agentMET4FOF Documentation

Bang Xiang Yong

Apr 29, 2020

Getting started:

agentMET4FOF is a Python library developed at the Institute for Manufacturing of the University of Cambridge (UK) as part of the European joint Research Project EMPIR 17IND12 Met4FoF.

For the *agentMET4FOF* homepage go to GitHub.

agentMET4FOF is written in Python 3.

CircleCI Documentation Status Codecov Badge

CHAPTER 1

Multi-Agent System for Metrology for Factory of the Future (Met4FoF) Code

This is supported by European Metrology Programme for Innovation and Research (EMPIR) under the project Metrology for the Factory of the Future (Met4FoF), project number 17IND12. (https://www.ptb.de/empir2018/met4fof/home/)

1.1 About

- How can metrological input be incorporated into an agent-based system for addressing uncertainty of machine learning in future manufacturing?
- Includes agent-based simulation and implementation
- Readthedocs documentation is available at (https://agentmet4fof.readthedocs.io)

1.2 Use agentMET4FOF

The easiest way to get started with *agentMET4FOF* is navigating to the folder in which you want to create a virtual Python environment (*venv*), create one based on Python 3.6 or later, activate it, first install numpy, then install *agentMET4FOF* from PyPI.org and then work through the tutorials or examples. To do this, issue the following commands on your Shell:

```
$ cd /LOCAL/PATH/TO/ENVS
$ python3 -m venv agentMET4FOF_venv
$ source agentMET4FOF_venv/bin/activate
(agentMET4FOF_venv) $ pip install numpy
Collecting numpy
...
Successfully installed numpy-...
(agentMET4FOF_venv) $ pip install agentMET4FOF
```

(continues on next page)

(continued from previous page)

```
Collecting agentMET4FOF
...
Successfully installed agentMET4FOF-... ...
(agentMET4FOF_venv) $ python
Python ... (default, ..., ...)
[GCC ...] on ...
Type "help", "copyright", "credits" or "license" for more information.
>>> from agentMET4FOF_tutorials import tutorial_1_generator_agent
>>> tutorial_1_generator_agent.demonstrate_generator_agent_use()
Starting NameServer...
Broadcast server running on 0.0.0.0:9091
NS running on 127.0.0.1:3333 (127.0.0.1)
URI = PYRO:Pyro.NameServer@127.0.0.1:3333
INFO [2020-02-21 19:04:26.961014] (AgentController): INITIALIZED
INFO [2020-02-21 19:04:27.032258] (Logger): INITIALIZED
 * Serving Flask app "agentMET4FOF.dashboard.Dashboard" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
...
```

Now you can visit `http://127.0.0.1:8050/` with any Browser and watch the SineGenerator agent you just spawned.

To get some insights and really get going please visit agentMET4FOF.readthedocs.io .

1.3 Get started developing

First clone the repository to your local machine as described here. To get started with your present *Anaconda* installation just go to *Anaconda prompt*, navigate to your local clone

```
cd /LOCAL/PATH/TO/agentMET4FOF
```

and execute

```
conda env create --file environment.yml
```

This will create an *Anaconda* virtual environment with all dependencies satisfied. If you don't have *Anaconda* installed already follow this guide first, then create the virtual environment as stated above and then proceed.

Alternatively, for non-conda environments, you can install the dependencies using pip

```
pip install -r requirements.txt
```

First take a look at the tutorials and examples or start hacking if you already are familiar with agentMET4FOF and want to customize your agents' network.

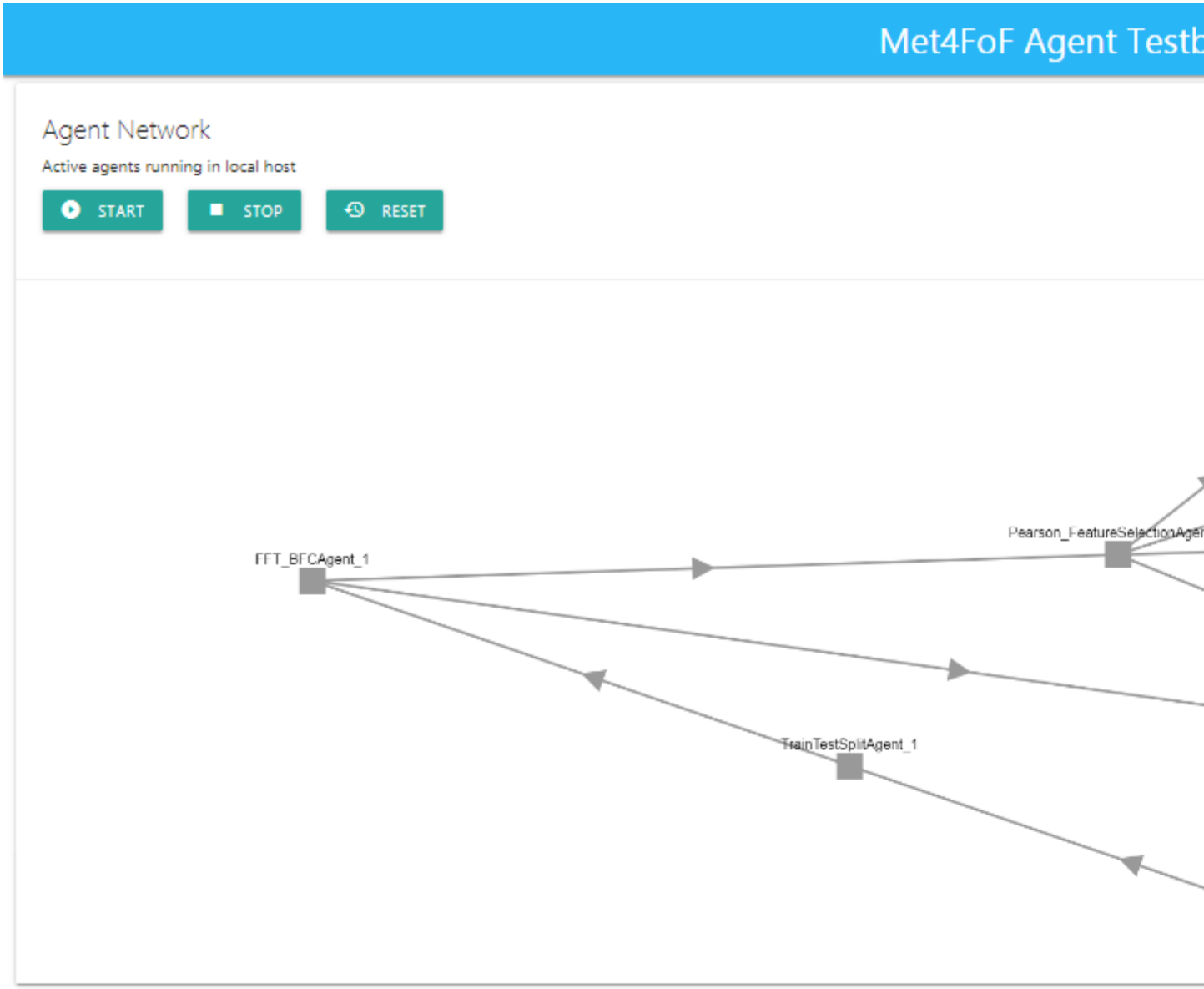
Alternatively, watch the tutorial webinar here

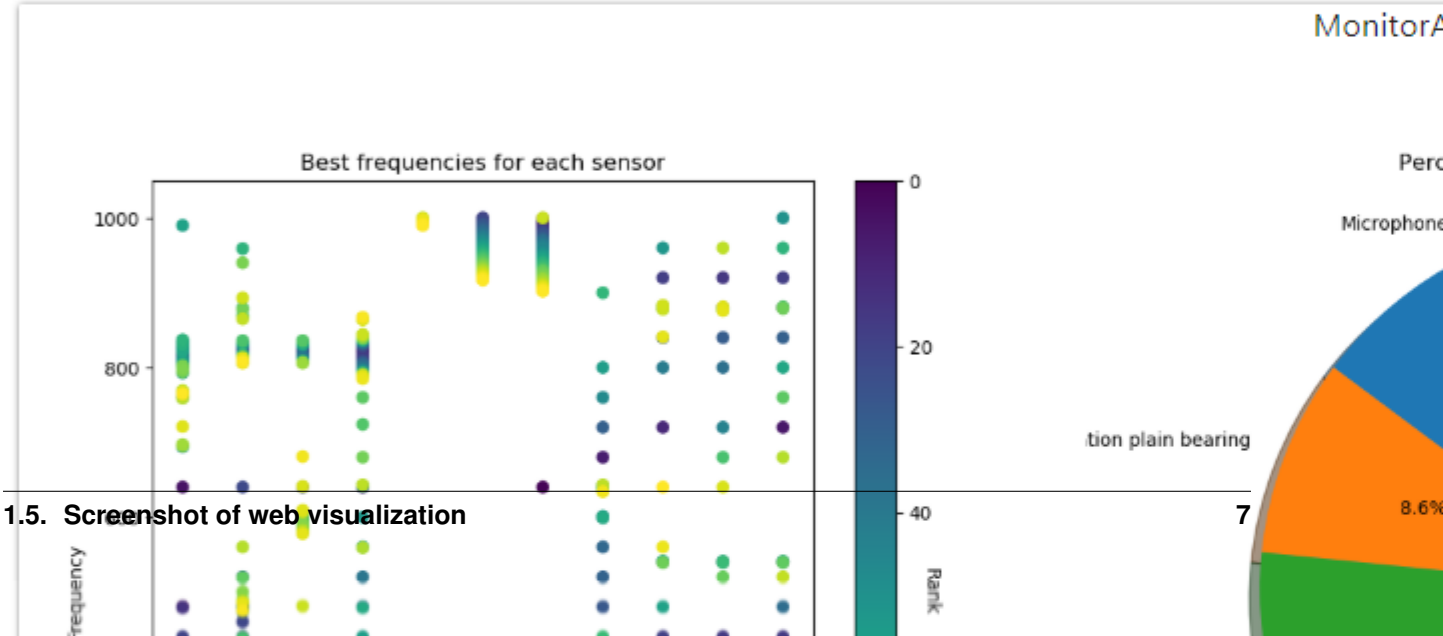
1.4 Updates

- Implemented base class AgentMET4FOF with built-in agent classes DataStreamAgent, MonitorAgent

- Implemented class AgentNetwork to start or connect to a agent server
- Implemented with ZEMA prognosis of Electromechanical cylinder data set as use case DOI
- Implemented interactive web application with user interface

1.5 Screenshot of web visualization





1.6 Note

- In the event of agents not terminating cleanly, run

```
taskkill /f /im python.exe /t
```

  in Windows Command Prompt to terminate all background python processes.

CHAPTER 2

Tutorial 1 - A simple pipeline to plot a signal

First we define a simple pipeline of two agents, of which one will generate a signal (in our case a *SineGeneratorAgent*) and the other one plots the signal on the dashboard (this is always a *MonitorAgent*).

We define a *SineGeneratorAgent* for which we have to override the functions `init_parameters()` & `agent_loop()` to define the new agent's behaviour.

- `init_parameters()` is used to setup the input data stream and potentially other necessary parameters.
- `agent_loop()` will be endlessly repeated until further notice. It will sample by sample extract the input data stream's content and push it to all agents connected to *SineGeneratorAgent*'s output channel by invoking `send_output()`.

The *MonitorAgent* is connected to the *SineGeneratorAgent*'s output channel and per default automatically plots the output.

Each agent has an internal `current_state` which can be used as a switch to change the behaviour of the agent. The available states are listed here.

As soon as all agents are initialized and the connections are set up, the agent network is started by accordingly changing all agents' state simultaneously.

[1]:
```
# %load tutorial_1_generator_agent.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator


class SineGeneratorAgent(AgentMET4FOF):
    """An agent streaming a sine signal

    Takes samples from the :py:mod:`SineGenerator` and pushes them sample by sample
    to connected agents via its output channel.
    """
    _sine_stream: SineGenerator

    def init_parameters(self):
        """Initialize the input data
```

(continues on next page)

(continued from previous page)

```
        Initialize the input data stream as an instance of the
        :py:mod:`SineGenerator` class
        """
        self._sine_stream = SineGenerator()

    def agent_loop(self):
        """Model the agent's behaviour

        On state *Running* the agent will extract sample by sample the input data
        streams content and push it via invoking :py:method:`AgentMET4FOF.send_output`.
        """
        if self.current_state == "Running":
            sine_data = self._sine_stream.next_sample()  # dictionary
            self.send_output(sine_data["x"])


def demonstrate_generator_agent_use():
    # Start agent network server.
    agent_network = AgentNetwork()

    # Initialize agents by adding them to the agent network.
    gen_agent = agent_network.add_agent(agentType=SineGeneratorAgent)
    monitor_agent = agent_network.add_agent(agentType=MonitorAgent)

    # Interconnect agents by either way:
    # 1) by agent network.bind_agents(source, target).
    agent_network.bind_agents(gen_agent, monitor_agent)

    # 2) by the agent.bind_output().
    gen_agent.bind_output(monitor_agent)

    # Set all agents' states to "Running".
    agent_network.set_running_state()

    # Allow for shutting down the network after execution
    return agent_network


if __name__ == "__main__":
    demonstrate_generator_agent_use()
```

```
Starting NameServer...
Broadcast server running on 0.0.0.0:9091
NS running on 127.0.0.1:3333 (127.0.0.1)
URI = PYRO:Pyro.NameServer@127.0.0.1:3333
INFO [2020-04-24 09:21:23.002156] (AgentController): INITIALIZED
INFO [2020-04-24 09:21:23.200110] (SineGeneratorAgent_1): INITIALIZED
 * Serving Flask app "agentMET4FOF.dashboard.Dashboard" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
INFO [2020-04-24 09:21:23.294149] (MonitorAgent_1): INITIALIZED
[2020-04-24 09:21:23.360214] (SineGeneratorAgent_1): Connected output module:␣
↪MonitorAgent_1
```

(continues on next page)

(continued from previous page)

```
SET STATE:   Running
[2020-04-24 09:21:24.218709] (SineGeneratorAgent_1): Pack time: 0.000897
[2020-04-24 09:21:24.223207] (SineGeneratorAgent_1): Sending: [0.]
[2020-04-24 09:21:24.225895] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.]), 'senderType': 'SineGeneratorAgent', 'channel': 'default'}
[2020-04-24 09:21:24.226708] (MonitorAgent_1): Tproc: 0.00017
[2020-04-24 09:21:25.217698] (SineGeneratorAgent_1): Pack time: 0.000419
[2020-04-24 09:21:25.220045] (SineGeneratorAgent_1): Sending: [0.47942554]
[2020-04-24 09:21:25.221406] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.47942554]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:25.223795] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554])}
[2020-04-24 09:21:25.224696] (MonitorAgent_1): Tproc: 0.00264
[2020-04-24 09:21:26.217785] (SineGeneratorAgent_1): Pack time: 0.000415
[2020-04-24 09:21:26.218995] (SineGeneratorAgent_1): Sending: [0.84147098]
[2020-04-24 09:21:26.220107] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.84147098]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:26.222038] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554, 0.84147098])}
[2020-04-24 09:21:26.223223] (MonitorAgent_1): Tproc: 0.002481
[2020-04-24 09:21:27.216058] (SineGeneratorAgent_1): Pack time: 0.000131
[2020-04-24 09:21:27.216323] (SineGeneratorAgent_1): Sending: [0.99749499]
[2020-04-24 09:21:27.216876] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.99749499]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:27.217220] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554, 0.84147098, 0.99749499])}
[2020-04-24 09:21:27.217288] (MonitorAgent_1): Tproc: 0.000314
[2020-04-24 09:21:28.215905] (SineGeneratorAgent_1): Pack time: 0.000102
[2020-04-24 09:21:28.216367] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.90929743]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-04-24 09:21:28.216186] (SineGeneratorAgent_1): Sending: [0.90929743]
[2020-04-24 09:21:28.216623] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554, 0.84147098, 0.99749499, 0.90929743])}
[2020-04-24 09:21:28.216678] (MonitorAgent_1): Tproc: 0.000229
```

```
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
```

```
<Figure size 432x288 with 0 Axes>
```

CHAPTER 3

Tutorial 2 - A simple pipeline with signal postprocessing.

Here we demonstrate how to build a *MathAgent* as an intermediate to process the *SineGeneratorAgent*'s output before plotting. Subsequently, a *MultiMathAgent* is built to show the ability to send a dictionary of multiple fields to the recipient. We provide a custom `on_received_message()` function, which is called every time a message is received from input agents.

The received message is a dictionary of the form:

```
{
'from':agent_name,
'data': data,
'senderType': agent_class_name,
'channel':'channel_name'
}
```

By default, `'channel'` is set to `"default"`, however a custom channel can be set when needed, which is demonstrated in the next tutorial.

[1]:
```
# %load tutorial_2_math_agent.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator


# Define simple math functions.
def divide_by_two(numerator: float) -> float:
    return numerator / 2


def minus(minuend: float, subtrahend: float) -> float:
    return minuend - subtrahend


def plus(summand_1: float, summand_2: float) -> float:
    return summand_1+summand_2
```

(continues on next page)

(continued from previous page)

```
class MathAgent(AgentMET4FOF):
    def on_received_message(self, message):
        data = divide_by_two(message['data'])
        self.send_output(data)

class MultiMathAgent(AgentMET4FOF):
    def init_parameters(self,minus_param=0.5,plus_param=0.5):
        self.minus_param = minus_param
        self.plus_param = plus_param

    def on_received_message(self, message):
        minus_data = minus(message['data'], self.minus_param)
        plus_data = plus(message['data'], self.plus_param)

        self.send_output({'minus':minus_data,'plus':plus_data})

class SineGeneratorAgent(AgentMET4FOF):
    def init_parameters(self):
        self.stream = SineGenerator()

    def agent_loop(self):
        if self.current_state == "Running":
            sine_data = self.stream.next_sample() #dictionary
            self.send_output(sine_data['x'])


def main():
    # start agent network server
    agentNetwork = AgentNetwork()
    # init agents
    gen_agent = agentNetwork.add_agent(agentType=SineGeneratorAgent)
    math_agent = agentNetwork.add_agent(agentType=MathAgent)
    multi_math_agent = agentNetwork.add_agent(agentType=MultiMathAgent)
    monitor_agent = agentNetwork.add_agent(agentType=MonitorAgent)
    # connect agents : We can connect multiple agents to any particular agent
    agentNetwork.bind_agents(gen_agent, math_agent)
    agentNetwork.bind_agents(gen_agent, multi_math_agent)
    # connect
    agentNetwork.bind_agents(gen_agent, monitor_agent)
    agentNetwork.bind_agents(math_agent, monitor_agent)
    agentNetwork.bind_agents(multi_math_agent, monitor_agent)
    # set all agents states to "Running"
    agentNetwork.set_running_state()

    # allow for shutting down the network after execution
    return agentNetwork


if __name__ == '__main__':
    main()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-1-e91ecde3d431> in <module>
      1 # %load tutorial_2_math_agent.py
----> 2 from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
      3 from agentMET4FOF.streams import SineGenerator
      4
      5

ModuleNotFoundError: No module named 'agentMET4FOF'
```

CHAPTER 4

Tutorial 3 - An advanced pipeline with multichannel signals.

We can use different channels for the receiver to handle specifically each channel name. This can be useful for example in splitting train and test channels in machine learning Then, the user will need to implement specific handling of each channel in the receiving agent.

In this example, the *MultiGeneratorAgent* is used to send two different types of data - Sine and Cosine generator. This is done via specifying `send_output (channel="sine")` and `send_output(channel="cosine")`.

Then on the receiving end, the `on_received_message()` function checks for `message['channel']` to handle it separately.

Note that by default, *MonitorAgent* is only subscribed to the `"default"` channel. Hence it will not respond to the `"cosine"` and `"sine"` channel.

[1]:
```
# %load tutorial_3_multi_channel.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator, CosineGenerator


def minus(data, minus_val):
    return data-minus_val


def plus(data,plus_val):
    return data+plus_val


class MultiGeneratorAgent(AgentMET4FOF):
    def init_parameters(self):
        self.sine_stream = SineGenerator()
        self.cos_stream = CosineGenerator()

    def agent_loop(self):
        if self.current_state == "Running":
            sine_data = self.sine_stream.next_sample() #dictionary
            cosine_data = self.sine_stream.next_sample() #dictionary
```

(continues on next page)

(continued from previous page)

```
            self.send_output(sine_data['x'], channel="sine")
            self.send_output(cosine_data['x'], channel="cosine")


class MultiOutputMathAgent(AgentMET4FOF):
    def init_parameters(self,minus_param=0.5,plus_param=0.5):
        self.minus_param = minus_param
        self.plus_param = plus_param

    def on_received_message(self, message):
        """
        Checks for message['channel'] and handles them separately
        Acceptable channels are "cosine" and "sine"
        """
        if message['channel'] == "cosine":
            minus_data = minus(message['data'], self.minus_param)
            self.send_output({'cosine_minus':minus_data})
        elif message['channel'] == 'sine':
            plus_data = plus(message['data'], self.plus_param)
            self.send_output({'sine_plus':plus_data})


def main():
    # start agent network server
    agentNetwork = AgentNetwork()
    # init agents
    gen_agent = agentNetwork.add_agent(agentType=MultiGeneratorAgent)
    multi_math_agent = agentNetwork.add_agent(agentType=MultiOutputMathAgent)
    monitor_agent = agentNetwork.add_agent(agentType=MonitorAgent)
    # connect agents : We can connect multiple agents to any particular agent
    # However the agent needs to implement handling multiple inputs
    agentNetwork.bind_agents(gen_agent, multi_math_agent)
    agentNetwork.bind_agents(gen_agent, monitor_agent)
    agentNetwork.bind_agents(multi_math_agent, monitor_agent)
    # set all agents states to "Running"
    agentNetwork.set_running_state()

    # allow for shutting down the network after execution
    return agentNetwork


if __name__ == '__main__':
    main()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-90d955fea48f> in <module>
      1 # %load tutorial_3_multi_channel.py
----> 2 from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
      3 from agentMET4FOF.streams import SineGenerator, CosineGenerator
      4
      5
```

(continues on next page)

(continued from previous page)

```
ModuleNotFoundError: No module named 'agentMET4FOF'
```

CHAPTER 5

agentMET4FOF agents

class agentMET4FOF.agents.**AgentMET4FOF**(*name=''*, *host=None*, *serializer=None*, *transport=None*, *attributes=None*)

Base class for all agents with specific functions to be overridden/supplied by user.

Behavioural functions for users to provide are init_parameters, agent_loop and on_received_message. Communicative functions are bind_output, unbind_output and send_output.

agent_loop()

User defined method for the agent to execute for *loop_wait* seconds specified either in *self.loop_wait* or explicitly via‘init_agent_loop(loop_wait)‘

To start a new loop, call *init_agent_loop(loop_wait)* on the agent Example of usage is to check the *current_state* of the agent and send data periodically

before_loop()

This action is executed before initiating the loop

bind_output(*output_agent*)

Forms Output connection with another agent. Any call on send_output will reach this newly binded agent

Adds the agent to its list of Outputs.

Parameters **output_agent** (*AgentMET4FOF or list*) – Agent(s) to be binded to this agent's output channel

convert_to_plotly(*matplotlib_fig*)

Internal method to convert matplotlib figure to plotly figure

Parameters **matplotlib_fig** (*plt.Figure*) – Matplotlib figure to be converted

handle_process_data(*message*)

Internal method to handle incoming message before calling user-defined on_received_message method.

If current_state is either Stop or Reset, it will terminate early before entering on_received_message

init_agent_loop(*loop_wait=1.0*)

Initiates the agent loop, which iterates every‘loop_wait‘ seconds

Stops every timers and initiate a new loop.

Parameters **loop_wait** (*int*) – The wait between each iteration of the loop

init_parameters()
User provided function to initialize parameters of choice.

log_info(*message*)
Prints logs to be saved into logfile with Logger Agent

Parameters **message** (*str*) – Message to be logged to the internal Logger Agent

on_init()
Internal initialization to setup the agent: mainly on setting the dictionary of Inputs, Outputs, PubAddr.

Calls user-defined *init_parameters()* upon finishing.

Inputs
Dictionary of Agents connected to its input channels. Messages will arrive from agents in this dictionary. Automatically updated when *bind_output()* function is called
Type dict

Outputs
Dictionary of Agents connected to its output channels. Messages will be sent to agents in this dictionary. Automatically updated when *bind_output()* function is called
Type dict

PubAddr_alias
Name of Publish address socket
Type str

PubAddr
Publish address socket handle
Type str

AgentType
Name of class
Type str

current_state
Current state of agent. Can be used to define different states of operation such as "Running", "Idle, "Stop", etc.. Users will need to define their own flow of handling each type of *self.current_state* in the *agent_loop*
Type str

loop_wait
The interval to wait between loop. Call *init_agent_loop* to restart the timer or set the value of loop_wait in *init_parameters* when necessary.
Type int

memory_buffer_size
The total number of elements to be stored in the agent *memory* When total elements exceeds this number, the latest elements will be replaced with the incoming data elements
Type int

on_received_message(*message*)
User-defined method and is triggered to handle the message passed by Input.

Parameters **message** (*Dictionary*) – The message received is in form {'from':agent_name, 'data': data, 'senderType': agent_class, 'channel':channel_name} agent_name is the name of the Input agent which sent the message data is the actual content of the message

pack_data(*data*, *channel='default'*)
Internal method to pack the data content into a dictionary before sending out.

Special case : if the *data* is already a *message*, then the *from* and *senderType* will be altered to this agent, without altering the *data* and *channel* within the message this is used for more succinct data processing and passing.

Parameters

- **data** (*argument*) – Data content to be packed before sending out to agents.
- **channel** (*str*) – Key of dictionary which stores data

Returns **Packed message data**

Return type dict of the form {'from':agent_name, 'data': data, 'senderType': agent_class, 'channel':channel_name}.

reset()
This method will be called on all agents when the global *reset_agents* is called by the AgentNetwork and when the Reset button is clicked on the dashboard.

Method to reset the agent's states and parameters. User can override this method to reset the specific parameters.

send_output(*data*, *channel='default'*)
Sends message data to all connected agents in self.Outputs.

Output connection can first be formed by calling bind_output. By default calls pack_data(data) before sending out. Can specify specific channel as opposed to 'default' channel.

Parameters

- **data** (*argument*) – Data content to be sent out
- **channel** (*str*) – Key of *message* dictionary which stores data

Returns **message**

Return type dict of the form {'from':agent_name, 'data': data, 'senderType': agent_class, 'channel':channel_name}.

send_plot(*fig=<Figure size 640x480 with 0 Axes>*)
Sends plot to agents connected to this agent's Output channel.

This method is different from send_output which will be sent to through the 'plot' channel to be handled.

Parameters **fig** (*Figure*) – Can be either matplotlib figure or plotly figure

Returns **The message format is {'from'**

Return type agent_name, 'plot': data, 'senderType': agent_class}.

stop_agent_loop()
Stops agent_loop from running. Note that the agent will still be responding to messages

unbind_output(*output_agent*)
Remove existing output connection with another agent. This reverses the bind_output method

Parameters **output_agent** (AgentMET4FOF) – Agent binded to this agent's output channel

update_data_memory(*message*)
Updates data stored in *self.memory* with the received message

Checks if sender agent has sent any message before If it did,then append, otherwise create new entry for it

Parameters **message** (*dict*) – Standard message format specified by AgentMET4FOF class

class agentMET4FOF.agents.**AgentNetwork**(*ip_addr='127.0.0.1'*, *port=3333*, *connect=False*, *log_filename='log_file.csv'*, *dashboard_modules=True*, *dashboard_update_interval=3*, *dashboard_max_monitors=10*, *dashboard_port=8050*)

Object for starting a new Agent Network or connect to an existing Agent Network specified by ip & port

Provides function to add agents, (un)bind agents, query agent network state, set global agent states Interfaces with an internal _AgentController which is hidden from user

add_agent(*name=''*, *agentType=<class 'agentMET4FOF.agents.AgentMET4FOF'>*, *log_mode=True*, *memory_buffer_size=1000000*, *ip_addr=None*)

Instantiates a new agent in the network.

Parameters

- **str** (*name*) – with the same name. Defaults to the agent's class name.
- **AgentMET4FOF** (*agentType*) – network. Defaults to *AgentMET4FOF*
- **bool** (*log_mode*) – Logger Agent. Defaults to *True*.

Returns **AgentMET4FOF**

Return type Newly instantiated agent

agents(*filter_agent=None*)

Returns all agent names connected to Agent Network.

Returns **list**

Return type names of all agents

bind_agents(*source*, *target*)

Binds two agents communication channel in a unidirectional manner from *source* Agent to *target* Agent

Any subsequent calls of *source.send_output()* will reach *target* Agent's message queue.

Parameters

- **source** (AgentMET4FOF) – Source agent whose Output channel will be binded to *target*
- **target** (AgentMET4FOF) – Target agent whose Input channel will be binded to *source*

connect(*ip_addr='127.0.0.1'*, *port=3333*, *verbose=True*)

Parameters

- **ip_addr** (*str*) – IP Address of server to connect to
- **port** (*int*) – Port of server to connect to

get_agent(*agent_name*)

Returns a particular agent connected to Agent Network.

Parameters **agent_name** (*str*) – Name of agent to search for in the network

set_agents_state(*filter_agent=None*, *state='Idle'*)

Blanket operation on all agents to set their *current_state* attribute to given state

Can be used to define different states of operation such as "Running", "Idle, "Stop", etc.. Users will need to define their own flow of handling each type of *self.current_state* in the *agent_loop*

Parameters

- **filter_agent** (*str*) – (Optional) Filter name of agents to set the states
- **state** (*str*) – State of agents to set

set_running_state(*filter_agent=None*)
Blanket operation on all agents to set their *current_state* attribute to "Running"

Users will need to define their own flow of handling each type of *self.current_state* in the *agent_loop*

Parameters **filter_agent** (*str*) – (Optional) Filter name of agents to set the states

set_stop_state(*filter_agent=None*)
Blanket operation on all agents to set their *current_state* attribute to "Stop"

Users will need to define their own flow of handling each type of *self.current_state* in the *agent_loop*

Parameters **filter_agent** (*str*) – (Optional) Filter name of agents to set the states

shutdown()
Shutdowns the entire agent network and all agents

start_server(*ip_addr='127.0.0.1'*, *port=3333*)

Parameters

- **ip_addr** (*str*) – IP Address of server to start
- **port** (*int*) – Port of server to start

unbind_agents(*source*, *target*)
Unbinds two agents communication channel in a unidirectional manner from *source* Agent to *target* Agent

This is the reverse of *bind_agents()*

Parameters

- **source** (AgentMET4FOF) – Source agent whose Output channel will be unbinded from *target*
- **target** (AgentMET4FOF) – Target agent whose Input channel will be unbinded from *source*

class agentMET4FOF.agents.**DataStreamAgent**(*name=''*, *host=None*, *serializer=None*, *transport=None*, *attributes=None*)
Able to simulate generation of datastream by loading a given DataStreamMET4FOF object.

Can be used in incremental training or batch training mode. To simulate batch training mode, set *pretrain_size=-1* , otherwise, set pretrain_size and batch_size for the respective See *DataStreamMET4FOF* on loading your own data set as a data stream.

agent_loop()
User defined method for the agent to execute for *loop_wait* seconds specified either in *self.loop_wait* or explicitly via'init_agent_loop(loop_wait)'

To start a new loop, call *init_agent_loop(loop_wait)* on the agent Example of usage is to check the *current_state* of the agent and send data periodically

init_parameters(*stream=<agentMET4FOF.streams.DataStreamMET4FOF object>*, *pretrain_size=None*, *batch_size=1*, *loop_wait=1*, *randomize=False*)

Parameters

- **stream** (DataStreamMET4FOF) – A DataStreamMET4FOF object which provides the sample data
- **pretrain_size** (*int*) – The number of sample data to send through in the first loop cycle, and subsequently, the batch_size will be used

- **batch_size** (*int*) – The number of sample data to send in every loop cycle
- **loop_wait** (*int*) – The duration to wait (seconds) at the end of each loop cycle before going into the next cycle
- **randomize** (*bool*) – Determines if the dataset should be shuffled before streaming

reset()
This method will be called on all agents when the global *reset_agents* is called by the AgentNetwork and when the Reset button is clicked on the dashboard.

Method to reset the agent's states and parameters. User can override this method to reset the specific parameters.

class agentMET4FOF.agents.**MonitorAgent**(*name=''*, *host=None*, *serializer=None*, *transport=None*, *attributes=None*)
Unique Agent for storing plots and data from messages received from input agents.

The dashboard searches for Monitor Agents' *memory* and *plots* to draw the graphs "plot" channel is used to receive base64 images from agents to plot on dashboard

memory
Dictionary of format *{agent1_name : agent1_data, agent2_name : agent2_data}*

Type dict

plots
Dictionary of format *{agent1_name : agent1_plot, agent2_name : agent2_plot}*

Type dict

plot_filter
List of keys to filter the 'data' upon receiving message to be saved into memory Used to specifically select only a few keys to be plotted

Type list of str

init_parameters(*plot_filter=[]*, *custom_plot_function=-1*, ***kwargs*)
User provided function to initialize parameters of choice.

on_received_message(*message*)
Handles incoming data from 'default' and 'plot' channels.

Stores 'default' data into *self.memory* and 'plot' data into *self.plots*

Parameters **message** (*dict*) – Acceptable channel values are 'default' or 'plot'

reset()
This method will be called on all agents when the global *reset_agents* is called by the AgentNetwork and when the Reset button is clicked on the dashboard.

Method to reset the agent's states and parameters. User can override this method to reset the specific parameters.

update_plot_memory(*message*)
Updates plot figures stored in *self.plots* with the received message

Parameters **message** (*dict*) – Standard message format specified by AgentMET4FOF class Message['data'] needs to be base64 image string and can be nested in dictionary for multiple plots Only the latest plot will be shown kept and does not keep a history of the plots.

class agentMET4FOF.agents.**TransformerAgent**(*name=''*, *host=None*, *serializer=None*, *transport=None*, *attributes=None*)

init_parameters(*method=None*, ***kwargs*)
User provided function to initialize parameters of choice.

on_received_message(*message*)
User-defined method and is triggered to handle the message passed by Input.

Parameters **message** (*Dictionary*) – The message received is in form {'from':agent_name, 'data': data, 'senderType': agent_class, 'channel':channel_name} agent_name is the name of the Input agent which sent the message data is the actual content of the message

CHAPTER 6

agentMET4FOF streams

class agentMET4FOF.streams.**CosineGenerator**(*num_cycles=1000*)

class agentMET4FOF.streams.**DataStreamMET4FOF**

Class for creating finite datastream for ML with *x* as inputs and *y* as target Data can be fetched sequentially using *next_sample()* or all at once *all_samples()*

For sensors data: The format shape for 2D data stream (num_samples, n_sensors) The format shape for 3D data stream (num_samples, sample_length , n_sensors)

all_samples()

Returns all the samples in the data stream

Returns samples

Return type dict of the form *{'x': current_sample_x, 'y': current_sample_y}*

next_sample(*batch_size=1*)

Fetches the samples from the data stream and advances the internal pointer *current_idx*

Parameters **batch_size** (*int*) – number of batches to get from data stream

Returns samples

Return type dict of the form *{'x': current_sample_x, 'y': current_sample_y}*

class agentMET4FOF.streams.**SineGenerator**(*num_cycles=1000*)

agentMET4FOF.streams.**extract_x_y**(*message*)

Extracts features & target from *message['data']* with expected structure such as :

1. tuple - (x,y)
2. dict - {'x':x_data,'y':y_data}

Handle data structures of dictionary to extract features & target

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

CHAPTER 8

References

Bibliography

[Bang2019] Bang X. Yong, A. Brintrup Multi Agent System for Machine Learning Under Uncertainty in Cyber Physical Manufacturing System, 9th Workshop on Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future

Python Module Index

a

Index