agentMET4FOF Documentation

Bang Xiang Yong

Sep 02, 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, then install *agentMET4FOF* from PyPI.org and then work through the tutorials or examples. At the moment there seems to be something wrong with one of our dependencies, so please install the specified dependency versions. We included the according step in the following guides.

1.2.1 Create a virtual environment on Windows

In your Windows PowerShell execute the following to set up a virtual environment in a folder of your choice and execute the first tutorial.

```
PS C:> cd C:\LOCAL\PATH\TO\ENVS
PS C:\LOCAL\PATH\TO\ENVS> py -3 -m venv agentMET4FOF_venv
PS C:\LOCAL\PATH\TO\ENVS> agentMET4FOF_venv\Scripts\activate
(agentMET4FOF_venv) PS C:\LOCAL\PATH\TO\ENVS> python -m pip install --upgrade pip␣
↪agentMET4FOF
Collecting agentMET4FOF
...
Successfully installed agentMET4FOF-... ...
(agentMET4FOF_venv) PS C:\LOCAL\PATH\TO\ENVS> python -m pip install --upgrade -r␣
↪requirements.txt
(agentMET4FOF_venv) PS C:\LOCAL\PATH\TO\ENVS> 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)
...
```

1.2.2 Create a virtual environment on Mac and Linux

In your terminal execute the following to set up a virtual environment in a folder of your choice and execute the first tutorial.

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

(continues on next page)

(continued from previous page)

```
 * 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)
...
```

1.2.3 Inspect dashboard

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
```

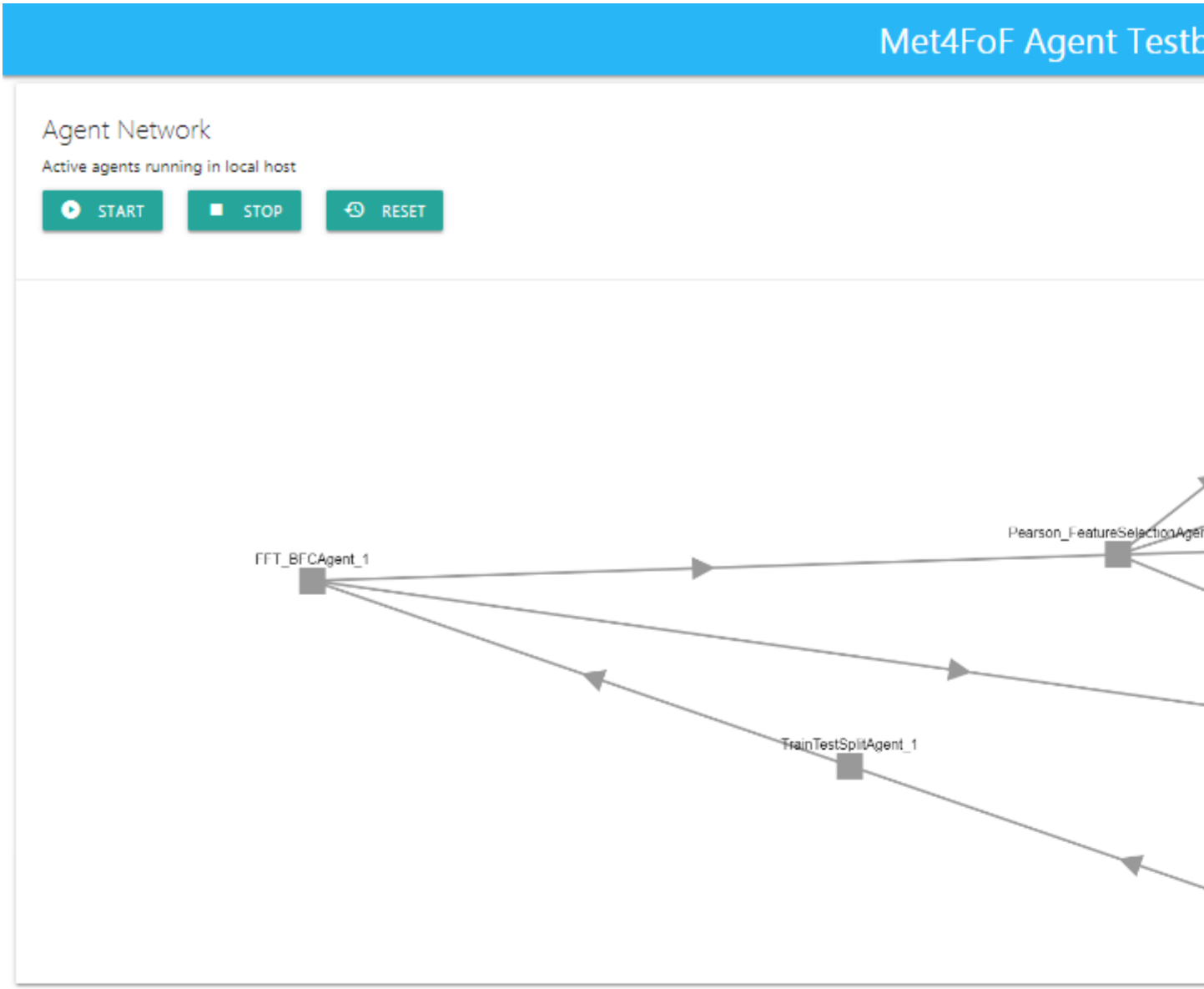
First take a look at the tutorials and examples or start hacking if you already are familiar with agentMET4FOF and want to customize your agents' network.

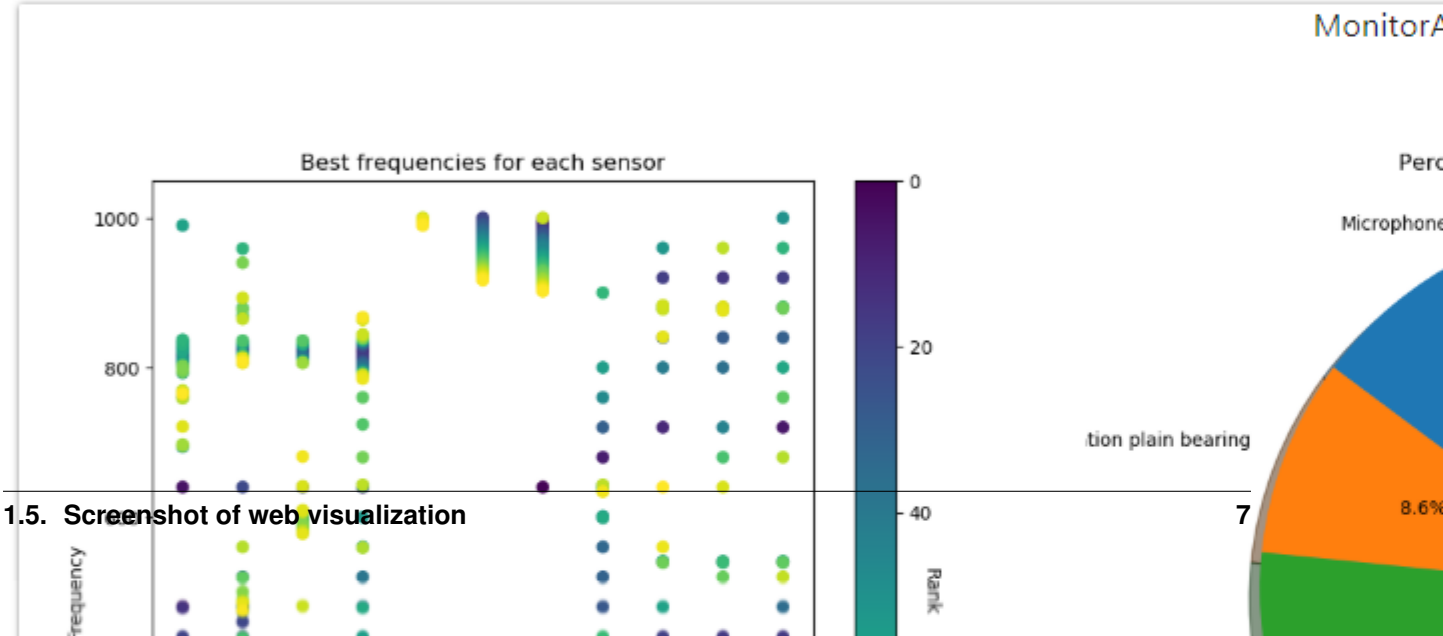
Alternatively, watch the tutorial webinar here

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 Orphaned processes

In the event of agents not terminating cleanly, you can end all Python processes running on your system (caution: the following commands affect **all** running Python processes, not just those that emerged from the agents).

1.6.1 Killing all Python processes in Windows

In your Windows command prompt execute the following to terminate all python processes.

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

1.6.2 Killing all Python processes on Mac and Linux

In your terminal execute the following to terminate all python processes.

```
$ pkill python
$
```

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.

[4]:
```
# %load tutorial_2_math_agent.py
from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork, MonitorAgent
from agentMET4FOF.streams import SineGenerator


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

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


class MultiMathAgent(AgentMET4FOF):
```

(continues on next page)

(continued from previous page)

```
    _minus_param: float
    _plus_param: float

    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 = self.minus(message["data"], self._minus_param)
        plus_data = self.plus(message["data"], self._plus_param)

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

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

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


class SineGeneratorAgent(AgentMET4FOF):

    _stream: SineGenerator

    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
```

(continues on next page)

(continued from previous page)

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

```
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 [2020-07-08 20:12:37.967786] (AgentController): INITIALIZED
Dash is running on http://127.0.0.1:8050/

 Warning: This is a development server. Do not use app.run_server
INFO [2020-07-08 20:12:38.108067] (SineGeneratorAgent_1): INITIALIZED
 in production, use a production WSGI server like gunicorn instead.

 * 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 [2020-07-08 20:12:38.185234] (MathAgent_1): INITIALIZED
INFO [2020-07-08 20:12:38.266143] (MultiMathAgent_1): INITIALIZED
INFO [2020-07-08 20:12:38.347055] (MonitorAgent_1): INITIALIZED
[2020-07-08 20:12:38.413035] (SineGeneratorAgent_1): Connected output module:␣
↪MathAgent_1
[2020-07-08 20:12:38.418991] (SineGeneratorAgent_1): Connected output module:␣
↪MultiMathAgent_1
[2020-07-08 20:12:38.425940] (SineGeneratorAgent_1): Connected output module:␣
↪MonitorAgent_1
[2020-07-08 20:12:38.465907] (MathAgent_1): Connected output module: MonitorAgent_1
[2020-07-08 20:12:38.538248] (MultiMathAgent_1): Connected output module:␣
↪MonitorAgent_1
SET STATE:   Running
[2020-07-08 20:12:39.125553] (SineGeneratorAgent_1): Pack time: 0.000893
[2020-07-08 20:12:39.128378] (SineGeneratorAgent_1): Sending: [0.]
[2020-07-08 20:12:39.134086] (MultiMathAgent_1): Received: {'from':
↪'SineGeneratorAgent_1', 'data': array([0.]), 'senderType': 'SineGeneratorAgent',
↪'channel': 'default'}
[2020-07-08 20:12:39.134202] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.]), 'senderType': 'SineGeneratorAgent', 'channel': 'default'}
[2020-07-08 20:12:39.135154] (MathAgent_1): Received: {'from': 'SineGeneratorAgent_1',
↪ 'data': array([0.]), 'senderType': 'SineGeneratorAgent', 'channel': 'default'}
[2020-07-08 20:12:39.137182] (MathAgent_1): Pack time: 0.001182
[2020-07-08 20:12:39.142027] (MultiMathAgent_1): Pack time: 0.00693
[2020-07-08 20:12:39.135086] (MonitorAgent_1): Tproc: 0.000153
[2020-07-08 20:12:39.138563] (MathAgent_1): Sending: [0.]
[2020-07-08 20:12:39.146332] (MultiMathAgent_1): Sending: {'minus': array([-0.5]),
↪'plus': array([0.5])}
[2020-07-08 20:12:39.140792] (MonitorAgent_1): Received: {'from': 'MultiMathAgent_1',
↪'data': {'minus': array([-0.5]), 'plus': array([0.5])}, 'senderType':
↪'MultiMathAgent', 'channel': 'default'}
[2020-07-08 20:12:39.140299] (MathAgent_1): Tproc: 0.00405
[2020-07-08 20:12:39.147118] (MultiMathAgent_1): Tproc: 0.012217
[2020-07-08 20:12:39.141319] (MonitorAgent_1): Tproc: 8.7e-05
[2020-07-08 20:12:39.143377] (MonitorAgent_1): Received: {'from': 'MathAgent_1', 'data
↪': array([0.]), 'senderType': 'MathAgent', 'channel': 'default'}
```

(continues on next page)

(continued from previous page)

```
[2020-07-08 20:12:39.144143] (MonitorAgent_1): Tproc: 4.8e-05
[2020-07-08 20:12:40.127624] (MultiMathAgent_1): Received: {'from':
↪'SineGeneratorAgent_1', 'data': array([0.47942554]), 'senderType':
↪'SineGeneratorAgent', 'channel': 'default'}
[2020-07-08 20:12:40.129917] (MultiMathAgent_1): Pack time: 0.000519
[2020-07-08 20:12:40.128351] (MathAgent_1): Received: {'from': 'SineGeneratorAgent_1',
↪ 'data': array([0.47942554]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-07-08 20:12:40.125310] (SineGeneratorAgent_1): Pack time: 0.000539
[2020-07-08 20:12:40.127820] (MonitorAgent_1): Received: {'from': 'SineGeneratorAgent_
↪1', 'data': array([0.47942554]), 'senderType': 'SineGeneratorAgent', 'channel':
↪'default'}
[2020-07-08 20:12:40.131738] (MultiMathAgent_1): Sending: {'minus': array([-0.
↪02057446]), 'plus': array([0.97942554])}
[2020-07-08 20:12:40.129241] (MathAgent_1): Pack time: 0.000428
[2020-07-08 20:12:40.127622] (SineGeneratorAgent_1): Sending: [0.47942554]
[2020-07-08 20:12:40.132038] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554]), 'MultiMathAgent_1': {'minus': array([-0.5]), 'plus
↪': array([0.5])}, 'MathAgent_1': array([0.])}
[2020-07-08 20:12:40.132147] (MultiMathAgent_1): Tproc: 0.004128
[2020-07-08 20:12:40.130428] (MathAgent_1): Sending: [0.23971277]
[2020-07-08 20:12:40.132367] (MonitorAgent_1): Tproc: 0.004143
[2020-07-08 20:12:40.135466] (MonitorAgent_1): Received: {'from': 'MathAgent_1', 'data
↪': array([0.23971277]), 'senderType': 'MathAgent', 'channel': 'default'}
[2020-07-08 20:12:40.130764] (MathAgent_1): Tproc: 0.002032
[2020-07-08 20:12:40.156668] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554]), 'MultiMathAgent_1': {'minus': array([-0.5]), 'plus
↪': array([0.5])}, 'MathAgent_1': array([0.        , 0.23971277])}
[2020-07-08 20:12:40.161541] (MonitorAgent_1): Tproc: 0.024927
[2020-07-08 20:12:40.164795] (MonitorAgent_1): Received: {'from': 'MultiMathAgent_1',
↪'data': {'minus': array([-0.02057446]), 'plus': array([0.97942554])}, 'senderType':
↪'MultiMathAgent', 'channel': 'default'}
[2020-07-08 20:12:40.171365] (MonitorAgent_1): Memory: {'SineGeneratorAgent_1':␣
↪array([0.        , 0.47942554]), 'MultiMathAgent_1': {'minus': array([-0.5       , -
↪0.02057446]), 'plus': array([0.5       , 0.97942554])}, 'MathAgent_1': array([0.   ␣
↪    , 0.23971277])}
[2020-07-08 20:12:40.171872] (MonitorAgent_1): Tproc: 0.006705
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
127.0.0.1 - - [08/Jul/2020 22:12:40] "POST /_dash-update-component HTTP/1.1" 200 -
127.0.0.1 - - [08/Jul/2020 22:12:40] "POST /_dash-update-component HTTP/1.1" 200 -
```

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


class MultiGeneratorAgent(AgentMET4FOF):

    _sine_stream: SineGenerator
    _cos_stream: CosineGenerator

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

(continues on next page)

(continued from previous page)

```
class MultiOutputMathAgent(AgentMET4FOF):

    _minus_param: float
    _plus_param: float

    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 = self.minus(message["data"], self._minus_param)
            self.send_output({"cosine_minus": minus_data})
        elif message["channel"] == "sine":
            plus_data = self.plus(message["data"], self._plus_param)
            self.send_output({"sine_plus": plus_data})

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

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


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()
```

```
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 [2020-07-08 20:15:03.999193] (AgentController): INITIALIZED
```

(continues on next page)

(continued from previous page)

```
INFO [2020-07-08 20:15:04.160762] (MultiGeneratorAgent_1): INITIALIZED
Dash is running on http://127.0.0.1:8050/

 Warning: This is a development server. Do not use app.run_server
 in production, use a production WSGI server like gunicorn instead.

 * 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 [2020-07-08 20:15:04.238194] (MultiOutputMathAgent_1): INITIALIZED
INFO [2020-07-08 20:15:04.307723] (MonitorAgent_1): INITIALIZED
[2020-07-08 20:15:04.364692] (MultiGeneratorAgent_1): Connected output module:␣
↪MultiOutputMathAgent_1
[2020-07-08 20:15:04.370956] (MultiGeneratorAgent_1): Connected output module:␣
↪MonitorAgent_1
[2020-07-08 20:15:04.379114] (MultiOutputMathAgent_1): Connected output module:␣
↪MonitorAgent_1
SET STATE:   Running
[2020-07-08 20:15:05.171751] (MultiGeneratorAgent_1): Pack time: 0.000315
[2020-07-08 20:15:05.172512] (MultiGeneratorAgent_1): Sending: [0.]
[2020-07-08 20:15:05.173200] (MonitorAgent_1): Received: {'from':
↪'MultiGeneratorAgent_1', 'data': array([0.]), 'senderType': 'MultiGeneratorAgent',
↪'channel': 'sine'}
[2020-07-08 20:15:05.175533] (MultiOutputMathAgent_1): Received: {'from':
↪'MultiGeneratorAgent_1', 'data': array([0.]), 'senderType': 'MultiGeneratorAgent',
↪'channel': 'sine'}
[2020-07-08 20:15:05.172748] (MultiGeneratorAgent_1): Pack time: 0.000102
[2020-07-08 20:15:05.173377] (MonitorAgent_1): Tproc: 5e-06
[2020-07-08 20:15:05.175921] (MultiOutputMathAgent_1): Pack time: 0.00023
[2020-07-08 20:15:05.173245] (MultiGeneratorAgent_1): Sending: [0.47942554]
[2020-07-08 20:15:05.174524] (MonitorAgent_1): Received: {'from':
↪'MultiGeneratorAgent_1', 'data': array([0.47942554]), 'senderType':
↪'MultiGeneratorAgent', 'channel': 'cosine'}
[2020-07-08 20:15:05.176514] (MultiOutputMathAgent_1): Sending: {'sine_plus':␣
↪array([0.5])}
[2020-07-08 20:15:05.176643] (MultiOutputMathAgent_1): Tproc: 0.000988
[2020-07-08 20:15:05.174650] (MonitorAgent_1): Tproc: 5e-06
[2020-07-08 20:15:05.177175] (MultiOutputMathAgent_1): Received: {'from':
↪'MultiGeneratorAgent_1', 'data': array([0.47942554]), 'senderType':
↪'MultiGeneratorAgent', 'channel': 'cosine'}
[2020-07-08 20:15:05.178346] (MonitorAgent_1): Received: {'from':
↪'MultiOutputMathAgent_1', 'data': {'sine_plus': array([0.5])}, 'senderType':
↪'MultiOutputMathAgent', 'channel': 'default'}
[2020-07-08 20:15:05.177369] (MultiOutputMathAgent_1): Pack time: 9.9e-05
[2020-07-08 20:15:05.178461] (MonitorAgent_1): Tproc: 1.9e-05
[2020-07-08 20:15:05.177650] (MultiOutputMathAgent_1): Sending: {'cosine_minus':␣
↪array([-0.02057446])}
[2020-07-08 20:15:05.178849] (MonitorAgent_1): Received: {'from':
↪'MultiOutputMathAgent_1', 'data': {'cosine_minus': array([-0.02057446])},
↪'senderType': 'MultiOutputMathAgent', 'channel': 'default'}
[2020-07-08 20:15:05.177710] (MultiOutputMathAgent_1): Tproc: 0.00046
[2020-07-08 20:15:05.179303] (MonitorAgent_1): Memory: {'MultiOutputMathAgent_1': {
↪'sine_plus': array([0.5]), 'cosine_minus': array([-0.02057446])}}
[2020-07-08 20:15:05.179361] (MonitorAgent_1): Tproc: 0.000437
```

```
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
127.0.0.1 - - [08/Jul/2020 22:15:05] "POST /_dash-update-component HTTP/1.1" 204 -
```

CHAPTER 5

Tutorial 4 - A metrologically enabled pipeline.

In this tutorial we introduce the new metrologically enabled agents. We initialize an agent, which generates an infinite sine signal. The signal is generated from an external class `Signal` and delivers on each call one timestamp and one value each with associated uncertainties.

The *MetrologicalSineGeneratorAgent* is based on the new class *agentMET4FOF.metrological_agents.MetrologicalAgent*. We only adapt the methods `init_parameters()` and `agent_loop()`. This we need to hand over an instance of the signal generating class and to generate the actual samples. The rest of the buffering and plotting logic is encapsulated inside of the new base classes.

[1]:
```
# %load tutorial_4_metrological_agents.py
import time
from typing import Dict

import numpy as np
from time_series_metadata.scheme import MetaData

from agentMET4FOF.agents import AgentMET4FOF, AgentNetwork
from agentMET4FOF.metrological_agents import MetrologicalAgent,␣
↪MetrologicalMonitorAgent


class Signal:
    """
    Simple class to request time-series datapoints of a signal
    """

    def __init__(self):
        self._description = MetaData(
            device_id="my_virtual_sensor",
            time_name="time",
            time_unit="s",
            quantity_names="pressure",
            quantity_units="Pa",
        )
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def _time():
        return time.time()

    @staticmethod
    def _time_unc():
        return time.get_clock_info("time").resolution

    @staticmethod
    def _value(timestamp):
        return 1013.25 + 10 * np.sin(timestamp)

    @staticmethod
    def _value_unc():
        return 0.5

    @property
    def current_datapoint(self):
        t = self._time()
        ut = self._time_unc()
        v = self._value(t)
        uv = self._value_unc()

        return np.array((t, ut, v, uv))

    @property
    def metadata(self) -> MetaData:
        return self._description


class MetrologicalSineGeneratorAgent(MetrologicalAgent):
    """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.
    """

    # The datatype of the stream will be SineGenerator.
    _sine_stream: Signal

    def init_parameters(self, signal: Signal = Signal(), **kwargs):
        """Initialize the input data

        Initialize the input data stream as an instance of the
        :py:mod:`SineGenerator` class

        Parameters
        ----------
        signal : Signal
            the underlying signal for the generator
        """
        self._sine_stream = signal
        super().init_parameters()

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

(continues on next page)

(continued from previous page)

```
        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":
            self.set_output_data(channel="default", data=[self._sine_stream.current_
↪datapoint])
            super().agent_loop()

    @property
    def metadata(self) -> Dict:
        return self._sine_stream.metadata.metadata


def main():

    # start agent network server
    agent_network = AgentNetwork(dashboard_modules=True)

    # create and init agent
    signal = Signal()
    t_name, t_unit = signal.metadata.time.values()
    v_name, v_unit = signal.metadata.get_quantity().values()
    source_name = signal.metadata.metadata["device_id"]
    source_agent = agent_network.add_agent(name=source_name,
                                           agentType=MetrologicalSineGeneratorAgent)
    source_agent.init_parameters(signal)
    source_agent.set_output_data(channel="default", metadata=signal.metadata)
    monitor_agent = agent_network.add_agent(
        "MonitorAgent", agentType=MetrologicalMonitorAgent
    )

    # bind it to the monitor and activate it
    source_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__":
    main()
```

```
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 [2020-07-08 19:57:12.604560] (AgentController): INITIALIZED
Dash is running on http://127.0.0.1:8050/

 Warning: This is a development server. Do not use app.run_server
 in production, use a production WSGI server like gunicorn instead.

INFO [2020-07-08 19:57:12.886800] (my_virtual_sensor_1): INITIALIZED
```

(continues on next page)

(continued from previous page)

```
 * 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 [2020-07-08 19:57:12.974322] (MonitorAgent_1): INITIALIZED
[2020-07-08 19:57:13.032609] (my_virtual_sensor_1): Connected output module: MonitorAgent_1
SET STATE:   Running
[2020-07-08 19:57:13.918868] (my_virtual_sensor_1): Pack time: 0.001148
[2020-07-08 19:57:13.922292] (my_virtual_sensor_1): Sending: [array([[1.59423823e+09, 1.00000000e-09, 1.01943921e+03, 5.00000000e-01]]), <time_series_metadata.scheme.MetaData object at 0x7f3c99d062e0>]
[2020-07-08 19:57:13.926203] (MonitorAgent_1): Received: {'from': 'my_virtual_sensor_1', 'data': array([[1.59423823e+09, 1.00000000e-09, 1.01943921e+03, 5.00000000e-01]]), 'metadata': <time_series_metadata.scheme.MetaData object at 0x7f3c99d111f0>, 'senderType': 'MetrologicalSineGeneratorAgent', 'channel': 'default'}
[2020-07-08 19:57:13.927291] (MonitorAgent_1): Tproc: 0.000397
[2020-07-08 19:57:14.916511] (my_virtual_sensor_1): Pack time: 0.00048
[2020-07-08 19:57:14.917917] (my_virtual_sensor_1): Sending: [array([[1.59423823e+09, 1.00000000e-09, 1.00998696e+03, 5.00000000e-01]]), <time_series_metadata.scheme.MetaData object at 0x7f3c99d062e0>]
[2020-07-08 19:57:14.919316] (MonitorAgent_1): Received: {'from': 'my_virtual_sensor_1', 'data': array([[1.59423823e+09, 1.00000000e-09, 1.00998696e+03, 5.00000000e-01]]), 'metadata': <time_series_metadata.scheme.MetaData object at 0x7f3c99d111c0>, 'senderType': 'MetrologicalSineGeneratorAgent', 'channel': 'default'}
[2020-07-08 19:57:14.920310] (MonitorAgent_1): Tproc: 0.000119
```

```
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
127.0.0.1 - - [08/Jul/2020 21:57:15] "POST /_dash-update-component HTTP/1.1" 204 -
127.0.0.1 - - [08/Jul/2020 21:57:15] "POST /_dash-update-component HTTP/1.1" 200 -
127.0.0.1 - - [08/Jul/2020 21:57:15] "POST /_dash-update-component HTTP/1.1" 200 -
```

[ ]:

CHAPTER 6

agentMET4FOF agents

class agentMET4FOF.agents.**AgentBuffer**(*buffer_size=1000*)

Buffer class which is instantiated in every agent to store data incrementally. This buffer is necessary to handle multiple inputs coming from agents. The buffer can be a dict of iterables, or a dict of dict of iterables for nested named data. The keys are the names of agents.

buffer_filled(*agent_from=None*)

Checks whether buffer is filled.

Parameters **agent_from** (*str*) – Name of input agent in the buffer dict to be looked up for. If *agent_from* is not provided, we check for all iterables in the buffer. For nested dict, this returns true for any iterable which is beyond the *buffer_size*.

clear(*agent_from=None*)

Clears the data in the buffer. if *agent_from* is not given, the entire buffer is removed.

agent_from [str] Name of agent

popleft(*n=1*)

Pops the first n entries in the buffer.

store(*agent_from*, *data=None*)

Stores data into *self.buffer* 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

- **agent_from** (*dict | str*) – if type is dict, we expect it to be the agentMET4FOF dict message to be compliant with older code otherwise, we expect it to be name of agent sender and *data* will need to be passed as parameter
- **data** – optional if agent_from is a dict. Otherwise this parameter is compulsory. Any supported data which can be stored in dict as buffering.
- **concat_axis** (*int*) – axis to concatenate on with the buffering for numpy arrays.

update(*agent_from: str*, *data*)
Overrides data in the buffer dict keyed by *agent_from* with value *data*

If *data* is a single value, this converts it into a list first before storing in the buffer dict.

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.

Behavioral 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

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

buffer_clear(*agent_name=None*)
Empties buffer which is a dict indexed by the *agent_name*.

Parameters **agent_name** (*str*) – Key of the memory dict, which can be the name of input agent, or self.name. If one is not supplied, we assume to clear the entire memory.

buffer_filled(*agent_name=None*)
Checks whether the internal buffer has been filled to the maximum allowed specified by self.buffer_size

Parameters **agent_name** (*str*) – Index of the buffer which is the name of input agent.

Returns **status of buffer filled**

Return type boolean

buffer_store(*agent_from: str*, *data=None*)
Updates data stored in *self.buffer* 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

- **agent_from** (*str*) – Name of agent sender
- **data** – Any supported data which can be stored in dict as buffer. See AgentBuffer for more information.

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: Optional[int] = None*)
Initiates the agent loop, which iterates every *loop_wait* seconds

Stops every timers and initiate a new loop.

Parameters **loop_wait** (*int, optional*) – 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(*default_buffer_size=1000*)
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

buffer_size
The total number of elements to be stored in the agent *buffer* 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: Union[matplotlib.figure.Figure, Dict[str, matplotlib.figure.Figure]], mode: str = 'image'*)

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.

Tradeoffs between "image" and "plotly" modes are that "image" are more stable and "plotly" are interactive. Note not all (complicated) matplotlib figures can be converted into a plotly figure.

Parameters

- **fig** (*matplotlib.figure.Figure or dict of matplotlib.figure.Figure*) – Alternatively, multiple figures can be nested in a dict (with any preferred keys) e.g {"Temperature":matplotlib.Figure, "Acceleration":matplotlib.Figure}
- **mode** (*str*) – "image" - converts into image via encoding at base64 string. "plotly" - converts into plotly figure using *mpl_to_plotly* Default: "image"

Returns graph

Return type str or plotly figure or dict of one of those converted figure(s)

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

class agentMET4FOF.agents.**AgentNetwork**(*ip_addr='127.0.0.1'*, *port=3333*, *connect=False*, *log_filename='log_file.csv'*, *dashboard_modules=True*, *dashboard_extensions=[]*, *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*, *buffer_size=1000*, *ip_addr=None*, *loop_wait=None*, ***kwargs*)

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

add_coalition(*name='Coalition_1'*, *agents=[]*)

Instantiates a coalition of agents.

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 ()
Shuts down 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.

CHAPTER 7

agentMET4FOF metrologically enabled agents

class agentMET4FOF.metrological_agents.**MetrologicalAgent**(*name=''*, *host=None*, *serializer=None*, *transport=None*, *attributes=None*)

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(*input_data_maxlen=25*, *output_data_maxlen=25*)
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

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}.

class agentMET4FOF.metrological_agents.**MetrologicalMonitorAgent**(*name=''*, *host=None*, *serializer=None*, *transport=None*, *attributes=None*)

init_parameters(**args*, ***kwargs*)

User provided function to initialize parameters of choice.

CHAPTER 8

agentMET4FOF streams

class agentMET4FOF.streams.**CosineGenerator**(*sfreq=500*, *F=5*)

Built-in class of cosine wave generator. *sfreq* is sampling frequency which determines the time step when next_sample is called *F* is frequency of wave function *cosine_wave_function* is a custom defined function which has a required keyword *time* as argument and any number of optional additional arguments (e.g *F*). to be supplied to the *set_generator_function*

class agentMET4FOF.streams.**DataStreamMET4FOF**

Abstract class for creating datastream.

Data can be fetched sequentially using *next_sample()* or all at once *all_samples()* This increments the internal sample index *sample_idx*.

For sensors data, we assume: The format shape for 2D data stream (timesteps, n_sensors) The format shape for 3D data stream (num_cycles, timesteps , n_sensors)

To create a new DataStreamMET4FOF class, inherit this class and call *set_metadata* in the constructor. Choose one of two types of datastreams to be created: from dataset file (*set_data_source*), or a waveform generator function (*set_generator_function*). Alternatively, override the *next_sample* function if neither option suits the application. For generator functions, *sfreq* is a required variable to be set on *init* which sets the sampling frequency and the time-step which occurs when next_sample() is called.

For an example implementation of using generator function, see the built-in *SineGenerator* class. See tutorials for more implementations.

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 latest *batch_size* samples from the iterables: quantities, time and target. This advances the internal pointer *current_idx* by *batch_size*.

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

Returns **samples**

Return type dict of the form *{'time':current_sample_time,'quantities': current_sample_quantities, 'target': current_sample_target}*

set_data_source(*quantities=None*, *target=None*, *time=None*)

This sets the data source by providing three iterables: *quantities* , *time* and *target* which are assumed to be aligned.

For sensors data, we assume: The format shape for 2D data stream (timesteps, n_sensors) The format shape for 3D data stream (num_cycles, timesteps , n_sensors)

Parameters

- **quantities** (*iterable*) – Measured quantities such as sensors readings.
- **target** (*iterable*) – (Optional) Target label in the context of machine learning. This can be Remaining Useful Life in predictive maintenance application. Note this can be an unobservable variable in real-time and applies only for validation during offline analysis.
- **time** (*iterable*) – (Optional) dtype can be either float or datetime64 to indicate the time when the *quantities* were measured.

set_data_source_type(*dt_type='function'*)

To explicitly account for the type of data source: either from dataset, or a generator function

Parameters **dt_type** (*str*) – Either "function" or "dataset"

set_generator_function(*generator_function=None*, *sfreq=None*, ***kwargs*)

Sets the data source to a generator function. By default, this function resorts to a sine wave generator function. Initialisation of the generator's parameters should be done here such as setting the sampling frequency and wave frequency. For setting it with a dataset instead, see *set_data_source*.

Parameters

- **generator_function** (*method*) – A generator function which takes in at least one argument *time* which will be used in *next_sample*. Parameters of the function can be fixed by providing additional arguments such as the wave frequency.
- **sfreq** (*int*) – Sampling frequency.
- ****kwargs** – Any addtional keyword arguments to be supplied to the generator function. The **kwargs will be saved as *generator_parameters*. The generator function call for every sample will be supplied with the **generator_parameters.

class agentMET4FOF.streams.**SineGenerator**(*sfreq=500*, *F=5*)

Built-in class of sine wave generator. *sfreq* is sampling frequency which determines the time step when next_sample is called *F* is frequency of wave function *sine_wave_function* is a custom defined function which has a required keyword *time* as argument and any number of optional additional arguments (e.g *F*). to be supplied to the *set_generator_function*

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 9

agentMET4FOF dashboard

class agentMET4FOF.dashboard.Dashboard.**AgentDashboard**(*dashboard_modules=[]*, *dashboard_layouts=[]*, *dashboard_update_interval=3*, *max_monitors=10*, *ip_addr='127.0.0.1'*, *port=8050*, *agentNetwork='127.0.0.1'*, *agent_ip_addr=3333*, *agent_port=None*)

Class for the web dashboard which runs with the AgentNetwork object, which by default are on the same IP. Optional to run the dashboard on a separate IP by providing the right parameters. See example for an implementation of a separate run of dashboard to connect to an existing agent network. If there is no existing agent network, error will show up. An internal _Dashboard_Control object is instantiated inside this object, which manages access to the AgentNetwork.

init_app_layout(*update_interval_seconds=3*, *max_monitors=10*, *dashboard_layouts=[]*)

Initialises the overall dash app "layout" which has two sub-pages (Agent network and ML experiment)

Parameters

- **update_interval_seconds** (*float or int*) – Auto refresh rate which the app queries the states of Agent Network to update the graphs and display
- **max_monitors** (*int*) – Due to complexity in managing and instantiating dynamic figures, a maximum number of monitors is specified first and only the each Monitor Agent will occupy one of these figures. It is not ideal, but will undergo changes for the better.

Returns app

Return type Dash app object

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

CHAPTER 11

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

G

H

I

L

M

N

O

P

R

S

U