suggested in the subject of the project. We suppose that:

- the main data is a dictionary
- there is a key `"bn"` and the value is the name of the sensor
- there is a key `"e"` and the value is the data, as a list of dictionaries, one for each data
- each data is a dictionary with:
  - a key `"t"` and the value is the timestamp
  - a key `"v"` and the value is the data of the sensor

This assumptions are done to be able to import the data that were suggested in the subject.

This might be too restrictive, see section **??** for a proposition of improvement.

The `_setup` function will instanciate the `data` attribute, that will be sent to the simulation. We use the `json` python package to import the json file.

The CsvImporter class This class will do the same as the previous one, but for CSV files. We use the `csv` python package to import the file. The assumptions for this format are the following:

- the first line contains the name of the sensors that are in the files
- the next lines contain as a first value the timestamp when the data was generated, and then the other data for each sensor (on the same column as the name of the sensor in the first line)

These conditions are less restrictive than for the JSON format, but they are still conditions and we can debate about how to remove them.

One tricky part is that the data in a CSV file is only plain text, (compared to JSON where we know when a data is an integer or a string). To find what is the type of the data, we try to cast it as an `int`, if it fails we try as a `float` if it fails it will remain a `string`. The `_setup` function will import the data, find the type of every element, associate every measure to its sensor, sort by timestamp and put this in the `data` attribute, to be sent to the simulation.

3.4.4 The AggregatedSensor class

This class is one of the most important, as it will allow to group sensors. The aggregated sensor will work as if it was a single sensor (only one method `_popNext`), and it will choose among the different sensors that it contains what is the one that will generate data the soonest (and will return this one). The only method used by the user is the method `addSensors` that will add many sensors to the aggregated sensor.

It also allows to build complex sensors. For example, the multiplication of one sensor by an integer will return an aggregated sensor with multiple occurences of the sensor.By default, Python sends references of objects, so we must make sure to use `deepcopy` of sensors, to send new objects. Adding two sensors will also create an aggregated sensor containing the two sensors.

3.5 Display

The display is the third big block of the whole simulation. It will output data on different possible outputs. Currently, two types of output are implemented: display to an influxDB database, and to a file in JSON format.

3.5.1 The Display abstract class

This interface contains the methods `addPlot`, `_setup` and `end`. The method addPlot will add a data in the output. The method end will be called by the simulation when there is no more data to send; this is used by displays in files, because we don't want to open and close the file each time we add a data in the file. To do this, we just open the file at the end of the simulation, write everything, and close right after.

3.5.2 Display to an influxDB database

In order to visualize data with Grafana, we must send data to an influxDB data base. Because this database is accessible with POST requests at a given url, we simply use the `requests` package to send custom requests to an url.

This way, during setup, we send a request to delete previous databases, create the new one. And when the simulation generates a Data, we send the adapted POST request to the server. If the user wants to plot this data in Grafana, simply plot influxDB data the regular way.

3.5.3 Display to a file in JSON format

The `Data` class gives a way to export a data in a dictionary. The class JsonDisplay will store in a list all the data it receive, and at the call of the `_end` it will store it in the file selected.

The output file will contain the name of the database, the base time (time when the simulation was done), the unit of the time, and then the list of data.

3.6 Generator

They are like sensors in construction. They gather objects and information, and when calling the `build()` function, they output the sensor adapted to the given informations.

We implemented two Generators, but it is easy to build new ones (see possible extensions section) :

- `Interpolator` : builds a polynomial sensor from a set of points.
- `MultipleSensor` : builds an Aggregated Sensor containing multiple copies of a sensor. We must be careful because Python sends references by default, so we had to overload all `__deepcopy__` operations to make sure that `sensor * 10` builds 10 different sensors, and not 10 references to the same object

4 Possible improvements

This section presents the improvements that can be done to the project. The improvements presented are not implemented because of a lack of time, or knowledge of the domain. The improvements are presented component by component.

4.1 Markov Chain

The way to define Markov Chains is currently a little unintuitive for the user. The user adds the nodes, and the starting node (which is fine), but the issue comes when defining the transition matrix. The current implementation uses the function `addTransition()`.

This can be improved because it is tedious to define each transition by hand. One could improve it by giving a function `addOneTransition(node,transition)` where the

transition parameter is a dictionary containing the probability for each associated key. To be even faster, one could add a function to define the whole transition matrix in one time, with a dictionnary contaning the transition for each associated node.

The python dictionaries seems to be a good way to do it because they really show the fact that the keys and the values are associated. Then it will be translated to a real transition matrix, which is less user-friendly, but more programmer-friendly.

4.2 JsonImporter

The current implementation of the JsonImporter class does not give possibilities for other key names in the dictionary. The key `"bn"` have to contain the name of the sensor, as well as the other forced keys. This suppositions were done because we saw on examples that it was one way to represent data from sensors.

To improve this importer, one can allow the user to specify how is the file loaded. Json files have a tree structure where nodes are given by types, links of the tree are given by keys (of dictionaries) or indices (of lists), and leaves of the tree are data of the file. By specifying the structure of the tree, the user would be able to have personalized json files.

With that, we should have predefined tree structures (for example, the one that is already implemented). This way, one could implement all the main json structures for sensor outputs, and make them available to the user.

A Class diagrams

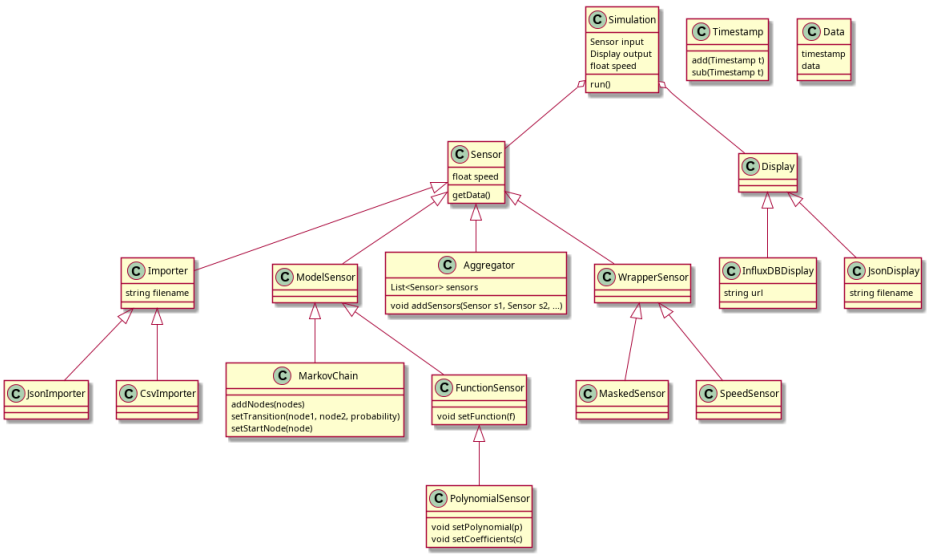

Figure 2: Class diagram showing the hierarchy of the project