FLAViz: Flexible Large-scale Agent Visualization library[⋆]

Krishna Raj Devkota[a] Sander van der Hoog[a]

February 7, 2018

Abstract

The software package FLAViz described in this technical report has been developed specifically to analyse and visualize data generated by agent-based simulation models. Such models typically generate the data spanning multiple dimensions, e.g. parameter sets, Monte Carlo replication runs, different agent types, multiple agent instances per type, many variables per agent, and time periods (iterations).

To deal with such a large heterogeneity in the data dimensions, the data has to be stored as organized data sets, to allow for proper data aggregation, data filtering, selection, slicing etc. The software package FLAViz can be used to filter, transform, and visualize time series data that has been stored using multiple hierarchical levels in the HDF5 file format. Various kinds of plots can be specified, e.g., time series, box plots, scatter plots, histograms, and delay plots.

[⋆]We gratefully acknowledge funding provided by the Conquaire Project under DFG contract no. ADD CONTRACT NUMBER.

[a]Email: {kdevkota,svdhoog}@uni-bielefeld.de, Chair for Economic Theory and Computational Economics (ETACE), Dept. of Business Administration and Economics, Bielefeld University, Germany.

1 Overview

This technical report described the release of FLAViz 0.1.0 (beta version), a new visualization library developed at the ETACE group at Bielefeld University (Economic Theory and Computational Economics).

Headlines:

- FLAViz is part of the FLAME ecology, a set of tools to develop, simulate and analyse large-scale agent-based models.[1]
- FLAViz targets the analysis and visualization of time series data produced from any ABM, but was specifically designed with output from FLAME models in mind.

FLAViz provides the following features:

- Easy-to-configure `yaml` files
- Fully written in Python (support for both Python 2.7 and 3.6)
- Based on Python pandas (0.21.0) and matplotlib as standards for data analysis and visualization
- Data conversion from XML files to HDF5 files
- Data transformations of agent variables
- Data selection based on data structure (e.g., select all data at iteration $t = x$, or select all data for agent $ID = i$)
- Data filtering based on agent or variable conditions (e.g., filter the selected data on the condition that agent variable X==value)
- Data visualization using various styles: time series, box plots, scatter plots, table output

Dependencies:

```
Python (3.6)
Python pandas (0.21.0)
matplotlib
hdf5
PyTables
```

[1]FLAME stands for Flexible Large-scale Agent Modelling Environment. The FLAME website is at: www.flame.ac.uk, FLAME software can be downloaded from the GitHub repository at: http://github/FLAME-HPC.

1.1 File structure

This package consists of several Python scripts, located in the folder `/src`, dealing with different steps of the data visualisation and data transformation process:

- `main.py` : Contains code to read in the input data and primary configuration files (`config.yaml,transform.yaml`), filter the data based on filter conditions, and link the different Summary, Plot, and Transformation modules.
- `summarystats.py` : Takes in a Pandas dataframe, and computes the summary, and outputs the result as a Pandas dataframe.
- `plots.py`: Takes in a Pandas dataframe, and returns the necessary plots as specified in `config.yaml` and `plot.yaml`.
- `transform.py` : Takes in a Pandas dataframe, and returns/writes to a file the data transformations specified in `transform.yaml`.

The configuration files are contained in a folder named `/config`, containing three configuration files:

- `config.yaml` : defines `i/o path`, `plot-types`, `agents` and specifies variables, filter options, summary.
- `plot.yaml` : defines plot properties i.e. `name,legends,linestyle,fill`, etc.
- `transform.yaml`: defines variables to transform, type of data transformations, and `i/o` info to store data in a file after transformations.

Note: Further details on how to use the configuration (`yaml`) files can be found in Section 1.4 of this documentation.

1.2 Running the main module

To run the main executable, simply use:

```
$ python main.py configuration-folder-path
```

where, `configuration-folder-path` is the path to the folder containing the `yaml` configuration files.

Note: The functionality of this software has only been tested under Unix/Unix-like systems. It has not been tested for Windows and therefore there is no guarantee of proper execution for such systems.

1.3 Input and Output data storage format

The input data is stored in a *HDF5* container file (.h5, .hdf5) in a hierarchical format. Currently, the HDF5 file is structured as follows:

- Each agent-type is contained in a separate HDF5 file, with the same name as the agent type name.

- Each HDF5 file has a single hierarchy, with the *agent-type* as the root, and the *set* and *runs* as the branches (data set and data groups, respectively).
- The set and runs branches each contain a *Pandas 3D Data Panel*, which itself contains three axes: `major,minor`, and `items` axis.
- The *Pandas 3D Data Panel* is written to the HDF5 file with the help of the *PyTables* module in Python.[2]

A HDF5 file as described above can be created from the SQLite db files by using the data processing scripts, included in the data processing directory

Note: To avoid any unwanted errors, it is imperative to name the SQLite database files using the following convention: `set_s_run_r_iters.db`, where s is the set number and r is the run number.

1.4 Configuration files

There are three configuration files, which allow the user to input the necessary parameters for the program. The configuration files are specified in the `yaml` format that uses a hierarchical format which is not just for clarity but also for specifying functionality. Hence, it is important to abide by the indentation of the yaml files in order for them to be interpreted correctly.

Note: Any error in a yaml file might be caught by the exception handler, but indentation errors go unnoticed sometimes, which may result in undesired output. Hence, extra care is advised when formulating a configuration file.

1.4.1 File: config.yaml

Data input/output section

`i/o`: Specify the input and output path.

- The input path is specified as a full path in the sub-hierarchy `input_path`:

```
i/o:
  input_path:
      Bank:       '/home/etace/Bank.h5'
      Eurostat:   '/home/etace/Eurostat.h5'
```

Note: The key name to the input path should correspond to the Agent-type (e.g., Bank, Eurostat, Firm etc.)

- The output path is specified as a full path in the sub-hierarchy `output_path`:

```
  output_path:  '/home/etace/Data/timeseries'
```

[2]For performance we have used PyTables using the 'fixed' write-only mode, which does not allow to append data to the HDF5 file lateron. An alternative would be to use the 'append' mode, but our tests have shown that this incurs considerable performance degradations.

Plot section

```
plot1:
    timeseries:
        agent: Bank
        analysis: multiple_set
        variables:
          var1: [total_credit]
          var2: [equity]
```

- **Plot-key** (e.g., `plot1`): Specify a key for the plot (mainly to keep track of the plot-number for other configuration files). Can be any string.
- **Plot-type** (e.g., `timeseries`): Nested under **Plot-key** (here `plot1`), "Plot-type" specifies the type of plot desired.
  - Possible values: `timeseries,boxplot,histogram,scatterplot`.

Note [Exception]: For the case of transform, simply specify `transform` in the Plot-type, and it will perform the transform and store the new data items to a specified output file (no plots will be produced).

```
plot1:
    transform:
        agent: Bank
        analysis: multiple_set
        variables:
          var1: [total_credit]
          var2: [equity]
```

- `agent`: Name of the agent-type, nested under **Plot-type**.
- `analysis`: Type of analysis. Possible types: `agent,multiple_run,multiple_batch,multiple_set`.
- `variables`: Variables from the particular agent-type that is to be processed or visualized. The sub-hierarchy `var1,var2` etc. allows the input of multiple variables for any agent type. The variable names can be inside a set of square brackets `[]` or simply inside a set of single-quotation marks `''`.

Conditional filtering

There is an option to filter variables based on filter conditions on the values, i.e. to retrieve only those values that satisfy a certain restrictions or that fall within a certain range. For conditional filtering, specify the variables as above, but with the filter conditions in addition.

- Possible operator types are: `<,>,<=,>=,==`.
- Simple or multiple filter conditions on a variable are possible.
- Filtering on multiple variables can be specified.

Listing 1: Filtering examples using single and multiple filter conditions.

```
var1: [variable name, 'operator[value]']
e.g.
var1: [total_credit , '>[700]']
# Select only values of total credit greater than 700.

var2: [variable name, 'operator1[value]','operator2[value]']
e.g.
var2: [equity , '>[700]', '<[1500]']
# Select only values of total credit between 700 and 1500.
```

Listing 2: Filtering example using multiple variables.

```
plot1:
    timeseries:
        agent: Bank
        analysis: multiple_set
        variables:
              var1: [total_credit , '>[700]']
              var2: [equity , '>[700]', '<[800]']
```

Data selection

For the **sets**, **runs**, **major** and **minor** axes, data selections can be specified as ranges or lists:

- `set` : Specify the sets to process. Input can be an explicit list, or (esp. for long lists) a custom way is to specify a range of values.
- `run` : Specify the runs to process. Syntax is similar to `sets` above.
- `major` : Specify the values from the major axis (iterations). Syntax similar to `sets` above.
- `minor` : Specify the values from the minor axis (agent instances). Syntax similar to `sets` above.

Example of selecting sets using an explicit list of values:

```
set: [val(1),val(2),...,val(N)]
e.g.
set: [1,2]
```

Example of selecting sets using a range of values (list with values from 1 to 10 with a step size of 2):

```
set: [range,[val(1),val(N),stepsize]]
e.g.
set: [range, [1,10,2]]
```

Note: The `set,run,major`, and `minor` values are nested under "Plot-type"

Full example:

```
plot1:
    timeseries:
        agent: Bank
        analysis: multiple_set
        variables:
              var1: [total_credit]
              var2: [equity]
        set: [1]
        run: [1,2]
        major: [range,[6020,26000,20]]
        minor: [1,5,7] # only consider agents 1,5,7
```

- `summary`: Specify the type of statistical summary. This is also nested under Plot-type.
  - Possible values: `mean`,`median`,`custom_quantile`,`upper_quartile`, `lower_quartile`,`maximum`,`minimum`.

Example:

```
plot1:
    timeseries:
        summary: mean
```

Complete config.yaml file

Hence, a typical complete `config.yaml` configuration file may look like this:

```
i/o:
    input_path:
        Bank: '/home/etace/Bank.h5' # key is the agent type
        Eurostat: '/home/etace/Eurostat.h5'
    output_path: '/home/etace/timeseries'

plot1:
    timeseries:
        agent: Bank
        analysis: multiple_set
        variables:
            var1: [total_credit]
            var2: [equity]
        set: [1]
        run: [1,2]
        major: [range,[6020,26000,20]]
        minor: [1,5,7]
        summary: mean

plot2:
    boxplot:
        agent: Eurostat
        analysis: multiple_run
        variables:
            var1: [total_credit]
        set: [1]
        run: [1]
        major: [range,[6020,6900,20]]
        minor: [1,8]
        summary: custom_quantile
```

1.4.2 File: plot.yaml

Whenever a plot is specified in the main configuration file `config.yaml`, the file `plot.yaml` is read for further specifications of the plot (line styles, axes labels, legend placement, etc.). The `plot.yaml` file contains all the necessary configurations for each plot to be generated, linked by the **Plot-key** as specified in the `config.yaml` file. Below we explain the parameters from the `plot.yaml` file.[3]

- **Plot-key** (e.g., `plot1`): This string should be the same as the **Plot-key** in the `config.yaml` file, to make sure the correct parameters are mapped to the respective plotting modules.
- `number_plots`: Specifies how many plots will be output per variable for a particular agent type.
  - Possible values: `one,many`.
  - `one`: all data series will be displayed in a single plot. For time series plots this means: many lines in one plot.
  - `many`: each data series is displayed in a separate plot. There will be as many plots as data series selected.
- `plot_name`: Specify a filename for the plot. (Note: In case of multiple plots, a numerical suffix (in increasing order) is added after the specified file name.)
- `plot_legend`: Specify whether a legend for the plot should be displayed.
- `legend_loc`: Specify the location of the legend, either inside or outside of the graph box.
  - Possible values: `in,out`.
- `legend_label`: Specify a name for the lines in the plot. Can be any string value.
- `x-axis label`: Specify a label for the x-axis. Can be any combination of string values.
- `y-axis label`: Specify a label for the y-axis. Can be any combination of string values.
- `linestyle`: Specify line characteristics.
  - Possible values: `solid,dashed,dashdot,dotted` etc.

Complete plot.yaml file

[3]The plotting options follow the specifications in the `matplotlib` library, which is the default library to plot data with Python pandas.

```
plot1:
    number_plots: one
    plot_name: timeseries_equity.png
    l_lim: no
    u_lim: no
    tmin: no
    tmax: no
    plot_legend: yes
    legend_loc: out
    legend_label: equity
    x-axis label: months
    y-axis label: equity value
    linestyle: solid


plot2:
    number_plots: many
    plot_name: boxplot_monthly_output.png
    l_lim: no
    u_lim: no
    tmin: no
    tmax: no
    plot_legend: no
    legend_loc: in
    legend_label: monthly_output
    x-axis label: months
    y-axis label: monthly_output
    linestyle: dashed
```

1.4.3 File: transform.yaml

The `transform.yaml` file contains all the specifications for any data transformations. Whenever a transformation is specified in the `config.yaml` file, the `transform.yaml` file is read to perform the required transformations, and store the resulting data in a new output file.

The parameters in the `transform.yaml` file are as follows:

- **Plot-key** (e.g., `plot1`): This string should be the same as the **Plot-key** used in the `config.yaml` file for identifying the data transformation block, to make sure the correct parameters are mapped to the respective data transformation modules. (Note: Although it is called Plot-key, the transform case is an exception and no plots are produced in this case.)
- `variables`: Variables from the particular agent-type that are to be transformed. The sub-hierarchy `var1,var2` etc. allows the input of multiple variables for any agent type.
- `transform_function`: The transformation function to apply for the given variables.

  Possible transform functions are:

  - Quarterly growth rate (quarter on quarter): `q_o_q`
  - Quarterly growth rate (for one cycle): `q_o_q_ONE_CYCLE`
  - Monthly growth rate (month on month): `m_o_m`
  - Monthly growth rate (for one cycle): `m_o_m_ONE_CYCLE`

  - Annual growth (for two given points on time): `annual_P_I_T`
  - Other custom functions can be user-specified.

  Note: Other elementary functions such as **sum**, **difference**, **product**, and **division** can also be performed, which has been left for the user (will be added as custom functions).

- `aggregate`: If the transformation is to be performed after calculating the summary stats, then a necessary aggregation method can be specified.
  - Possible values: `mean,median,maximum,minimum,custom_quantile,` `upper_quartile,lower_quartile`.
- `write_file`: Specify whether to write the transformation to a file.
  - Possible values: `yes,no`.
- `output_path`: If the `write_file` option above is set to `yes`, then an output path for the file needs to be specified. This can be any valid file path, as a string, including the filename.
- `hdf_groupname`: Specify the rootname for the HDF5 group name (internal hierarchy) for the transformed variable. Can be any valid string.

Complete transform.yaml file
A particular complete `transform.yaml` file may look as follows:

```
plot2:
    variables:
        var1: total_credit
        var2: equity
    transform_function: m_o_m_ONE_CYCLE
    aggregate: mean

    write_file: yes
    output_path: '/home/etaceguest/transformed.h5'
    hdf_groupname: 'total_credit_equity_ratio'
```

2 Walkthrough and Tutorial examples

In this section we give examples of input configuration files and their respective output plots. Once the necessary parameters are set, by following the instructions specified in Section 1.4, the module can be run to get the desired results. To demonstrate some of the functionalities, the parameters of the configuration files are shown below, along with the plots they yield.

Using the dataset `<INSERTDATASETURL>`, and the following parameter settings, the following plots can be produced.

2.0.4 Example 1

For agent Firm, one set, ten runs, eighty instances, plotted in a single plot.

```
config.yaml::

    plot1:
        timeseries:
            agent: Firm
            analysis: multiple_run
            variables:
                var1: [price]
            set: [13]
            run: [range,[1,10]]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: mean

plot.yaml::

    plot1:
        number_plots: one
        plot_name: one_set_multiple_runs_timeseries.png
        plot_legend: yes
        legend_location: best
        xaxis_label: Time
        yaxis_label: price
        linestyle: solid
        marker: None
```

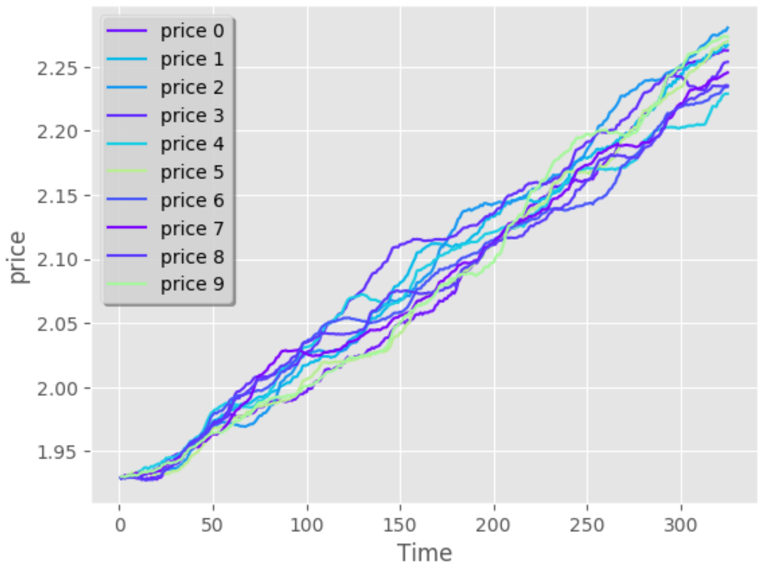

Figure 1: Example 1: one set, multiple runs, timeseries, price.

2.0.5 Example 2

For agent Firm, one set, one run, 20-80 quantiles of population distribution across 80 agent instances, plotted in a single plot.

```
config.yaml::

    plot2:
        timeseries:
            agent: Firm
            analysis: multiple_run
            variables:
                var1: [price]
            set: [10]
            run: [1]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: custom_quantile
            quantile_values:
               lower_quantile : 0.20
               upper_quantile : 0.80

plot.yaml::

    plot2:
        number_plots: one
        plot_name: one_set_multiple_runs_ts_quantile.png
        plot_legend: yes
        legend_location: best
        xaxis_label: Time
        yaxis_label: price
        linestyle: solid
        marker: None
        fill_between_quartiles: yes
        fillcolor: red
```

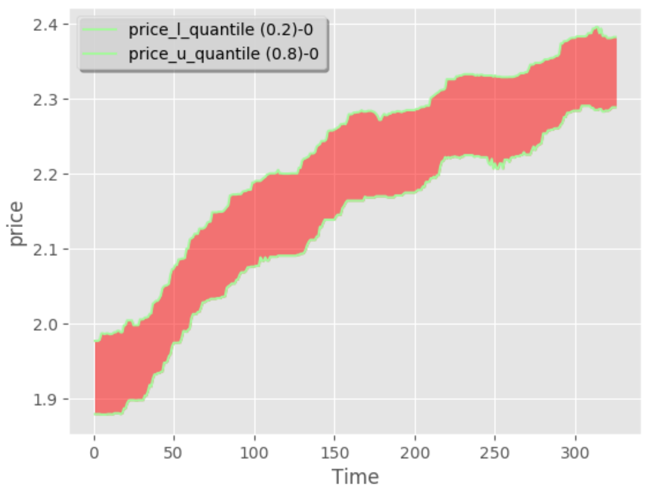

Figure 2: Example 2: Quantiles 20 and 80 of the population distribution across 80 agent instances.

2.0.6 Example 3

For agent Firm, one set, one run, eighty agent instances, boxplot.

```
config.yaml::

    plot3:
        boxplot:
            agent: Firm
            analysis: multiple_set
            variables:
                var1: [price]
            set: [13]
            run: [1]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]

plot.yaml::

    plot3:
        number_plots: one
        plot_name: one_set_one_run_bp_price.png
        plot_legend: yes
        legend_label: (Agent = Firm, var = Price)
        legend_location: best
        xaxis_label: Time
        yaxis_label: Distribution over price
        number_bars: 5
```

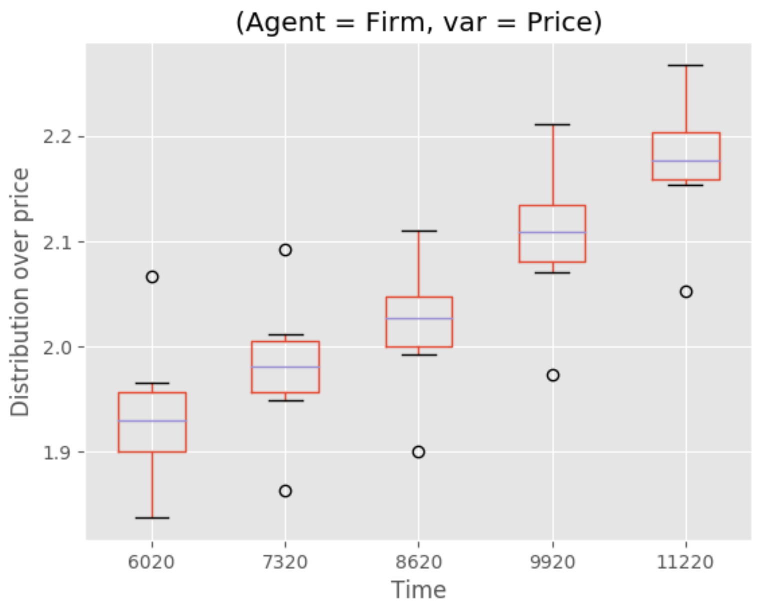

Figure 3: Example 3: boxplots of the population distribution across 80 agent instances, the number of boxplots can be specified (here: 5).

2.0.7 Example 4

For agent Firm, one set, twenty runs, 80 agent instances, scatter plot of two variables.

```
config.yaml::

    plot4:
        scatterplot:
            agent: Firm
            analysis: multiple_batch
            variables:
                var1: [price]
                var2: [output]
            delay: no
            set: [13]
            run: [range,[1,20]]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: mean

plot.yaml::

    plot4:
        number_plots: one
        plot_name: one_set_multiple_runs_sp_price_output.png
        plot_legend: yes
        legend_location: best
        legend_label: price vs. output
        linestyle: solid
        marker: +
```

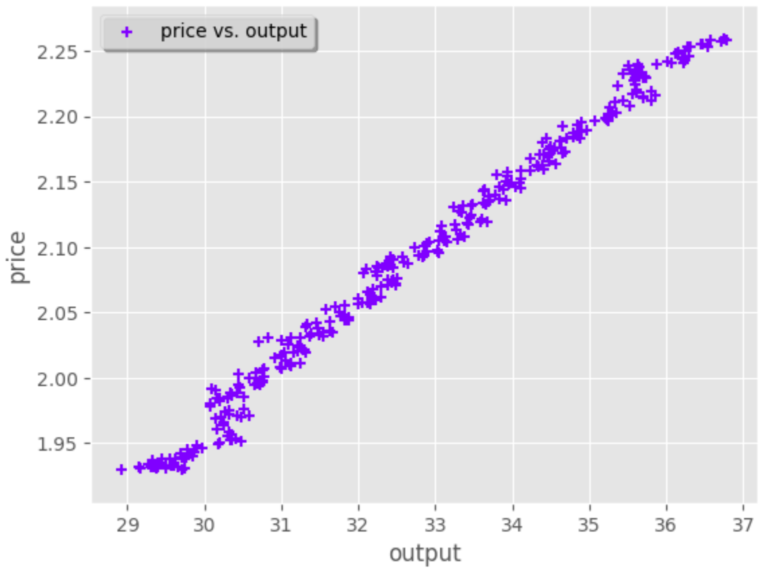

Figure 4: Example 4: Scatter plot of 2 variables, for a data set consisting of 20 runs, 80 agent instances. We first took the mean across the agents, then plot the ensemble data across all runs, all iterations (the plot shows 20 × 325 points).

2.0.8 Example 5

For agent Firm, one set, twenty runs each, eighty instances each, delay plot for one variable.

```
config.yaml::

    plot5:
        scatterplot:
            agent: Firm
            analysis: multiple_batch
            variables:
                var1: [price]
            delay: yes
            set: [13]
            run: [range,[1,20]]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: mean

plot.yaml::

    plot5:
        number_plots: one
        plot_name: one_set_multiple_runs_sp_price_delay.png
        plot_legend: yes
        legend_location: best
        legend_label: price vs. price_delay
        linestyle: solid
        marker: +
```

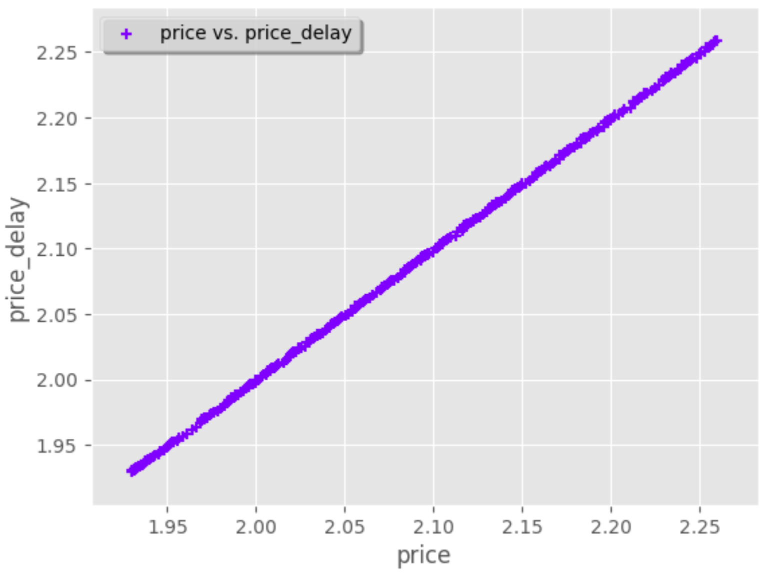

Figure 5: Example 5: Delay plot (x_t, x_{t+1}).

2.0.9 Example 6

For agent Firm, one set, one run, 80 instances, histogram of one variable.

```
config.yaml::

    plot6:
        histogram:
            agent: Firm
            analysis: multiple_run
            variables:
                var1: [price]
            set: [10]
            run: [1]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: mean

plot.yaml::

    plot6:
        number_plots: one
        plot_name: one_set_one_run_hg_price.png
        plot_title: (Agent = Firm, var = Price)
        number_bins: 50
        histtype: bar
        plot_legend: yes
        fill: yes
        stacked: False
        legend_location: best
        xaxis_label: xlabel
        yaxis_label: ylabel
```

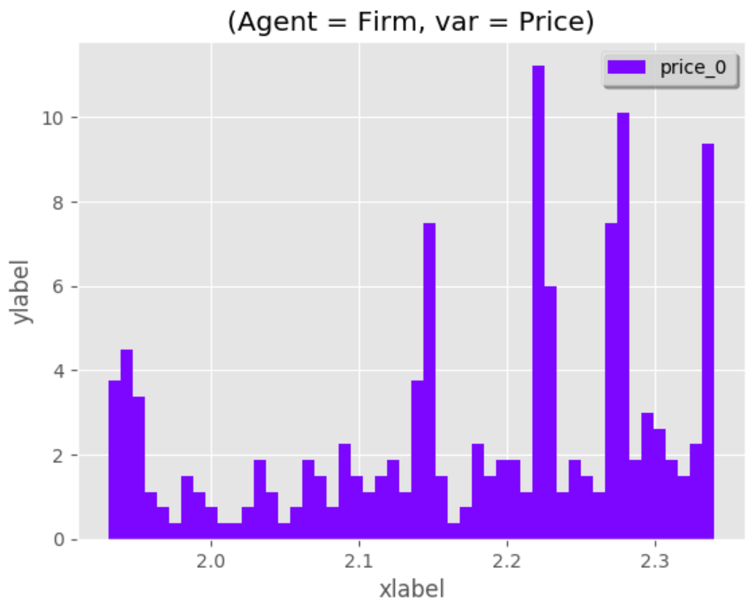

Figure 6: Example 6: Histogram of a single variable across the agent population.

2.0.10 Example 7

For agent Firm, one set, twenty runs, 80 instances, histogram of distribution over the sets.

```
config.yaml::

    plot7:
        histogram:
            agent: Firm
            analysis: multiple_set
            variables:
                var1: [price]
            set: [10]
            run: [range,[1,20]]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: mean

plot.yaml::

    plot7:
        number_plots: one
        plot_name: one_set_multiple_runs_hg_price.png
        plot_title: (Agent = Firm, var = Price)
        number_bins: 50
        histtype: step
        plot_legend: yes
        fill: no
        stacked: False
        legend_location: best
        xaxis_label: xlabel
        yaxis_label: ylabel
```

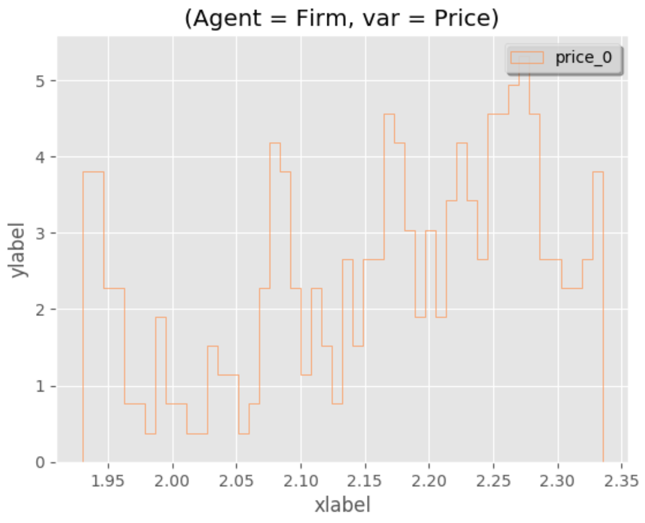

Figure 7: Example 7: Histogram of a single variable across 20 runs, and across the agent population. Shown is the ensemble distribution across all runs.

2.0.11 Example 8

For agent Firm, four sets, twenty runs each, 80 agent instances, the variable price is plotted in a single plot.

```
config.yaml::

    plot8:
        timeseries:
            agent: Firm
            analysis: multiple_batch
            variables:
                var1: [output]
            set: [10,13,16,17]
            run: [range,[1,20]]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: mean

plot.yaml::

    plot8:
        number_plots: one
        plot_name: timeseries_agentanalysis.png
        plot_legend: yes
        legend_location: best
        x-axis label: Time
        y-axis label: output
        linestyle: dashed
        marker: None
```

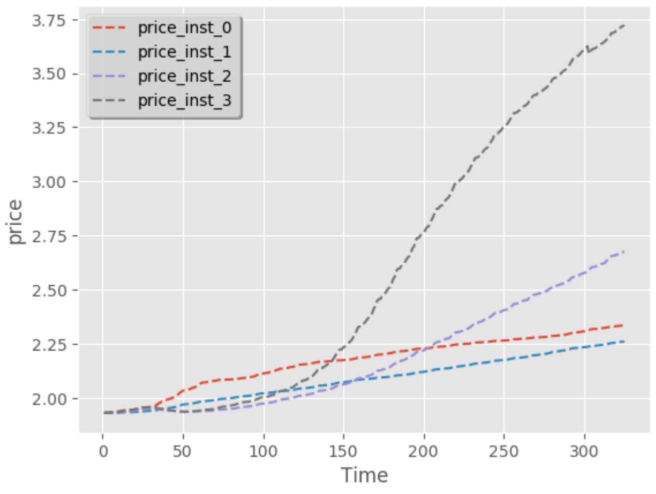

Figure 8: Example 8: Time series plot of multiple sets (4), multiple runs (20), and multiple agent instances (80). Each line represents one set, and displays the mean across runs, and across agents.

2.0.12 Example 9

For agent Firm, four sets, twenty runs each, 80 agent instances, 20-80 quantiles of the population distribution of the variable price are plotted in a single plot.

```
config.yaml::

    plot7:
        timeseries:
            agent: Firm
            analysis: multiple_batch
            variables:
                var1: [price]
            set: [10,13,16,17]
            run: [range,[1,20]]
            major: [range,[6020,12500,20]]
            minor: [range,[1,80]]
            summary: custom_quantile
            quantile_values:
                lower_quantile : 0.20
                upper_quantile : 0.80


plot.yaml::

    plot7:
        number_plots: one
        plot_name: ts_multibatch_analysis.png
        plot_legend: yes
        legend_location: best
        x-axis label: Time
        y-axis label: price
        linestyle: solid
        marker: None
        fill_between_quartiles: yes
```

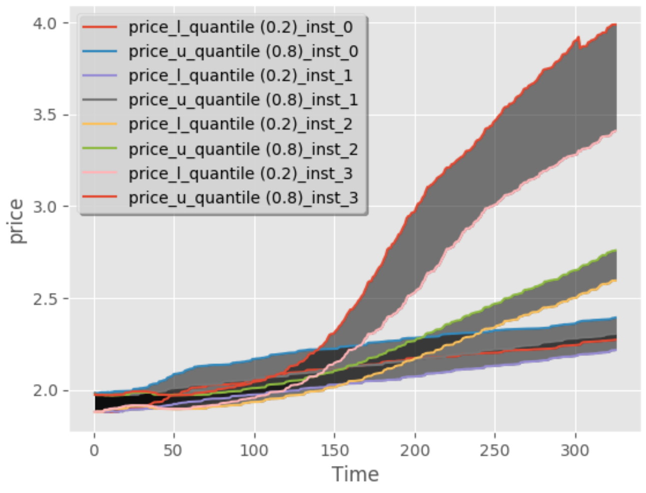

Figure 9: Example 9: Quantile ranges of the variable 'price' for multiple sets (4), multiple runs (20), and multiple agent instances (80). For each set the quantile range (20-80) is displayed.

3 FLAViz FAQ

Q: Why is FLAViz based on Python pandas and matplotlib?

A: Python pandas is open source, has an active user community, and has many developers. It is the current library of choice for time series data analysis. It provides many in-build statistical functionalities. There are two main functionalities of pandas that are especially important for us:

- hierarchical indexing: this allows a high dimensional data frame (the ndarray format)
- bygroup

Q: Why the conversion from XML files to HDF5 format?

A: The XML files that FLAME outputs is a fully tagged data format, so it is very verbose. For large scale simulations this is prohibitive, due to the sheer size of the data volumes this generates. To reduce this storage footprint, but still keep all data together in a structured format, the HDF5 standard was chosen for its hierarchical structure. The data for each agent type is stored in a single HDF5 file, which can be of any size (we have so far dealt with single files of up to 20 GB without any problems). Inside of the HDF5 file there is a POSIX-style folder hierarchy, with data groups and data sets. A particular requirement for HDF5 is that the 'data set' has a homogeneous data structure. For this data structure we have selected the 3D DataFrame from Python pandas (but we should note that the current development of Python pandas goes so quickly, that currently this is shifting to the ndarray, which will replace the 3D Data Panel in a more generic data format).

Q: How do you do the data transformations, data selection and data filtering?

We use the bygroup function of pandas to sort the hierarchical index, and to rotate the data frame.

Q: Can I use FLAViz with other agent-based simulation platforms than FLAME?

Yes, you can! The only requirement is that the final data output is either in XML or in HDF5 format (see the file format specifications).

Q: What file format specifications do you have?

See: file format specifications (ADD LINK).

Data formats

Simulations We adopted the following ontology to describe how we run simulations:

- "Sets": a set reflects a model parameter setting. Each set differs from another set only in the parameter setting of the model.
- "Runs": a run is a replication, for a fixed parameter setting. Each run differs from other runs by the random seed only. The other initial conditions are kept exactly the same across runs.

Thus, parameter variations are captured in "settings" or "sets". Each set reflects a different parametrization of the simulation model. If we have any stochasticity in the model, we need to explore the statistical properties using the random seed for the Random Number Generator. By default we use the RNG from the GNU/GSL library (a Mersenne Twister, mt19937, with a periodicity of $2^{19937} - 1$).

For each set, we then perform multiple runs using different random seeds, producing different simulation output for each run. These runs can be called Monte Carlo replication runs since the random seeds are themselves varied in a random fashion. The seed is set randomly based on the system time at launch time, and then stored for later replication of the data if needed.

Data heterogeneity The data is heterogeneous across several dimensions:

- agent types: there can be many different agent types (e.g., household agents, firm agents, bank agents, etc.)
- agent instances: there can be a different number of individual agents (called agent instances) per agent type.
- agent memory variables: there can be a different number of memory variables per agent type (but all agents of the same type have the same set of memory variables, specified a priori in the model.xml file that fully specifies the model's structure).

Due to this large data heterogeneity the file sizes may vary across simulations with the same model, even when using exactly the same input file, due to stochasticity.

Data dimensions in the XML output (listed in the order in which data is being produced by FLAME):

```
1 Sets
2 Runs
3 Iterations
4 Agent types
5 Agent instances per type
6 Variables
```

However, due to several conditions we have to impose on the data structure, the order in which data should be stored in the HDF5 file format differs from the order above. There are two restrictions:

1. For the HDF5 file format it is important that the atomic data set at the lowest hierarchical level is a homgeneous data format. This means that the choice of the 6 dimensions above requires us to choose those dimensions that remain invariant across all model simulations. These dimensions are: 5 Agent instances, 3 Iterations, 6 Variables. These dimensions are invariant because we simulate the same model many times, and we do not change the model structure across simulations. Therefore the number of variables per agent remains the same, the number of agent instances (individual agents) per agent type is constant, and the total number of iterations also remains constant across simulation runs.

2. The 3D DataFrame format in Python pandas is specified as row-major. This means that the 3D data frame requires the largest dimension to be on the major axis (recall it has 3 axis; item, major and minor). In our case, the largest dimension is the number of iterations, typically 1000 or higher. The other dimensions are the number of agent instances (100), and the number of variables (100).

Therefore, we specify the 3D DataFrame with:

- item axis: agent instances
- major axis (table rows): iterations
- minor axis (table columns): variables

Data dimensions in the HDF5 file (listed in the order in which data is stored in the HDF5 files)

Agent type: HDF5 filename (eg., Bank.h5)

Hierarchy inside the HDF5 file:

```
1 Sets: data group
2 Runs: data set inside data group
```

Hierarchy inside the pandas 3D DataPanel:

```
3 Agent instances
4 Iterations
5 Variables
```

File format specifications

Step 1: XML output This is the native format in which FLAME generates output. Each iteration produces an XML file that contains a full snapshot of all agents, and all agent memory variables. This can be a large file per iteration, so therefore FLAME also provides the possibility to only output one XML file at a certain frequency (using the command 'main -f freq', where main is the simulator executable and freq is an integer number that specifies at which periodicity the output should be generated). To further subsample the data outputted, it is possible to select only a subset of agents to output, or even to specify a **shadow model xml file** with only a subset of agent variables. Whatever method chosen, the data format is XML with fully tagged variables for each individual agent. Since the XML tags are rather verbose and redundant (of the format: `'<variable_name>value</variable_name>'`). This format can be reduced drastically by extracting the data to a more structured data format.

Step 2: HDF5-per-run files In this step we store data in one HDF5-per-run file per simulation run.

Each simulation run in FLAME produces a set of XML files, one file per iteration. This set can be tansformed into a flat table format, with iteration number on the table rows, and all variables of an individual agent on the table columns.

This would produce many tables, as many as there are individual agents. To reduce this further, we structure the data using the following dimensions (see also above, Data dimensions in the HDF5 file):

File hierarchy:

```
1 Set
2 Run
4 Agent type
```

Inside each HDF5-per-run file:

```
5 Agent instance
3 Iteration
6 Variable
```

Step 3: HDF5-per-agent-type files In this step we store data in one HDF5-per-agent-type file. this aggregates across all the files generated at the previous step 2 (all HDF5-per-run files).

Structure:

```
HDF5 file:
Data group: "Sets" (parameter setting)
Data set:   "Runs" (replications) stored as 3D DataPanel

3D DataPanel:
item axis: agent instances
major axis (table rows): iterations
minor axis (table columns): variables
```