
TrafPy

Release 0.0.1

Christopher W. F. Parsonson

Aug 23, 2022

CONTENTS:

1	Getting Started	3
2	Free Software	5
3	Documentation	7
4	Index	73
	Python Module Index	75
	Index	77

TrafPy is a Python package for the generation, management and standardisation of network traffic.

TrafPy contains the following key modules:

- **trafpy.generator**: A package for generating custom and/or literature traffic trace data which can be exported into universally compatible file formats (e.g. CSV, Pickle, JSON, etc.) and imported into any simulation, emulation, or experimentation environment. It also comes with an interactive Jupyter Notebook tool for visually building and shaping distributions.
- **trafpy.benchmark**: A package for generating, reproducing, and establishing standard network traffic benchmarks.
- **trafpy.manager**: A package for simulating a data centre network with various routing and scheduling protocols following the standard OpenAI Gym reinforcement learning interface.

TrafPy can be used to quickly and easily replicate traffic distributions from the literature even in the absence of raw open-access data. Furthermore, it is hoped that TrafPy will help towards standardising the traffic patterns used by networks researchers to benchmark their management systems.

GETTING STARTED

Follow the [instructions](#) to install TraffPy, then have a look at the [tutorial](#) and the [examples](#) on the GitHub page.

FREE SOFTWARE

TrafPy is free software; you can redistribute it and/or modify it under the terms of the Apache License 2.0. Contributions are welcome. Check out the [guidelines](#) on how to contribute. Contact cwfparsonson@gmail.com for questions.

DOCUMENTATION

3.1 Install

Open Git Bash. Change the current working directory to the location where you want to clone this [GitHub](#) project, and run:

```
$ git clone https://github.com/cwfpersonson/trafpy
```

In the project's root directory, run:

```
$ python setup.py install
```

Then, still in the root directory, install the required packages with either pip:

```
$ pip install -r requirements/default.txt
```

or conda:

```
$ conda install --file requirements/default.txt
```

You should then be able to import TrafPy into your Python script from any directory on your machine:

```
>>> import trafpy
```

3.2 Tutorial

This guide can help you start with TrafPy. For more detailed examples, see the [examples](#) directory of the TrafPy [GitHub](#). It is recommended that you both read this guide and then look at the basic GitHub examples to learn how to use TrafPy.

Note: TrafPy users who are not familiar with Python can read the whole *TrafPy Generator* section to understand how some of the TrafPy functions work. However, they may find it more useful to skip to the *Visually Shaping TrafPy Distributions* section, which describes how to use the TrafPy Jupyter Notebook as a stand-alone tool (universally compatible file formats (CSV, JSON, Pickle, etc.) can be generated and imported into e.g. MATLAB scripts; no Python knowledge is required), and also to look at the simple GitHub examples.

3.2.1 TrafPy Generator

The TrafPy Generator can generate custom and/or literature network traffic.

TrafPy Distributions

Network traffic patterns can be characterised by **distributions**. By accurately describing a distribution, one can sample from it to generate arbitrary amounts of realistic network traffic. This is fundamentally how TrafPy generates traffic; by sampling from pre-defined discrete distributions.

TrafPy distributions are defined as **hashtables** (Python dictionaries). These tables map each possible value taken by the random variable to some fractional value between 0 and 1 (where this value could be e.g. probability, fraction of network requested, etc.). The fractional values should sum to 1.0.

E.g. If you have integer random variable values 1-10 each with an equal probability of occurring, this distribution would be represented by the following hash table in TrafPy:

```
dist = {1: 0.1, 2: 0.1, 3: 0.1, 4: 0.1, 5: 0.1, 6: 0.1, 7: 0.1, 8: 0.1, 9: 0.1, 10: 0.1}
```

Similarly, a distribution where a random variable value of 5 always occurs would be:

```
dist = {5: 1.0}
```

and so on.

TrafPy distributions are typically referred to as either **value distributions** which map random variables such as flow size and inter-arrival times to ‘probability of occurring’, or **node distributions** which map node pairs to ‘fraction of the overall traffic load requested’.

Value Distributions

To import the TrafPy generator, run:

```
>>> import trafpy.generator as tpg
```

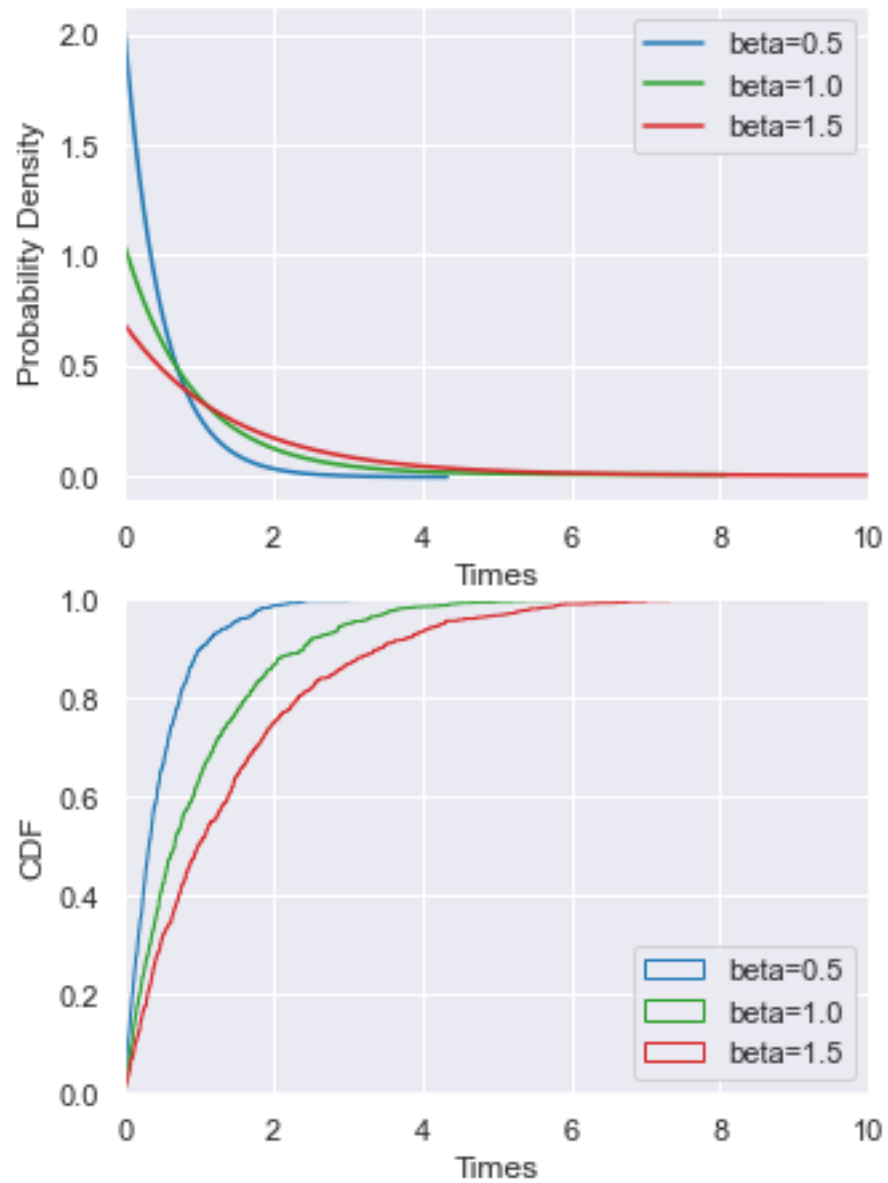
The most simple probability distribution for random variable values is the **uniform distribution**, where each random variable value has an equal probability of occurring:

```
>>> prob_dist, rand_vars, fig = tpg.gen_uniform_val_dist(min_val=0, max_val=100, round_
↳ to_nearest=1, return_data=True, show_fig=True, num_bins=101)
```

Demand characteristics of real network traffic patterns are rarely uniform. However, they can often be described by certain well-defined **named distributions**. These named distributions are themselves characterised by just a few parameters, making them easy to reproduce.

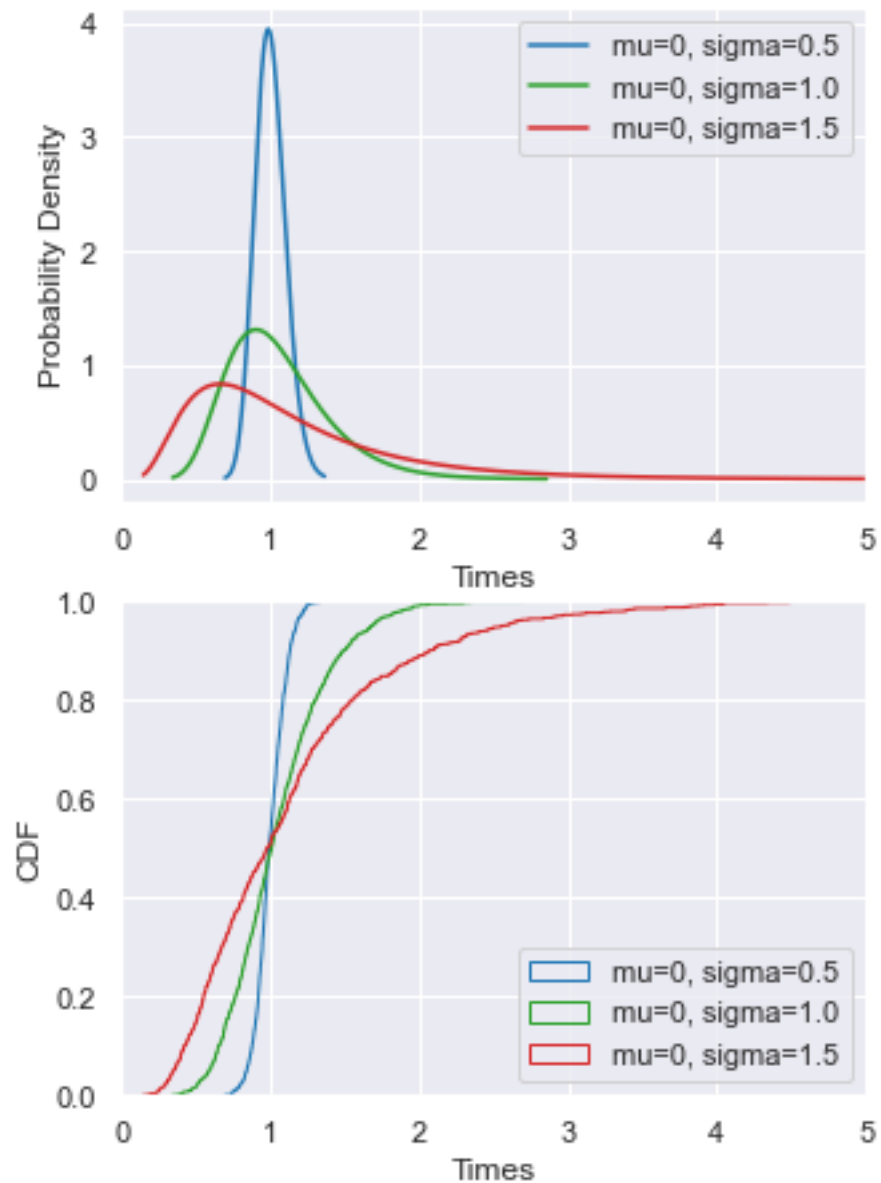
Named distributions supported by TrafPy include the *exponential distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='exponential', params={'_beta
↳ ': 1.0}, return_data=True, show_fig=True, xlim=[0,10], num_bins=101)
```



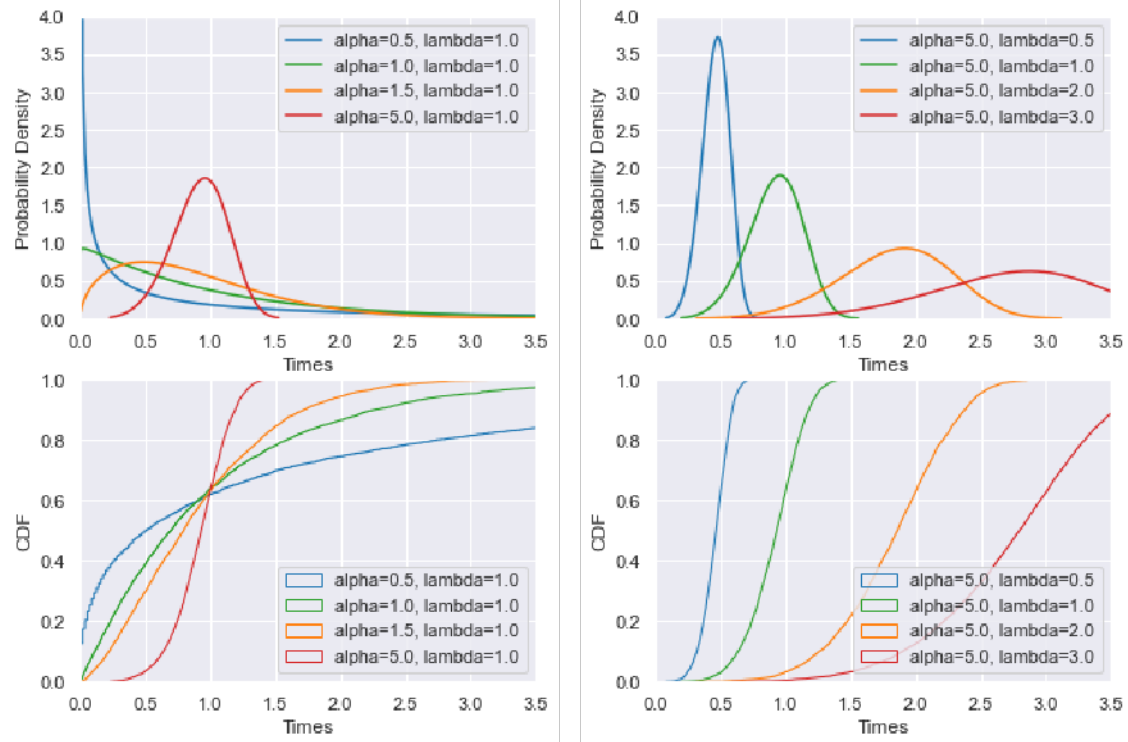
the *log-normal distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='lognormal', params={'_mu': 0,
↳ '_sigma': 1.0}, return_data=True, show_fig=True, xlim=[0,5], num_bins=10000)
```



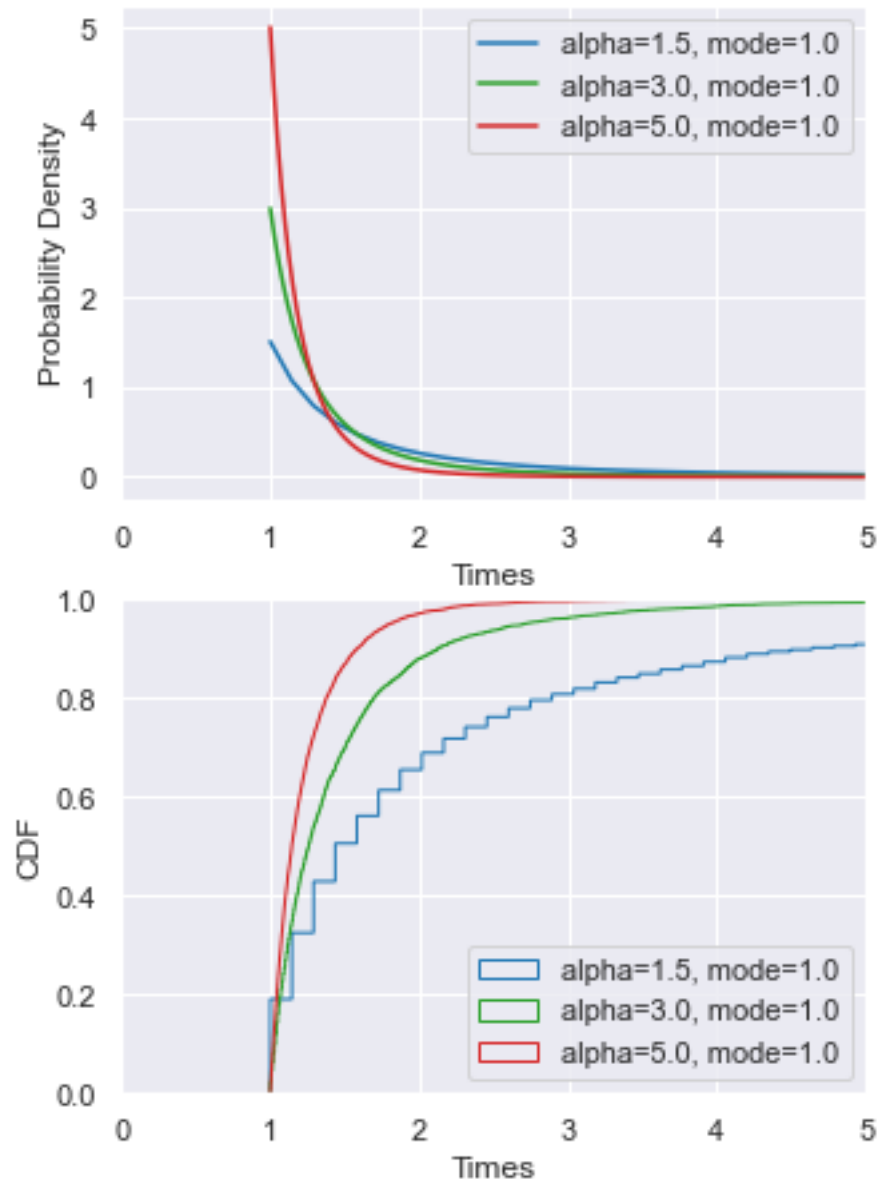
the *Weibull distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='weibull', params={'_alpha': 1.5,
↳ '_lambda': 1.0}, return_data=True, show_fig=True, xlim=[0,3.5], num_bins=101)
```



and the *Pareto distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='pareto', params={'_alpha': 3.0, '_mode': 1.0}, return_data=True, show_fig=True, xlim=[0,5], num_bins=101)
```



However, some demand characteristics cannot be accurately described by these named distributions. Instead, they might be described by arbitrarily **skewed** distributions which may or may not have some amount of **skewness** and/or **kurtosis**

```
>>> prob_dist, rand_vars, fig = tpg.gen_skewnorm_val_dist(location=50, skew=-5, scale=10,
↪ return_data=True, show_fig=True, num_bins=15)
```

Note that setting `skew=0` simply generates a **normal** distribution with standard deviation scale and mean location

```
>>> prob_dist, rand_vars, fig = tpg.gen_skewnorm_val_dist(location=50, skew=0, scale=10,
↪ return_data=True, show_fig=True, num_bins=15)
```

You can also use TrafPy to generate **multimodal** distributions with an arbitrary number of modes

```
>>> prob_dist, rand_vars, fig = tpg.gen_multimodal_val_dist(min_val=10,max_val=7000,
```

(continues on next page)

(continued from previous page)

```
↪ locations=[20,4000],skews=[6,-1],scales=[150,1500],num_skew_samples=[10000,650],bg_
↪ factor=0.05,return_data=True,show_fig=True,logscale=True,xlim=[10,10000],num_bins=18)
```

Later in this tutorial, you will see how to visually shape a multimodal distribution using TrafPy, allowing for almost any distribution to be generated.

Once you have your value distribution, you can use it to generate as many random variable values as you like

```
>>> rand_vars = tpg.gen_rand_vars_from_discretised_dist(unique_vars=list(prob_dist.
↪ keys()),probabilities=list(prob_dist.values()),num_demands=1000)
```

Node Distributions

```
>>> import trafpy.generator as tpg
```

Network traffic travels from a **source** node to a **destination** node. Source-destination nodes are **endpoints** in a network

```
>>> endpoints = ['server_'+str(i) for i in range(5)]
```

How regularly each node is selected as a source or destination is determined by a **node distribution matrix**. The elements in this matrix might refer to probabilities, but in TrafPy they usually refer to **load fractions** (i.e. what fraction of the overall traffic arriving is requested by a particular node pair).

The most simple node distribution is the **uniform distribution**

```
>>> node_dist, fig = tpg.gen_uniform_node_dist(eps=endpoints, show_fig=True)
```

Since different endpoint nodes in a network likely have different hardware capabilities, network node distributions are rarely uniform. Instead, some nodes become ‘hot nodes’ and are requested more than others, forming a **multimodal node distribution**

```
>>> node_dist, fig = tpg.gen_multimodal_node_dist(eps=endpoints, skewed_nodes=['server_2
↪'], show_fig=True)
```

Instead of certain *nodes* being requested more regularly, sometimes certain *node pairs* in the network might be skewed, forming a **multimodal node pair distribution**

```
>>> node_dist, fig = tpg.gen_multimodal_node_pair_dist(eps=endpoints, skewed_pairs=[[
↪ 'server_1','server_3'], ['server_4','server_2']], show_fig=True)
```

N.B. The above graph plots are **chord diagrams**. Each network end point is a node, and the colour of the node indicates how much of the overall traffic requestes that particular end point. The width of edges between nodes indicates how much traffic is travelling between a particular endpoint pair, with pair edges below a certain load threshold being excluded from the plot for visual clarity. Chord diagrams are just an alternative way of visualising the more standard 2D traffic matrix.

Different networks have different node distributions. Sometimes you may want a simple uniform distribution, or a slightly skewed distribution, or certain nodes being heavily in demand, or certain node pairs being heavily in demand. Furthermore, you may want all of the above, but may also want to specify certain things yourself (e.g. which specific nodes/pairs to bias, how high demand they’re in, how many nodes are in high demand etc.), or you may want these specifics to be randomly generated. The above functions handle all of the above functionality. See their documentation for further details.

You can create any size of node distribution you like to fit any network

```
>>> endpoints = ['server_'+str(i) for i in range(64)]
>>> node_dist, fig = tpg.gen_multimodal_node_pair_dist(eps=endpoints, show_fig=True)
```

Network endpoints/servers are often grouped into physically local clusters or ‘racks’. Different networks may have different levels of inter- (between) and intra- (within) rack communication. One way to specify this would be to set individual node pair probabilities with the `gen_multimodal_node_pair_dist` function you’ve already seen, however this would be inconvenient and laborious. Instead, when using the above node distribution functions, you can specify the `rack_prob_config` argument, which allows you to set the proportion of traffic which should be inter-rack. TrafPy will then use your shaped node distribution to create an adjusted node distribution which accounts for your specified rack probabilities. For example, if you specify `rack_prob_config` in `gen_uniform_node_dist`, you will not generate a perfectly uniform node distribution as you would if you left `rack_prob_config` as `None`, but instead a node distribution with set inter- and intra-rack probabilities sampled from a uniform distribution. You will need to specify which endpoints are in which rack with a dictionary (this is automatically done for you if you use one of the TrafPy networks). E.g. Making 10% of traffic inter-rack in a fat-tree topology:

```
>>> net = tpg.gen_fat_tree(k=4, n=8)
>>> fig = tpg.plot_network(net, draw_node_labels=True, network_node_size=1000)
>>> print('Racks dict:\n{}'.format(net.graph['rack_to_ep_dict']))
Racks dict:
{'rack_0': ['server_0', 'server_1', 'server_2', 'server_3'], 'rack_1':
['server_4', 'server_5', 'server_6', 'server_7'], 'rack_2': ['server_8',
'server_9', 'server_10', 'server_11'], 'rack_3': ['server_12', 'server_13',
'server_14', 'server_15']}
```

```
>>> rack_prob_config = {'racks_dict': net.graph['rack_to_ep_dict'], 'prob_inter_rack': 0.
↪ 10}
>>> node_dist, _ = tpg.gen_uniform_node_dist(net.graph['endpoints'], rack_prob_
↪ config=rack_prob_config, show_fig=True, print_data=False)
```

Making 90% of traffic inter-rack:

```
>>> rack_prob_config = {'racks_dict': net.graph['rack_to_ep_dict'], 'prob_inter_rack': 0.
↪ 90}
>>> node_dist, _ = tpg.gen_uniform_node_dist(net.graph['endpoints'], rack_prob_
↪ config=rack_prob_config, show_fig=True, print_data=False)
```

Once you have your node distribution, you can use it to generate as many source-destination node pairs as you like

```
>>> sn, dn = tpg.gen_node_demands(eps=net.graph['endpoints'], node_dist=node_dist, num_
↪ demands=1000)
```

Networks

```
>>> import trafpy.generator as tpg
```

By definition, a network is a collection of nodes (vertices) which together form pairs of nodes connected by links (edges). Some or all of these nodes can act as **sources** and **destinations** for network traffic **demands**. Such network nodes are referred to as **endpoints**. Endpoints might be separated by multiple links and nodes, some of which may be endpoints and some not.

Generate a simple 5-node network

```
>>> network = tpg.gen_simple_network(ep_label='server', show_fig=True)
```

or the 14-node NSFNET network

```
>>> network = tpg.gen_nsfnet_network(ep_label='server', show_fig=True)
```

or a fat-tree network

```
>>> network = tpg.gen_fat_tree(k=4, show_fig=True)
```

A key feature of TrafPy is that it can generate traffic for any network. If your network does not fall into one of the above networks (which is likely that it will not), you should use the `trafpy.generator.gen_arbitrary_network` function to generate your network. This will generate an arbitrary network given your number of end points, but will format the network in a way recognised by TrafPy.

```
>>> network = tpg.gen_arbitrary_network(num_eps=10)
```

Flow-Centric Traffic Demands

```
>>> import trafpy.generator as tpg
```

A single demand in a network can be considered as either a **flow** or a computation graph (a **job**) whose dependencies (edges) may form flows. Both flow-centric and job-centric network traffic demand generation and management are supported by TrafPy.

A flow is some information being sent from a source node to a destination node in a network (e.g. a data centre network).

Common flow demand characteristics include:

- size;
- interarrival time; and
- source-destination node distribution.

Using the value and node distribution generation functions you've seen so far, you can use TrafPy to generate realistic flow demands. Later in this tutorial, you will see how to use TrafPy's Jupyter Notebook tool to visually shape your distributions such that they match real data/literature distributions. For now, assume that you already know the distribution parameters you want. Consider that you want to create 1,000 realistic data centre flows in a simple 5-node network

```
>>> network = tpg.gen_simple_network(ep_label='endpoint', show_fig=True)
```

You could start by defining the flow size distribution

```
>>> flow_size_dist, _ = tpg.gen_named_val_dist(dist='weibull', params={'_alpha': 1.4, '_lambda': 7000}, show_fig=True, rand_var_name='Flow Size', logscale=True, round_to_nearest=25, xlim=[1e2, 1e12], num_bins=15)
```

then the flow interarrival time distribution

```
>>> interarrival_time_dist, _ = tpg.gen_named_val_dist(dist='weibull', params={'_alpha': 0.9, '_lambda': 6000}, rand_var_name='Interarrival Time', min_val=1, round_to_nearest=25, show_fig=True, logscale=True, print_data=False, num_bins=15)
```

and then the source-destination node distribution

```
>>> endpoints = network.graph['endpoints']
>>> node_dist = tpg.gen_multimodal_node_dist(eps=endpoints,num_skewed_nodes=1,show_
↳fig=True)
```

The network load refers to the overall amount of traffic received by the network. This is commonly referred to as a load rate (information units arriving per unit time) or as a load fraction (the fraction of the total network capacity being requested for a given duration). TrafPy typically uses the load fraction definition for load, therefore loads can be varied between 0 and 1.

A key feature of TrafPy is that you can generate any load for your custom network. To do this, you should provide TrafPy with a `network_load_config` dictionary which tells TrafPy (1) the end point capacity of your network, (2) the maximum capacity of your network, and (3) the overall load fraction you would like TrafPy to generate for your network. Consider that you would like TrafPy to generate a 0.1 load traffic trace for your network (i.e. around 10% of your total network capacity will be requested per unit time):

```
>>> network_load_config = {'network_rate_capacity': network.graph['max_nw_capacity'],
↳'ep_link_capacity': network.graph['ep_link_capacity'], 'target_load_fraction': 0.1}
```

You can then use your distributions and load config to generate flow-centric demand data formatted neatly into a single dictionary:

```
flow_centric_demand_data = tpg.create_demand_data(eps=endpoints,node_dist=node_dist,flow_
↳size_dist=flow_size_dist,max_num_demands=1000,interarrival_time_dist=interarrival_time_
↳dist,network_load_config=network_load_config)
```

Don't forget to save your data as a pickle:

```
tpg.pickle_data(data=flow_centric_demand_data,path_to_save='data/flow_centric_demand_
↳data.pickle',overwrite=True,zip_data=True)
```

or as a csv:

```
tpg.save_data_as_csv(data=flow_centric_demand_data,path_to_save='data/flow_centric_
↳demand_data.csv',overwrite=True)
```

N.B. You can also re-load previously pickled data:

```
flow_centric_demand_data = tpg.unpickle_data(path_to_load='data/flow_centric_demand_data.
↳pickle',zip_data=True)
```

TrafPy flow-centric demand data dictionaries are organised as:

```
{
  'flow_id': ['flow_0', ..., 'flow_n'],
  'sn': [flow_0_sn, ..., flow_n_sn],
  'dn': [flow_0_dn, ..., flow_n_dn],
  'flow_size': [flow_0_size, ..., flow_n_size],
  'event_time': [event_time_flow_0, ..., event_time_flow_n],
  'index': [index_flow_0, ..., index_flow_1]
}
```

Job-Centric Traffic Demands

```
>>> import trafpy.generator as tpg
```

A job is a task sent to a network (such as a data centre) to execute. Jobs are computation graphs made up of **operations** (ops). Jobs might be e.g. a Google search query, generating a user's Facebook feed, performing a TensorFlow machine learning task (e.g. backpropagation), etc.

In this context, an op is a data process ran on some machine where the result is specified by a pre-determined rule/programme. Each op requires ≥ 0 tensors/data objects as input, and produces ≥ 0 tensors as output.

In a job computation graph, if an op v requires ≥ 1 input(s) produced by op u , the ops will be connected by a directed edge, $[u, v]$, representing the **dependency** between the two ops. The edge attributes here are features of the tensor (e.g. size, source machine, destination machine, etc.).

In a data centre, when a job arrives, each op in the job is placed onto some machine to execute the op. These ops might be placed all on one machine or, as is often the case for many applications, spread out across different machines in the network according to e.g. some heuristic. The **network** is used to pass the tensors around between the machines executing the ops. These tensors/data objects flowing between ops are **flows**. The flows of a given job might flow through the network at the same time or at different times depending on e.g. scheduling decisions, constraints, dependencies, etc.

Note: In a job graph, edges between ops represent 1 of 2 types of op dependency:

- **Data dependency:** Op j can only begin when op i 's output tensor(s) have arrived. Therefore, data dependencies become network flows *if* op j and op i are ran on separate network endpoints.
- **Control dependency:** Op j can only begin when op i has finished. No data is exchanged, therefore control dependencies never become network flows.

Common job demand characteristics include:

- job interarrival time;
- which machine each op in the job is placed on;
- number of ops in the job;
- run times of the ops;
- size of data dependencies (flows) between ops;
- ratio of control to data dependencies in job computation graph; and
- connectivity of job graph.

You can use the same value and node distributions as before to generate realistic job demands. The only difference is that now you will pass additional arguments into `tpg.create_demand_data()`. TrafPy will respond by generating job computation graphs rather than flows as the demands in the returned dictionary.

Consider that you want to create 10 realistic data centre jobs in the same simple 5-node network as before (but now omitting `show_fig` to save page space).

```
>>> network = tpg.gen_simple_network(ep_label='endpoint')
>>> network_load_config = {'network_rate_capacity': network.graph['max_nw_capacity'],
↪ 'ep_link_capacity': network.graph['ep_link_capacity'], 'target_load_fraction': 0.1}
```

You could start by defining the flow size distribution of the flows inside the job graphs

```
>>> flow_size_dist = tpg.gen_multimodal_val_dist(min_val=1,max_val=100,locations=[50],
↳skews=[0],scales=[10],num_skew_samples=[10000],bg_factor=0,round_to_nearest=1,num_
↳bins=34)
```

then the job interarrival time distribution

```
>>> interarrival_time_dist = tpg.gen_multimodal_val_dist(min_val=1,max_val=1e8,
↳locations=[1,1,3000,1,1800000,100000000],skews=[0,100,-10,10,50,6],scales=[0.1,62,2000,
↳7500,3500000,200000000],num_skew_samples=[800,1000,2000,4000,4000,3000],bg_factor=0.025,
↳round_to_nearest=1)
```

then the number of ops in each job

```
>>> num_ops_dist = tpg.gen_multimodal_val_dist(min_val=50,max_val=200,locations=[100],
↳skews=[0.05],scales=[50],num_skew_samples=[10000],bg_factor=0.05,round_to_nearest=1)
```

and then the source-destination node (i.e. op machine placement) distribution

```
>>> endpoints = network.graph['endpoints']
>>> node_dist = tpg.gen_multimodal_node_dist(eps=endpoints,num_skewed_nodes=1)
```

You can then use your distributions to generate your job-centric demand data returned neatly into a single dictionary:

```
>>> job_centric_demand_data = tpg.create_demand_data(eps=endpoints,node_dist=node_dist,
↳flow_size_dist=flow_size_dist,interarrival_time_dist=interarrival_time_dist,num_ops_
↳dist=num_ops_dist,c=1.5,max_num_demands=10,network_load_config=network_load_config,use_
↳multiprocessing=False)
```

Don't forget to save your data:

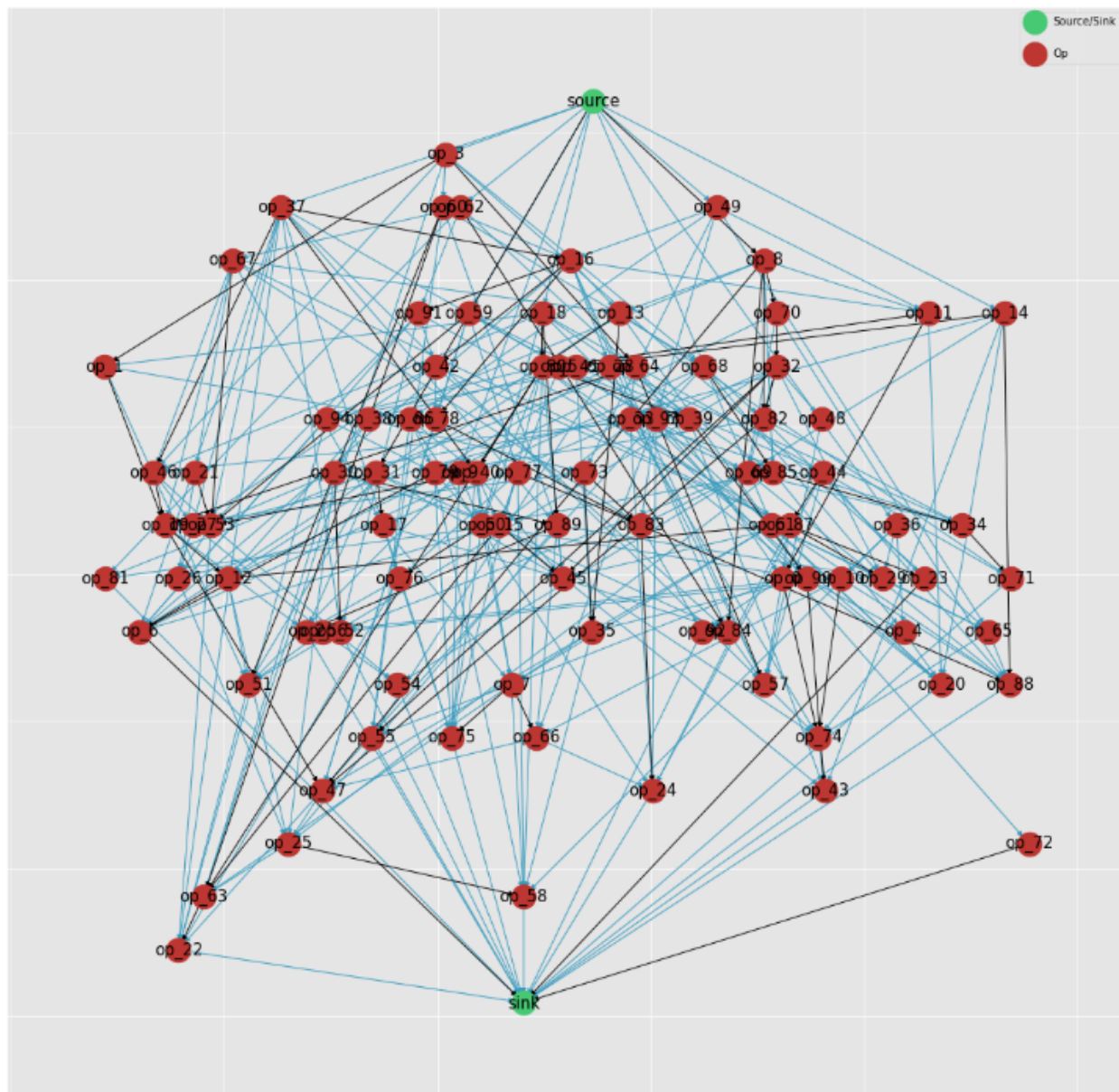
```
tpg.pickle_data(data=job_centric_demand_data,path_to_save='data/job_centric_demand_data.
↳pickle',overwrite=True,zip_data=True)
```

TrafPy job-centric demand data dictionaries are organised as:

```
{
  'job_id': ['job_0', ..., 'job_n'],
  'job': [networkx_graph_job_0, ..., networkx_graph_job_n],
  'event_time': [event_time_job_0, ..., event_time_job_n],
  'establish': [event_establish_job_0, ..., event_establish_job_1],
  'index': [index_job_0, ..., index_job_1]
}
```

Where the 'job' key contains the list of job computation graphs with all the embedded demand data. You can visualise the job computation graph(s):

```
>>> jobs = list(job_centric_demand_data['job'][0:2])
>>> fig = tpg.draw_job_graphs(job_graphs=jobs,show_fig=True)
```



of the generated distributions continuously output to aid accuracy.

Navigate to the directory where you cloned TrafPy and launch [the Jupyter Notebook](#):

```
$ jupyter-notebook main.ipynb
```

The Notebook has a few main sections with markdown descriptions for each:

- Import `trafpy.generator`
- Set global variables
- Generate random variables from 'named' distribution
- Generate random variables from arbitrary 'multimodal' distribution
- Generate discrete probability distribution from random variables
- Generate random variables from discrete probability distribution
- Generate source-destination node distribution
- Use node distribution to generate source-destination node demands
- Use previously generated distributions to create single 'demand data' dictionary
- Generate distributions in sets (extension)

All of the above sections can be used together or independently depending on which functionalities you need to shape your specific distribution. Below are demonstrations of how to use the interactive distribution-shaping cells.

Note: To run a Jupyter Notebook cell, click on the cell and click 'Run' on the top ribbon. If you are running a cell with a TrafPy interactive graph, some configurable parameters will appear. Adjust these parameters and click the Run Interact button to update your plot (and the returned values).

Note: Once you have shaped your distribution, you can simply plug your shaped parameters into the previously described functions to generate your required random variable data/distributions in your own scripts. I.e. There is no need to have to save your Notebook data if you note down your shaped parameters and enter them into your own TrafPy scripts.

Set Global Variables

Set the `PATH` global variable to the directory where you want any data generated with the Notebook to be saved. You can also set the `NUM_DEMANDS` global variable, which will ensure that each time you shape a distribution for a certain traffic demand characteristic, the correct number of demands will be generated.

Set Global Variables

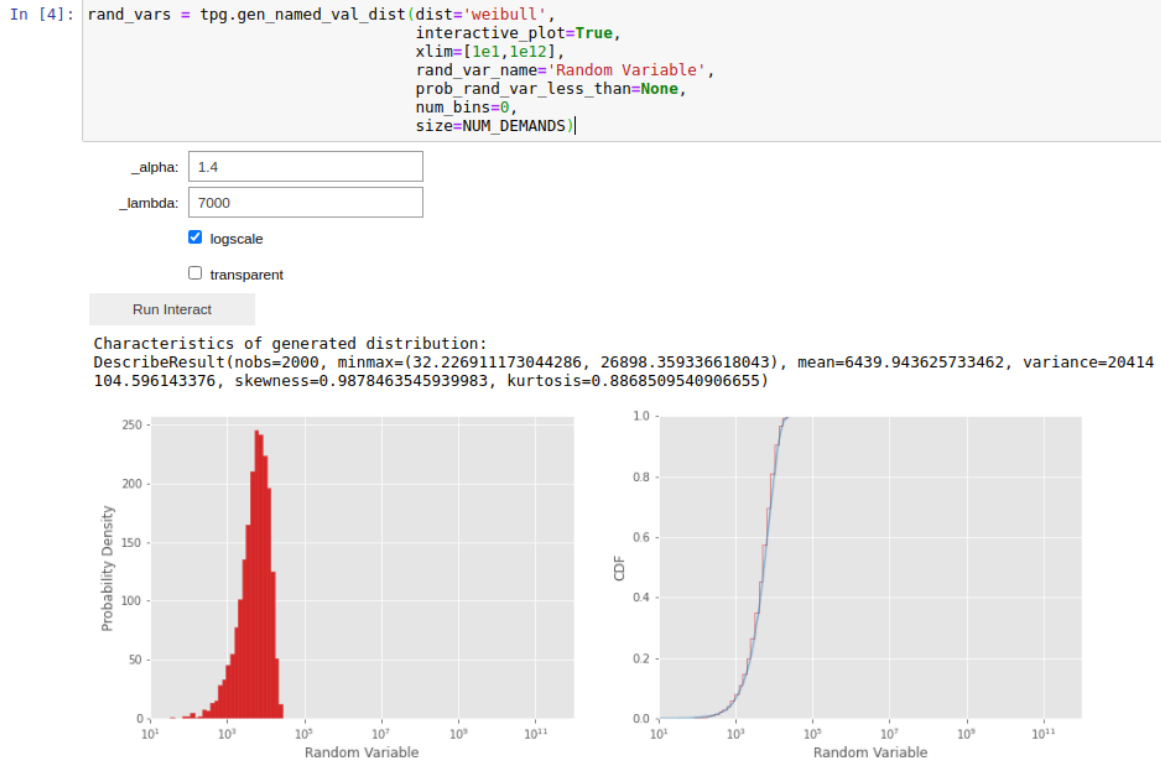
Set global vars. To change a global a global var, edit this cell and re-run the cell.

- `PATH`: The path to the folder where you want to save and/or load data.
- `NUM_DEMANDS`: Number of demands to generate.

```
In [4]: PATH = r'data/interactive_test/'
        NUM_DEMANDS = 2000
```

Generate Random Variables from ‘Named’ Distribution

Use this section to shape the previously described ‘named’ value distributions (Pareto, Weibull, etc.) generated by `trafpy.gen_named_val_dist()`.



Generate Random Variables from Arbitrary ‘Multimodal’ Distribution

Use this section to shape the previously described ‘multimodal’ value distribution generated by `trafpy.gen_multimodal_val_dist()`.

There are a few steps to generating a multimodal distribution with TrafPy:

1. Define the random variables of your multimodal distribution. Set the minimum and maximum possible values, the number of modes, the name of your random variable, the x-axis limits, what to round the values to, and how many decimal places to include. Run the 1st cell.

Generate Random Variables from Arbitrary 'Multimodal' Distribution

In previous cells we considered standard distributions (exponential, lognormal, weibull, pareto...). These are common distributions which occur in many different scenarios. However, sometimes in real scenarios distributions might not fall into these well-defined distribution categories.

Multimodal distributions are distributions with ≥ 2 different modes. A multimodal distribution with 2 modes is a special case called a 'bimodal distribution', which is very common.

The traffic toolbox allows you to generate arbitrary multimodal distributions. This is very powerful because with access to the above standard distributions and the arbitrary multimodal distribution generator, any distribution can be generated if you are able to shape it sufficiently.

Generating multimodal distributions is a little more involved than generating the standard distributions was, but it can still be done in a matter of seconds using this notebook's visualisation tool.

There are a few simple steps to generating an arbitrary multimodal distribution:

1. Decide the number of modes (i.e. peaks) and other distribution characteristics
2. Shape each mode individually
3. Combine all of modes together and add some 'background noise' to the distribution such that the modes are 'joined' together to form a single multimodal distribution (background noise can be set to 0 if desired)
4. Use your multimodal distribution to generate demands
5. Save the generated demands

```
In [8]: # 1. define distribution variables
min_val=10
max_val=7000
num_modes=2
xlim=[10,1000]
rand_var_name='Multimodal Random Variable'
round_to_nearest=1
num_decimal_places=1
```

2. Run the 2nd cell to launch the visualisation tool. A set of tuneable parameters for each mode (where you specified `num_modes` in the previous cell) will appear. Adjust the parameters and click **Run Interact** until you are happy with the shape of each mode. Use **Location** for the mode position, **Skew** for the mode skew, **Scale** for the mode standard deviation, **Samples** for the height of the mode's probability distribution, and **bins** for how many bins to plot (default of 0 automatically chooses number of bins).

```
In [4]: # 2. shape each mode
data_dict = tpg.gen_skew_dists(min_val=min_val,
                               max_val=max_val,
                               num_modes=num_modes,
                               xlim=xlim,
                               rand_var_name=rand_var_name,
                               round_to_nearest=round_to_nearest,
                               num_decimal_places=num_decimal_places)
```

Location:

Skew:

Scale:

Samples:

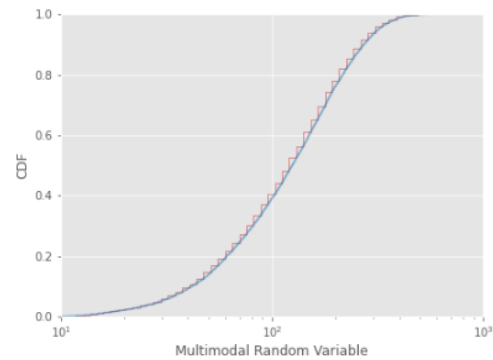
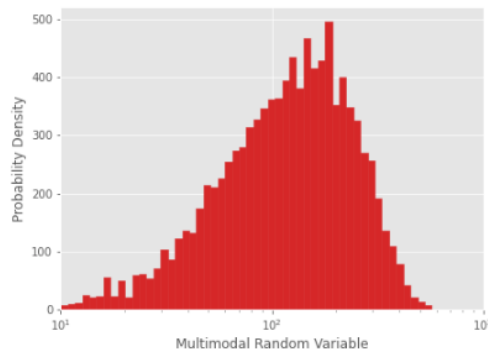
☒ logscale

☐ transparent

bins:

Run Interact

Characteristics of generated distribution:
DescribeResult(nobs=10000, minmax=(10.0, 617.0), mean=142.9147, variance=8362.637887698771, skewness=0.9731968325099416, kurtosis=0.8121307203415284)



Location:

Skew:

Scale:

Samples:

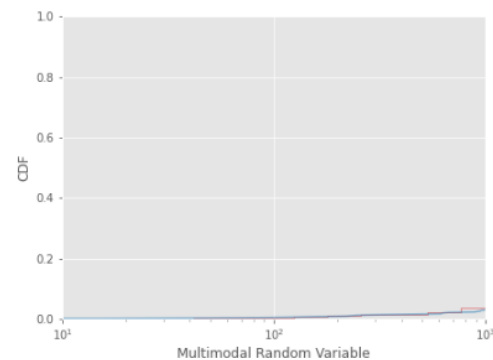
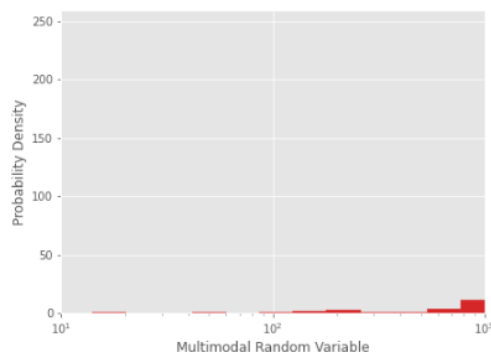
☒ logscale

☐ transparent

bins:

Run Interact

Characteristics of generated distribution:
DescribeResult(nobs=650, minmax=(14.0, 6783.0), mean=3198.8384615384616, variance=1496743.53164632, skewness=-0.003905559579357388, kurtosis=-0.18372149442452423)



- Run the 3rd cell to combine the above modes. Adjust `bg_factor` to increase or decrease the ‘background noise’ amongst your shaped nodes.

In [6]: *# 3. combine modes to form multimodal probability distribution*

```
multimodal_prob_dist = tpg.combine_multiple_mode_dists(data_dict,
min_val=min_val,
max_val=max_val,
xlim=xlim,
rand_var_name=rand_var_name,
round_to_nearest=round_to_nearest,
num_decimal_places=num_decimal_places)
```

bg_factor

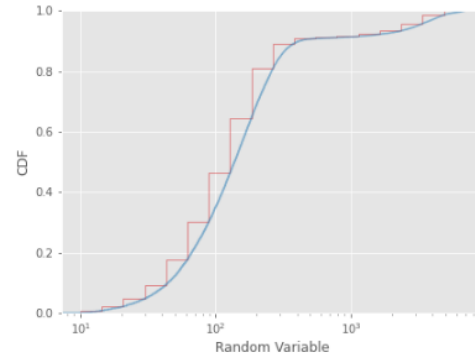
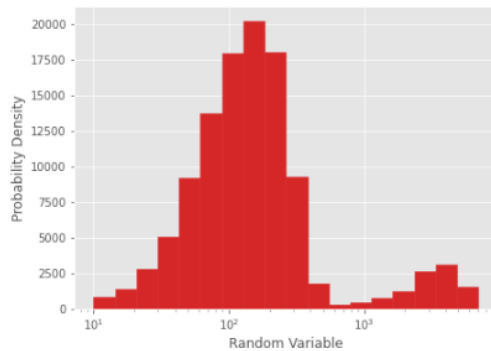
☒ logscale

☐ transparent

bins:

Run Interact

Chosen skew 1 stats: {'location': 20, 'skew': 6.0, 'scale': 150.0, 'min_val': 10, 'max_val': 7000, 'num_skew_samples': 10000, 'xlim': [10, 1000], 'logscale': True, 'transparent': False, 'rand_var_name': 'Multimodal Random Variable', 'num_bins': 0, 'round_to_nearest': 1, 'num_decimal_places': 1}
Chosen skew 2 stats: {'location': 4000, 'skew': -1.0, 'scale': 1500.0, 'min_val': 10, 'max_val': 7000, 'num_skew_samples': 650, 'xlim': [10, 1000], 'logscale': True, 'transparent': False, 'rand_var_name': 'Multimodal Random Variable', 'num_bins': 0, 'round_to_nearest': 1, 'num_decimal_places': 1}
Characteristics of generated distribution:
DescribeResult(nobs=110667, minmax=(10.0, 6999.0), mean=438.1551320628552, variance=1076764.9295058353, skewness=3.756236025280682, kurtosis=14.160120264250057)



Note: You may find it useful to jump between the 2nd and 3rd cells to improve the accuracy of the modes relative to one-another.

- (Optional) Run the 4th cell to use your shaped multimodal distribution to sample random variable data.
- (Optional) Run the 5th cell to save your multimodal random variable data

3.2.2 TrafPy Benchmark

A key advantage of using TrafPy to generate network traffic is that, given just a few TrafPy parameters, you can re-generate any traffic trace for any network. This therefore provides a standardised method for benchmarking your own network systems. Useful TrafPy benchmark generation functionality is provided by the `trafpy.benchmarker` module.

Using Default TrafPy Benchmarks

To load the pre-defined default TrafPy benchmarks, use the `trafpy.benchmarker.BenchmarkImporter` class.

```
>>> from trafpy.benchmarker import BenchmarkImporter
```

Initialise a network of your choice for which you would like to generate some benchmark traffic for

```
>>> import trafpy.generator as tpg
>>> net = tpg.gen_fat_tree(k=4)
```

TrafPy is able to generate arbitrary distributions and therefore arbitrary traffic traces. As the nature of applications handled by real data centres changes, so too will the resultant traffic traces. Therefore, the initial benchmarks established for TrafPy have been stored under `benchmark_version='v001'` with the anticipation that future benchmark versions will be established with new and evolving traffic traces.

TrafPy benchmarks are generated from sets of TrafPy parameters rather than stored as data sets of flows. This enables the same benchmark to be used to generate the same traffic trace characteristics for different networks. By setting `load_prev_dists=False`, TrafPy will re-generate distributions for your network by sampling from these underlying TrafPy parameters. If you want to use the same distributions to generate new data sets to e.g. repeat some tests sampling from the same benchmark distributions, you should set `load_prev_dists=True`, however bare in mind that if the number of end points in your network has changed since you last generated the benchmark distributions with `load_prev_dists=False`, then TrafPy will raise an error.

Initialise the `trafpy.benchmarker.BenchmarkImporter`

```
>>> importer = BenchmarkImporter(benchmark_version='v001', load_prev_dists=False)
```

The `trafpy.benchmarker.BenchmarkImporter` can be used to import pre-defined default TrafPy benchmarks and generate their corresponding distributions for your custom network. To see the valid default benchmark names supported by the importer, run:

```
>>> print(importer.default_benchmark_names)
['commercial_cloud', 'jobcentric_prototyping', 'private_enterprise',
'rack_sensitivity_0', 'rack_sensitivity_02', 'rack_sensitivity_04',
'rack_sensitivity_06', 'rack_sensitivity_08', 'skewed_nodes_sensitivity_0',
'skewed_nodes_sensitivity_005', 'skewed_nodes_sensitivity_01',
'skewed_nodes_sensitivity_02', 'skewed_nodes_sensitivity_04',
'social_media_cloud', 'tensorflow', 'uniform', 'university']
```

Decide which default benchmarks you would like to generate for your system, and then simply call the `trafpy.benchmarker.BenchmarkImporter.get_benchmark_dists` function to get the node, flow size, and flow inter-arrival time distributions of this benchmark adapted to your custom network. For example, we can generate the 'university' benchmark distributions:

```
>>> dists = importer.get_benchmark_dists(benchmark_name='university', eps=net.graph[
↪ 'endpoints'], racks_dict=net.graph['rack_to_ep_dict'])
```

The distributions are returned as a dictionary

```
>>> print(dists.keys())
      'node_dist', 'flow_size_dist', 'interarrival_time_dist'
```

You can now use the `trafpy.generator.create_demand_data()` function as usual to use these benchmark distributions to generate traffic for your own network.

Creating Custom TrafPy Benchmarks

It is easy to create your own custom benchmarks in TrafPy using the `trafpy.benchmark.Benchmark` abstract base class. You need only share your child class code for others to be able to reproduce the traffic you generated for their own networks.

Import the `trafpy.benchmark.Benchmark` abstract parent class

```
>>> from trafpy.benchmark import Benchmark
```

All flow-centric demands have 3 key properties; flow size, flow interarrival time, and node distribution. Therefore, all child classes inheriting from `trafpy.benchmark.Benchmark` must define the abstract methods `get_node_dist`, `get_interarrival_time_dist`, and `get_flow_size_dist`.

Inside each of these methods, you should first call the `trafpy.benchmark.Benchmark` parent abstract method to initialise (1) the TrafPy distribution hashtable, and (2) the path in which to store the distribution hashtable (automatically decided by TrafPy). You should then implement your own code to generate the distribution hashtable (using what you've learned from the previous TrafPy tutorials and examples on the GitHub), before finally calling the `trafpy.benchmark.Benchmark.save_dist` abstract method to save the distribution.

For example, consider that you want to create a simple benchmark called 'my_benchmark' with a uniform node distribution, a constant flow size of 1 information unit, and a constant interarrival time of 10 time units:

```
import trafpy.generator as tpg

class MyBenchmark(Benchmark):
    def __init__(self, benchmark_name='my_benchmark', benchmark_version='v001', load_prev_dists=True):
        super(MyBenchmark, self).__init__(benchmark_name, benchmark_version, load_prev_dists)

    def get_node_dist(self, eps, racks_dict=None, dist_name='node_dist'):
        dist, path = super().get_node_dist(eps, racks_dict, dist_name)
        if dist is None or not self.load_prev_dists:
            dist = tpg.gen_uniform_node_dist(eps, show_fig=False, print_data=False)
            super().save_dist(dist, dist_name)
        return dist

    def get_interarrival_time_dist(self, dist_name='interarrival_time_dist'):
        dist, path = super().get_interarrival_time_dist(dist_name)
        if dist is None or not self.load_prev_dists:
            dist = {10: 1}
            super().save_dist(dist, dist_name)
        return dist

    def get_flow_size_dist(self, dist_name='flow_size_dist'):
        dist, path = super().get_flow_size_dist(dist_name)
```

(continues on next page)

(continued from previous page)

```

    if dist is None or not self.load_prev_dists:
        dist = {1: 1}
        super().save_dist(dist, dist_name)
    return dist

```

You can now use your MyBenchmark class to generate your 'my_benchmark' traffic for any network:

```

benchmark = MyBenchmark(benchmark_name='my_benchmark', benchmark_version='v001', load_
↳ prev_dists=False)

# init network
net = tpg.gen_fat_tree(k=4)

# generate my_benchmark distributions for this network
dists = {}
dists['node_dist'] = benchmark.get_node_dist(net.graph['endpoints'], racks_dict=None,
↳ dist_name='node_dist')
dists['interarrival_time_dist'] = benchmark.get_interarrival_time_dist(dist_name=
↳ 'interarrival_time_dist')
dists['flow_size_dist'] = benchmark.get_flow_size_dist(dist_name='flow_size_dist')

```

You can now use the `trafpy.generator.create_demand_data()` function as usual to use these benchmark distributions to generate traffic for your own network.

Warning: Note that to generate distribution data from your custom benchmark, you have not used the `trafpy.benchmark.BenchmarkImporter` class. This importer class is only for default TrafPy benchmarks.

Overwriting and Adding New Default TrafPy Benchmarks

Warning: This tutorial is for advanced TrafPy developers who want to commit their own benchmarks to the community by adding them to the set of default TrafPy benchmarks. If you simply want to make your own benchmark but not make it a default TrafPy benchmark, see the previous tutorial.

TrafPy benchmarks are stored in the `benchmarks/` directory of wherever TrafPy is installed on your machine. You can check this with:

```

>>> import os
>>> import trafpy
>>> benchmarks_path = os.path.dirname(trafpy.__file__) + '/benchmarker/versions/
↳ benchmark_v001/'
>>> print(benchmarks_path)
/home/cwfparsonson/Insync/zciccw@ucl.ac.uk/OneDriveBiz/ipes_cdt/phd_project/projects/
↳ trafpy/trafpy/benchmarker/versions/benchmark_v001/

```

Inside this directory will be a 2 key folders: the `benchmarks/` folder, which is where the `trafpy.benchmark.Benchmark` child classes are stored for the default TrafPy benchmarks, and the `data/` folder which is where any generated distribution hashtables are stored for each benchmark.

To add a new benchmark, simply open the `benchmarks/` folder and create a python file called `<benchmark_name>.py`.

Warning: The name of the file will determine the name of your default benchmark. E.g. to create a benchmark called 'university', you must create a file called `university.py`.

Inside this file, you can write your own default benchmark class inheriting from the `trafpy.benchmark.Benchmark` abstract parent class. The only constraint is that your class **must** be called `DefaultBenchmark`; failing to set the class variable to this will result in errors with the `trafpy.benchmark.BenchmarkImporter` class.

Warning: Your default benchmark class variable **must** be called `DefaultBenchmark`.

For example, this is how you might create a new default benchmark class called 'my_default_benchmark' in `benchmarks_path+'/'benchmarks/my_default_benchmark.py`:

```
from trafpy.benchmark.versions.benchmark import Benchmark
from trafpy.generator.src.dists import node_dists
from trafpy.generator.src.dists import val_dists
from trafpy.generator.src.dists import plot_dists

import math
import numpy as np

class DefaultBenchmark(Benchmark):
    def __init__(self, benchmark_name='my_default_benchmark', benchmark_version='v001',
    ↪load_prev_dists=True):
        super(DefaultBenchmark, self).__init__(benchmark_name, benchmark_version, load_
    ↪prev_dists)

    def get_node_dist(self, eps, racks_dict=None, dist_name='node_dist'):
        dist, path = super().get_node_dist(eps, racks_dict, dist_name)
        if dist is None or not self.load_prev_dists:
            num_skewed_nodes = math.ceil(0.2 * len(eps))
            skewed_node_probs = [0.55/num_skewed_nodes for _ in range(num_skewed_nodes)]
            if racks_dict is None:
                rack_prob_config = None
            else:
                rack_prob_config = {'racks_dict': racks_dict, 'prob_inter_rack': 0.7}
            dist = node_dists.gen_multimodal_node_dist(eps,
                rack_prob_config=rack_prob_
    ↪config,
                num_skewed_nodes=num_skewed_
    ↪nodes,
                skewed_node_probs=skewed_node_
    ↪probs,
                show_fig=False,
                print_data=False)

        super().save_dist(dist, dist_name)
        return dist

    def get_interarrival_time_dist(self, dist_name='interarrival_time_dist'):
        dist, path = super().get_interarrival_time_dist(dist_name)
        if dist is None or not self.load_prev_dists:
            dist = val_dists.gen_named_val_dist(dist='weibull',
```

(continues on next page)

(continued from previous page)

```

        params={'_alpha': 0.9, '_lambda': 6000},
        min_val=1,
        round_to_nearest=25,
        show_fig=False,
        print_data=False)

    super().save_dist(dist, dist_name)
    return dist

def get_flow_size_dist(self, dist_name='flow_size_dist'):
    dist, path = super().get_flow_size_dist(dist_name)
    if dist is None or not self.load_prev_dists:
        dist = val_dists.gen_named_val_dist(dist='lognormal',
            params={'_mu': 7, '_sigma': 2.5},
            min_val=1,
            max_val=2e7,
            round_to_nearest=25,
            show_fig=False,
            print_data=False)

    super().save_dist(dist, dist_name)
    return dist

```

Once you've saved this file, you should be able to use the `trafpy.benchmark.BenchmarkImporter` class to import your default benchmark just as you would any default TrafPy benchmark:

```

from trafpy.benchmark import BenchmarkImporter

importer = BenchmarkImporter(benchmark_version='v001', load_prev_dists=False)
net = tpg.gen_fat_tree(k=4)
dists = importer.get_benchmark_dists(benchmark_name='my_default_benchmark', eps=net.
    ↳graph['endpoints'], racks_dict=net.graph['rack_to_ep_dict'])

```

If you wish to share this default benchmark with the community, please feel free to make a contribution to the open-source TrafPy project. See the Contribute guide for details.

3.2.3 TrafPy Manager

Warning: The `trafpy.manager` package is still a working progress. The aim of it is to integrate easily with demand data generated by the `trafpy.generator` package to enable end-to-end network benchmarking, standardisation, learning-agent training etc. using only TrafPy. Most people will find the `trafpy.generator` module the only one they need in order to generate traffic and import it into their own simulations.

As this tutorial has shown, TrafPy can be used as a stand-alone tool for generating, replicating, and reproducing network traffic data using the `trafpy.generator` package and the interactive Jupyter Notebook tool. TrafPy also comes with another package, `trafpy.manager`, which uses generated network traffic data to simulate networks. `trafpy.manager` can be used as a tool for e.g. benchmarking and comparing different network managers (routers, schedulers, machine placers, etc.) and for e.g. a reinforcement learning training environment.

`trafpy.manager` works by initialising a network environment (e.g. a data centre network) which itself is initialised with a TrafPy demand object, a scheduling agent, a routing agent, and a network object. TrafPy comes with pre-built versions of each of these, but has been designed such that users can write their own e.g. scheduler and benchmark it with `trafpy.manager` and with network demands generated with `trafpy.generator`.

Import the `trafpy.generator` package and the required objects from the `trafpy.manager` package:

```
import trafpy.generator as tpg
from trafpy.manager import Demand, RWA, SRPT, DCN
from imports import config
```

Where the `config.py` file might be defined as

```
LOAD_DEMANDS = None
NUM_EPISODES = 1
NUM_K_PATHS = 1
NUM_CHANNELS = 1
NUM_DEMANDS = 10
MIN_FLOW_SIZE = 1
MAX_FLOW_SIZE = 100
MIN_NUM_OPS = 50
MAX_NUM_OPS = 200
C = 1.5
MIN_INTERARRIVAL = 1
MAX_INTERARRIVAL = 1e8
SLOT_SIZE = 10000
MAX_FLOWS = None
MAX_TIME = None
ENDPOINT_LABEL = 'server'
ENDPOINT_LABELS = [ENDPOINT_LABEL+'_'+str(ep) for ep in range(5)]
PATH_FIGURES = '../figures/'
PATH_PICKLES = '../pickles/demand/tf_graphs/real/'

print('Demand config file imported.')
if ENDPOINT_LABELS is None:
    print('Warning: ENDPOINTS left as None. Will need to provide own networkx \
graph with correct labelling. To avoid this, specify list of endpoint \
labels in config.py')
```

Load your previously saved TrafPy demand data dictionary (see the TrafPy Generator section above):

```
demand_data = tpg.unpickle_data(path_to_load='data/flow-centric-demand_data.pickle', zip_
↪data=True)
```

Initialise the `trafpy.manager` objects:

```
network = tpg.gen_simple_network(ep_label=config.ENDPOINT_LABEL,num_channels=config.NUM_
↪CHANNELS)
demand = Demand(demand_data=demand_data)
rwa = RWA(tpg.gen_channel_names(config.NUM_CHANNELS), config.NUM_K_PATHS)
scheduler = SRPT(network, rwa, slot_size=config.SLOT_SIZE)
env = DCN(network, demand, scheduler, slot_size=config.SLOT_SIZE, max_flows=config.MAX_
↪FLOWS, max_time=config.MAX_TIME)
```

And run your simulation using the standard OpenAI Gym reinforcement learning framework:

```
for episode in range(config.NUM_EPISODES):
    print('\nEpisode {}/{}'.format(episode+1,config.NUM_EPISODES))
```

(continues on next page)

(continued from previous page)

```

observation = env.reset(config.LOAD_DEMANDS)
while True:
    print('Time: {}'.format(env.curr_time))
    action = scheduler.get_action(observation)
    print('Action:\n{}'.format(action))
    observation, reward, done, info = env.step(action)
    if done:
        print('Episode finished.')
        break

```

When completed, you can print TrafPy’s summary of the scheduling session:

```

>>> env.get_scheduling_session_summary(print_summary=True)
----- Scheduling Session Ended -----
SUMMARY:
~* General Info *~
Total session duration: 80000.0 time units
Total number of generated demands (jobs or flows): 10
Total info arrived: 56623.0 info units
Load: 0.7672975615099775 info unit demands arrived per unit time (from first to last,
→flow arriving)
Total info transported: 56623.0 info units
Throughput: 0.7077875 info units transported per unit time

~* Flow Info *~
Total number generated flows (src!=dst,dependency_type=='data_dep'): 10
Time first flow arrived: 0.0 time units
Time last flow arrived: 73795.36028834846 time units
Time first flow completed: 10000.0 time units
Time last flow completed: 80000.0 time units
Total number of demands that arrived and became flows: 10
Total number of flows that were completed: 10
Total number of dropped flows + flows in queues at end of session: 0
Average FCT: 7669.998225473775 time units
99th percentile FCT: 18035.645744379803 time units

```

3.3 Contribute

This guide will help you contribute to e.g. fix a bug or add a new feature for TrafPy.

3.3.1 Development Workflow

1. If you are a first-time contributor:

- Go to <https://github.com/cwfpinson/trafpy> and click the “fork” button to create your own copy of the project.
- Clone the project to your local computer:

```
git clone git@github.com:your-username/trafpy.git
```

- Navigate to the folder trafpy and add the upstream repository:

```
git remote add upstream git@github.com:cwfpersonson/trafpy.git
```

- Now, you have remote repositories named:
 - upstream, which refers to the trafpy repository
 - origin, which refers to your personal fork
- Next, you need to set up your build environment. Here are instructions for two popular environment managers:
 - venv (pip based)

```
# Create a virtualenv named ``trafpy-dev`` that lives in the directory of
# the same name
python -m venv trafpy-dev
# Activate it
source trafpy-dev/bin/activate
# Install main development and runtime dependencies of trafpy
pip install -r <(cat requirements/{default,docs}.txt)
#
# These packages require that you have your system properly configured
# and what that involves differs on various systems.
#
# In the trafpy root directory folder, run
python setup.py develop
# Test your installation in a .py file
import trafpy.generator as tpg
from trafpy.manager import Demand, DCN, SRPT, RWA
```

- conda (Anaconda or Miniconda)

```
# Create a conda environment named ``trafpy-dev``
conda create --name trafpy-dev
# Activate it
conda activate trafpy-dev
# Install main development and runtime dependencies of trafpy
conda install -c conda-forge `for i in requirements/{default,doc}.txt; do
  echo -n " --file $i "; done`
#
# These packages require that you have your system properly configured
# and what that involves differs on various systems.
#
# In the trafpy root directory folder, run
python setup.py develop
# Test your installation in a .py file
import trafpy.generator as tpg
from trafpy.manager import Demand, DCN, SRPT, RWA
```

- Finally, it is recommended you use a pre-commit hook, which runs black when you type `git commit`:

```
pre-commit install
```

2. Develop your contribution:

- Pull the latest changes from upstream:

```
git checkout master
git pull upstream master
```

- Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as 'bugfix-for-issue-1480':

```
git checkout -b bugfix-for-issue-1480
```

- Commit locally as you progress (`git add` and `git commit`)

3. Submit your contribution:

- Push your changes back to your fork on GitHub:

```
git push origin bugfix-for-issue-1480
```

- Go to GitHub. The new branch will show up with a green Pull Request button—click it.
- If you want, email cwfparsonson@gmail.com to explain your changes or to ask for review.

4. Review process:

- Your pull request will be reviewed.
- To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.

Note: If the PR closes an issue, make sure that GitHub knows to automatically close the issue when the PR is merged. For example, if the PR closes issue number 1480, you could use the phrase “Fixes #1480” in the PR description or commit message.

5. Document changes

If your change introduces any API modifications, please update `doc/release/release_dev.rst`.

If your change introduces a deprecation, add a reminder to `doc/developer/deprecations.rst` for the team to remove the deprecated functionality in the future.

Note: To reviewers: make sure the merge message has a brief description of the change(s) and if the PR closes an issue add, for example, “Closes #123” where 123 is the issue number.

3.3.2 Divergence from upstream master

If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, merge the master branch into yours:

```
git fetch upstream master
git merge upstream/master
```

If any conflicts occur, they need to be fixed before continuing. See which files are in conflict using:

```
git status
```

Which displays a message like:

```
Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   file_with_conflict.txt
```

Inside the conflicted file, you'll find sections like these:

```
<<<<<< HEAD
The way the text looks in your branch
=====
The way the text looks in the master branch
>>>>>> master
```

Choose one version of the text that should be kept, and delete the rest:

```
The way the text looks in your branch
```

Now, add the fixed file:

```
git add file_with_conflict.txt
```

Once you've fixed all merge conflicts, do:

```
git commit
```

3.4 License

TrafPy is distributed with the Apache License 2.0.

3.5 Citing

3.6 setup module

3.7 trafpy package

3.7.1 Subpackages

trafpy.benchmark package

Subpackages

trafpy.benchmark.versions package

Subpackages

trafpy.benchmark.versions.benchmark_v001 package

Submodules

`trafpy.benchmark.versions.benchmark_v001.benchmark_plot_script` module

`trafpy.benchmark.versions.benchmark_v001.config` module

`trafpy.benchmark.versions.benchmark_v001.distribution_generator` module

`trafpy.benchmark.versions.benchmark_v001.old_distribution_generator` module

Module contents

Submodules

`trafpy.benchmark.versions.benchmark` module

```
class trafpy.benchmark.versions.benchmark.Benchmark(benchmark_name,
                                                    benchmark_version='v001',
                                                    load_prev_dists=True, jobcentric=False)
```

Bases: ABC

get_dist_and_path(*dist_name*)

Gets distribution data and corresponding path.

If distribution data does not exist, will return `dist=None` and the path will be where the dist data should be saved if it is generated.

abstract get_flow_size_dist(*dist_name*='flow_size_dist')

Loads previously saved flow size dist (if it exists).

This is an abstract method and therefore must be defined by any child class.

Parameters

dist_name (*str*) – Name of distribution (determines path to search for previously saved distribution).

abstract get_interarrival_time_dist(*dist_name*='interarrival_time_dist')

Loads previously saved interarrival time dist (if it exists).

This is an abstract method and therefore must be defined by any child class.

Parameters

dist_name (*str*) – Name of distribution (determines path to search for previously saved distribution).

abstract get_node_dist(*eps*, *racks_dict*, *dist_name*='node_dist')

Loads previously saved node dist (if it exists).

This is an abstract method and therefore must be defined by any child class.

Parameters

- **eps** (*list*) – List of network end points.
- **racks_dict** (*dict*) – Dict mapping racks to the corresponding end points contained within each rack.

- **dist_name** (*str*) – Name of distribution (determines path to search for previously saved distribution).

get_num_ops_dist(*dist_name='num_ops_dist'*)

Loads previously saved number of operations dist (if it exists).

This method only needs to be defined by a child class if jobcentric=True, since flow-centric data have no notion of ‘number of operations’ (in relation to job DAGs).

Parameters

- **dist_name** (*str*) – Name of distribution (determines path to search for previously saved distribution).

load_dist(*benchmark_name, dist_name*)

Loads previously saved distribution data for given benchmark.

save_dist(*dist_data, dist_name*)

Saves distribution data for a given benchmark.

trafpy.benchmarker.versions.benchmark_importer module

class trafpy.benchmarker.versions.benchmark_importer.**BenchmarkImporter**(*benchmark_version='v001', load_prev_dists=True*)

Bases: object

get_benchmark_dists(*benchmark_name, eps, racks_dict=None*)

Retrieves pre-defined TrafPy benchmark distributions for a network.

Parameters

- **benchmark_name** (*str*) – Name of benchmark (e.g. ‘university’).
- **eps** (*list*) – List of end points/machines/leaf nodes in network.
- **racks_dict** (*dict*) – Mapping of which end points are in which racks. Keys are rack ids, values are list of end points. If None, assume there is not clustering/rack system in the network where have different end points in different clusters/racks.

trafpy.benchmarker.versions.old_benchmark_importer module

class trafpy.benchmarker.versions.old_benchmark_importer.**BenchmarkImporter**(*benchmark_version, load_prev_dists=True*)

Bases: object

get_benchmark_dists(*benchmark, eps, racks_dict=None*)

Retrieves pre-defined TrafPy benchmark distributions for a network.

Parameters

- **benchmark** (*str*) – Name of benchmark (e.g. ‘university’).
- **eps** (*list*) – List of end points/machines/leaf nodes in network.
- **racks_dict** (*dict*) – Mapping of which end points are in which racks. Keys are rack ids, values are list of end points. If None, assume there is not clustering/rack system in the network where have different end points in different clusters/racks.

Module contents

Submodules

`trafpy.benchmarker` module

`trafpy.benchmarker` module

`trafpy.benchmarker` module

`trafpy.benchmarker.config` module

`trafpy.benchmarker.gen_new_slots_dict` module

`trafpy.benchmarker.main_gen_benchmark_data` module

`trafpy.benchmarker.main_testbed_benchmark_data` module

`trafpy.benchmarker.speed_test` module

`trafpy.benchmarker.test` module

`trafpy.benchmarker.test.do_work(x)`

`trafpy.benchmarker.test.update_pbar(_pbar)`

`trafpy.benchmarker.tools` module

Module contents

`trafpy.generator` package

Subpackages

`trafpy.generator.src` package

Subpackages

`trafpy.generator.src.dists` package

Submodules

`trafpy.generator.src.dists.node_dists` module

Module for generating node distributions.

```
trafpy.generator.src.dists.node_dists.adjust_node_dist_for_rack_prob_config(rack_prob_config,
                                                                            eps, node_dist,
                                                                            print_data=False)
```

Unlike the other `adjust_node_dist_from_multinomial_exp_for_rack_prob_config` function, this function does not use a multinomial experiment to adjust the prob dist, but rather uses a deterministic method of distorting the probabilities from the original node distribution such that the required inter-/intra-rack probabilities are met.

```
trafpy.generator.src.dists.node_dists.adjust_node_dist_from_multinomial_exp_for_rack_prob_config(rack_prob_config,
                                                                                               eps,
                                                                                               node_dist,
                                                                                               num_exp,
                                                                                               print_data=False)
```

Unlike the other `adjust_node_dist_for_rack_prob_config` function, this function adjusts the node dist by running multinomial experiments on the initial node distribution to sample from it. It therefore takes much much longer than the other function, especially for networks with >1,000 nodes.

Takes node dist and uses it to generate new node dist given inter- and intra-rack configuration.

Different DCNs have different inter and intra rack traffic. This function allows you to specify how much of your traffic should be inter and intra rack.

Parameters

- **rack_prob_config** (*dict*) – Network endpoints/servers are often grouped into physically local clusters or ‘racks’. Different networks may have different levels of inter- (between) and intra- (within) rack communication. If `rack_prob_config` is left as `None`, will assume that server-server srs-dst requests are independent of which rack they might be in. If specified, dict should have a ‘racks_dict’ key, whose value is a dict with keys as rack labels (e.g. ‘rack_0’, ‘rack_1’ etc.) and whose value for each key is a list of the endpoints in the respective rack (e.g. [‘server_0’, ‘server_24’, ‘server_56’, ...]), and a ‘prob_inter_rack’ key whose value is a float (e.g. 0.9), setting the probability that a chosen src endpoint has a destination which is outside of its rack. If you want to e.g. designate an entire rack as a ‘hot rack’ (many traffic requests occur from this rack), would specify `skewed_nodes` to contain the list of servers in this rack and configure `rack_prob_config` appropriately.
- **eps** (*list*) – List of network node endpoints that can act as sources & destinations.
- **node_dist** (*numpy array*) – 2D matrix array of source-destination pair probabilities of being chosen.
- **num_exps_factor** (*int*) – Factor by which to multiply number of ep pairs to get the number of multinomial experiments to run when generating new node dist.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

```
trafpy.generator.src.dists.node_dists.adjust_probability_array_sum(probs, target_sum=1,
                                                                    print_data=False)
```

For array.

```
trafpy.generator.src.dists.node_dists.adjust_probability_dict_sum(probs, target_sum=1,
                                                                    print_data=False)
```

For dict.

```
trafpy.generator.src.dists.node_dists.assign_matrix_to_probs(eps, node_dist)
```

Assigns probabilities in 2D matrix to a src-dst pair prob dist dict.

```
trafpy.generator.src.dists.node_dists.assign_probs_to_matrix(eps, probs, matrix=None)
```

Assigns probabilities to 2D matrix.

probs can be list of pair probabilities or dict of key-value pair-probability

N.B. if probs is list, assumes probs are given in order of matrix indices when looping for src in eps for dst in eps

trafpy.generator.src.dists.node_dists.**convert_pair_prob_dist_dict_to_matrix_pair_prob_dist_dict**(*pair_probs*, *eps*)

Parameters

pair_prob_dist (*dict*) – Dict whose keys are node pairs and whose values are probabilities or fractions.

Returns

Dict whose keys are matrix indices of the node

pairs and whose values are the pairs' corresponding probabilities or fractions.

Return type

matrix_pair_prob_dist (*dict*)

trafpy.generator.src.dists.node_dists.**convert_sampled_pairs_into_node_dist**(*sampled_pairs*, *eps*)

trafpy.generator.src.dists.node_dists.**gen_demand_nodes**(*eps*, *node_dist*, *size*, *axis*, *path_to_save=None*, *check_sum_valid=True*)

Generates demand nodes following the node_dist distribution

Parameters

- **eps** (*list*) – List of node endpoint labels.
- **node_dist** (*numpy array*) – Probability distribution each node is chosen
- **size** (*int*) – Number of demand nodes to generate
- **axis** (*int, 1 or 0*) – Which axis of normalised node distribution to consider. E.g. If generating src nodes, axis=0. If dst nodes, axis=1
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. path_to_save='data/dists/my_dist'.
- **check_sum_valid** (*bool*) – Whether or not to ensure node dist sums to 1. If need efficiency, should set to False.

trafpy.generator.src.dists.node_dists.**gen_multimodal_node_dist**(*eps*, *skewed_nodes=[]*, *skewed_node_probs=[]*, *num_skewed_nodes=None*, *rack_prob_config=None*, *path_to_save=None*, *plot_fig=False*, *show_fig=False*, *print_data=False*)

Generates a multimodal node distribution.

Generates a multimodal node demand distribution i.e. certain nodes have a certain specified probability of being chosen. If no skewed nodes given, randomly selects random no. node(s) to skew. If no skew node probabilities given, random selects probability with which to skew the node between 0.5 and 0.8. If no num skewed nodes given, randomly chooses number of nodes to skew.

Parameters

- **eps** (*list*) – List of network node endpoints that can act as sources & destinations

- **skewed_nodes** (*list*) – Node(s) to whose probability of being chosen you want to skew/specify
- **skewed_node_probs** (*list*) – Probabilit(y)ies of node(s) being chosen/specified skews
- **num_skewed_nodes** (*int*) – Number of nodes to skew. If None, will gen a number between 10% and 30% of the total number of nodes in network
- **rack_prob_config** (*dict*) – Network endpoints/servers are often grouped into physically local clusters or ‘racks’. Different networks may have different levels of inter- (between) and intra- (within) rack communication. If rack_prob_config is left as None, will assume that server-server srs-dst requests are independent of which rack they might be in. If specified, dict should have a ‘racks_dict’ key, whose value is a dict with keys as rack labels (e.g. ‘rack_0’, ‘rack_1’ etc.) and whose value for each key is a list of the endpoints in the respective rack (e.g. [‘server_0’, ‘server_24’, ‘server_56’, ...]), and a ‘prob_inter_rack’ key whose value is a float (e.g. 0.9), setting the probability that a chosen src endpoint has a destination which is outside of its rack. If you want to e.g. designate an entire rack as a ‘hot rack’ (many traffic requests occur from this rack), would specify skewed_nodes to contain the list of servers in this rack and configure rack_prob_config appropriately.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. path_to_save=‘data/dists/my_dist’.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will return and display fig.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

Returns

Tuple containing:

- **node_dist** (*numpy array*): 2D matrix array of souce-destination pair probabilities of being chosen.
- **fig** (*matplotlib.figure.Figure, optional*): Node distributions plotted as a 2D matrix. To return, set show_fig=True and/or plot_fig=True.

Return type

tuple

```
trafpy.generator.src.dists.node_dists.gen_multimodal_node_pair_dist(eps, skewed_pairs=[],
                                                                    skewed_pair_probs=[],
                                                                    num_skewed_pairs=None,
                                                                    rack_prob_config=None,
                                                                    path_to_save=None,
                                                                    plot_fig=False,
                                                                    show_fig=False,
                                                                    print_data=False)
```

Generates a multimodal node pair distribution.

Generates a multimodal node pair demand distribution i.e. certain node pairs have a certain specified probability of being chosen. If no skewed pairs given, randomly selects pair to skew. If no skew pair probabilities given, random selects probability with which to skew the pair between 0.1 and 0.3. If no num skewed pairs given, randomly chooses number of pairs to skew.

Parameters

- **eps** (*list*) – List of network node endpoints that can act as sources & destinations.
- **skewed_pairs** (*list of lists*) – List of the node pairs [src,dst] to skew.

- **skewed_pair_probs** (*list*) – Probabilities of node pairs being chosen.
- **num_skewed_pairs** (*int*) – Number of pairs to randomly skew.
- **rack_prob_config** (*dict*) – Network endpoints/servers are often grouped into physically local clusters or ‘racks’. Different networks may have different levels of inter- (between) and intra- (within) rack communication. If `rack_prob_config` is left as `None`, will assume that server-server src-dst requests are independent of which rack they might be in. If specified, dict should have a ‘racks_dict’ key, whose value is a dict with keys as rack labels (e.g. ‘rack_0’, ‘rack_1’ etc.) and whose value for each key is a list of the endpoints in the respective rack (e.g. [‘server_0’, ‘server_24’, ‘server_56’, ...]), and a ‘prob_inter_rack’ key whose value is a float (e.g. 0.9), setting the probability that a chosen src endpoint has a destination which is outside of its rack. If you want to e.g. designate an entire rack as a ‘hot rack’ (many traffic requests occur from this rack), would specify `skewed_nodes` to contain the list of servers in this rack and configure `rack_prob_config` appropriately.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. `path_to_save='data/dists/my_dist'`.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will return and display fig.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

Returns

Tuple containing:

- **node_dist** (*numpy array*): 2D matrix array of source-destination pair probabilities of being chosen.
- **fig** (*matplotlib.figure.Figure, optional*): Node distributions plotted as a 2D matrix. To return, set `show_fig=True` and/or `plot_fig=True`.

Return type

tuple

`trafpy.generator.src.dists.node_dists.gen_node_demands(eps, node_dist, num_demands, rack_prob_config=None, duplicate=False, path_to_save=None)`

Uses node distribution to generate src-dst node pair demands.

Parameters

- **eps** (*list*) – List of network node endpoints that can act as sources & destinations.
- **node_dist** (*numpy array*) – 2D matrix array of source-destination pair probabilities of being chosen.
- **num_demands** (*int*) – Number of src-dst node pairs to generate.
- **duplicate** (*bool*) – Whether or not to duplicate src-dst node pairs. Use this if demands you’re generating have a ‘take down’ event as well as an ‘establish’ event.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. `path_to_save='data/dists/my_dist'`.

Returns

Tuple containing:

- **sn** (*numpy array*): Selected source nodes.
- **dn** (*numpy array*): Selected destination nodes.

Return type

tuple

```
trafpy.generator.src.dists.node_dists.gen_uniform_multinomial_exp_node_dist(eps,
                                                                              rack_prob_config=None,
                                                                              path_to_save=None,
                                                                              plot_fig=False,
                                                                              show_fig=False,
                                                                              print_data=False)
```

Runs multinomial exp with uniform initial probability to generate slight skew.

Runs a multinomial experiment where each node pair has same (uniform) probability of being chosen. Will generate a node demand distribution where a few pairs & nodes have a slight skew in demand

Parameters

- **eps** (*list*) – List of network node endpoints that can act as sources & destinations
- **rack_prob_config** (*dict*) – Network endpoints/servers are often grouped into physically local clusters or ‘racks’. Different networks may have different levels of inter- (between) and intra- (within) rack communication. If *rack_prob_config* is left as *None*, will assume that server-server srs-dst requests are independent of which rack they might be in. If specified, dict should have a ‘racks_dict’ key, whose value is a dict with keys as rack labels (e.g. ‘rack_0’, ‘rack_1’ etc.) and whose value for each key is a list of the endpoints in the respective rack (e.g. [‘server_0’, ‘server_24’, ‘server_56’, ...]), and a ‘prob_inter_rack’ key whose value is a float (e.g. 0.9), setting the probability that a chosen src endpoint has a destination which is outside of its rack. If you want to e.g. designate an entire rack as a ‘hot rack’ (many traffic requests occur from this rack), would specify *skewed_nodes* to contain the list of servers in this rack and configure *rack_prob_config* appropriately.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. *path_to_save*=‘data/dists/my_dist’.
- **plot_fig** (*bool*) – Whether or not to plot fig. If *True*, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If *True*, will return and display fig.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

Returns**Tuple containing:**

- **node_dist** (*numpy array*): 2D matrix array of source-destination pair probabilities of being chosen.
- **fig** (*matplotlib.figure.figure, optional*): node distribution plotted as a 2d matrix. to return, set *show_fig*=*true* and/or *plot_fig*=*true*.

Return type

tuple

```
trafpy.generator.src.dists.node_dists.gen_uniform_node_dist(eps, rack_prob_config=None,
                                                            path_to_save=None, plot_fig=False,
                                                            show_fig=False, print_data=False)
```

Generates a uniform node distribution.

Parameters

- **eps** (*list*) – List of network node endpoints that can act as sources & destinations

- **rack_prob_config** (*dict*) – Network endpoints/servers are often grouped into physically local clusters or ‘racks’. Different networks may have different levels of inter- (between) and intra- (within) rack communication. If rack_prob_config is left as None, will assume that server-server srs-dst requests are independent of which rack they might be in. If specified, dict should have a ‘racks_dict’ key, whose value is a dict with keys as rack labels (e.g. ‘rack_0’, ‘rack_1’ etc.) and whose value for each key is a list of the endpoints in the respective rack (e.g. [‘server_0’, ‘server_24’, ‘server_56’, ...]), and a ‘prob_inter_rack’ key whose value is a float (e.g. 0.9), setting the probability that a chosen src endpoint has a destination which is outside of its rack. If you want to e.g. designate an entire rack as a ‘hot rack’ (many traffic requests occur from this rack), would specify skewed_nodes to contain the list of servers in this rack and configure rack_prob_config appropriately.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. path_to_save=‘data/dists/my_dist’.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will return and display fig.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

Returns

Tuple containing:

- **node_dist** (*numpy array*): 2D matrix array of source-destination pair probabilities of being chosen.
- **fig** (*matplotlib.figure.Figure, optional*): Node distributions plotted as a 2D matrix. To return, set show_fig=True and/or plot_fig=True.

Return type

tuple

trafpy.generator.src.dists.node_dists.**get_inter_intra_rack_pair_prob_dicts**(*pair_prob_dict*,
ep_to_rack_dict)

trafpy.generator.src.dists.node_dists.**get_network_pair_mapper**(*eps*)

Gets dicts mapping network endpoint indices to and from node dist matrix.

trafpy.generator.src.dists.node_dists.**get_pair_prob_dict_of_node_dist_matrix**(*node_dist*, *eps*,
all_combinations=False,
bidirectional=False)

Gets prob dict of each pair being chosen given node dist of probabilities.

If all_combinations, will record pair probabilities for all possible pair combinations i.e. src-dst and dst-src. If False, assumes src-dst==dst-src.

If bidirectional, will multiply probabilities by 2 as pair can be src-dst or dst-src. If bidirectional=True -> values sum to 1, if bidirectional=False -> values sum to 0.5.

trafpy.generator.src.dists.node_dists.**get_suitable_destination_node_for_rack_config**(*sn*,
node_dist,
eps,
ep_to_rack,
rack_to_ep,
inter_rack)

Given source node, finds destination node given inter and intra rack config.

trafpy.generator.src.dists.plot_dists module**trafpy.generator.src.dists.val_dists module**

Module for generating value distributions.

```
trafpy.generator.src.dists.val_dists.combine_multiple_mode_dists(data_dict, min_val, max_val,
                                                                xlim=None,
                                                                rand_var_name='Unknown',
                                                                round_to_nearest=None,
                                                                num_decimal_places=2)
```

```
trafpy.generator.src.dists.val_dists.combine_skews(data_dict, min_val, max_val, bg_factor=0.5,
                                                    xlim=None, logscale=False, transparent=True,
                                                    rand_var_name='Unknown', num_bins=0,
                                                    round_to_nearest=None,
                                                    num_decimal_places=2)
```

Combines multiple probability distributions for multimodal plotting.

Parameters

- **data_dict** (*dict*) – Keys are mode iterations, values are random variable values for the mode iteration.
- **min_val** (*int/float*) – Minimum random variable value.
- **max_val** (*int/float*) – Maximum random variable value.
- **bg_factor** (*int/float*) – Factor used to determine amount of noise to add amongst shaped modes being combined. Higher factor will add more noise to distribution and make modes more connected, lower will reduce noise but make nodes less connected.
- **xlim** (*list*) – X-axis limits of plot. E.g. xlim=[0,10] to plot random variable values between 0 and 10.
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **rand_var_name** (*str*) – Name of random variable to label plot's x-axis.
- **num_bins** (*int*) – Number of bins to use in plot. Default is 0, in which case the number of bins chosen will be automatically selected.
- **round_to_nearest** (*int/float*) – Value to round random variables to nearest. E.g. if round_to_nearest=0.2, will round each random variable to nearest 0.2.
- **num_decimal_places** (*int*) – Number of decimal places to random variable values. Need to explicitly state otherwise Python's floating point arithmetic will cause spurious unique random variable value errors when discretising.

Returns

Probability distribution whose key-value pairs are random variable value-probability pairs.

Return type

dict

```
trafpy.generator.src.dists.val_dists.convert_data_to_key_occurrences(data)
```

Converts random variable data into value keys and corresponding occurrences.

Parameters

- **data** (*list*) – Random variables to convert into key-num_occurrences pairs.

Returns

Random variable value - number of occurrences key-value pairs generated from random variable data.

Return type

dict

`trafpy.generator.src.dists.val_dists.convert_key_occurrences_to_data(keys, num_occurrences)`

Converts value keys and their number of occurrences into random vars.

Parameters

- **keys** (*list*) – Random variable values.
- **num_occurrences** (*list*) – Number of each random variable to generate.

Returns

Random variables generated.

Return type

list

`trafpy.generator.src.dists.val_dists.gen_discrete_prob_dist(rand_vars, unique_vars=None, round_to_nearest=None, num_decimal_places=2, path_to_save=None)`

Generate discrete probability distribution from list of random variables.

Takes rand var values, rounds to nearest value (specified as arg, defaults by not rounding at all) to discretise the data, and generates a probability distribution for the data

Parameters

- **rand_vars** (*list*) – Random variable values
- **unique_vars** (*list*) – List of unique random variables that can occur. If given, will init each as having occurred 0 times and count number of times each occurred. If left as None, will only record probabilities of random variables that actually occurred in rand_vars.
- **round_to_nearest** (*int/float*) – Value to round rand vars to nearest when discretising rand var values. E.g. is round_to_nearest=0.2, will round each rand var to nearest 0.2
- **num_decimal_places** (*int*) – Number of decimal places for discretised rand vars. Need to explicitly state this because otherwise Python's floating point arithmetic will cause spurious unique random var values

Returns

Tuple containing:

- **xk** (*list*): List of (discretised) unique random variable values that occurred
- **pmf** (*list*): List of corresponding probabilities that each unique value in xk occurs

Return type

tuple

`trafpy.generator.src.dists.val_dists.gen_exponential_dist(C_beta, size, round_to_nearest=None, num_decimal_places=2, min_val=None, max_val=None, interactive_params=None, logscale=False, transparent=True)`

Generates an exponential distribution of random variable values.

The exponential distribution often fits scenarios whose events' random variable values (e.g. 'time between events') are made of many small values (e.g. time intervals) and a few large values. Often used to predict time until next event occurs.

E.g. Real-world scenarios: Time between earthquakes, car accidents, mail delivery, and data centre demand arrival.

Parameters

- **_beta** (*int/float*) – Mean random variable value.
- **size** (*int*) – Number of random variable values to sample from distribution.
- **interactive_params** (*dict*) – Dictionary of distribution parameter values (must provide if in interactive mode).
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.

Returns

Random variable values generated by sampling from the distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.gen_lognormal_dist(_mu, _sigma, size,
                                                         round_to_nearest=None,
                                                         num_decimal_places=2, min_val=None,
                                                         max_val=None, interactive_params=None,
                                                         logscale=False, transparent=True)
```

Generates a log-normal distribution of random variable values.

Log-normal distributions often fit scenarios whose random variable values have a low mean value but a high degree of variance, leading to a distribution that is positively skewed (i.e. has a long tail to the right of its peak).

The log-normal distribution is mathematically similar to the normal distribution, since its random variable is normally distributed when its logarithm is taken. I.e. for a log-normally distributed random variable X , $Y=\ln(X)$ would have a normal distribution.

E.g. of real-world scenarios: Length of a chess game, number of hospitalisations during an epidemic, the time after which a mechanical system needs repair, data centre demand interarrival times, etc.

Parameters

- **_mu** (*int/float*) – Mean value of underlying normal distribution.
- **_sigma** (*int/float*) – Standard deviation of underlying normal distribution.
- **size** (*int*) – Number of random variable values to sample from distribution.
- **interactive_params** (*dict*) – Dictionary of distribution parameter values (must provide if in interactive mode).
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.

Returns

Random variable values generated by sampling from the distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.gen_multimodal_val_dist(min_val, max_val, locations=[],
                                                             skews=[], scales=[],
                                                             num_skew_samples=[],
                                                             bg_factor=0.5,
                                                             round_to_nearest=None,
                                                             num_decimal_places=2,
                                                             occurrence_multiplier=10,
                                                             path_to_save=None, plot_fig=False,
                                                             show_fig=False, return_data=False,
                                                             xlim=None, logscale=False,
                                                             rand_var_name='Random Variable',
                                                             prob_rand_var_less_than=None,
                                                             num_bins=0, print_data=False)
```

Generates a multimodal distribution of random variable values.

Multimodal distributions are arbitrary distributions with ≥ 2 different modes. A multimodal distribution with 2 modes is a special case called a 'bimodal distribution'. Bimodal distributions are the most common multi-modal distribution.

E.g. Real-world scenarios of bimodal distributions: Starting salaries for lawyers, book prices, peak resaurant hours, age groups of disease victims, packet sizes in data centre networks, etc.

Parameters

- **min_val** (*int/float*) – Minimum random variable value.
- **max_val** (*int/float*) – Maximum random variable value.
- **locations** (*list*) – Position value(s) of skewed distribution(s) (mean shape parameter).
- **skews** (*list*) – Skew value(s) of skewed distribution(s) (skewness shape parameter).
- **scales** (*list*) – Scale value(s) of skewed distribution(s) (standard deviation shape parameter).
- **num_skew_samples** (*list*) – Number(s) of random variables to sample from distribution(s) to generate skew data and plot.
- **bg_factor** (*int/float*) – Factor used to determine amount of noise to add amongst shaped modes being combined. Higher factor will add more noise to distribution and make modes more connected, lower will reduce noise but make nodes less connected.
- **round_to_nearest** (*int/float*) – Value to round random variables to nearest. E.g. if round_to_nearest=0.2, will round each random variable to nearest 0.2.
- **num_decimal_places** (*int*) – Number of decimal places to random variable values. Need to explicitly state otherwise Python's floating point arithmetic will cause spurious unique random variable value errors when discretising.
- **occurrence_multiplier** (*int/float*) – When sampling random variables from distribution to create plot and random variable data, use this multiplier to determine number of data points to sample. A higher value will cause the random variable data to match the probability distribution more closely, but will take longer to generate.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. path_to_save='data/dists/my_dist'.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.

- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will
- **tuple** – return and display fig.
- **return_data** (*bool*) – from generated distribution.
- **xlim** (*list*) – X-axis limits of plot. E.g. xlim=[0,10] to plot random variable values between 0 and 10.
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **rand_var_name** (*str*) – Name of random variable to label plot's x-axis.
- **prob_rand_var_less_than** (*list*) – List of values for which to print the probability that a variable sampled randomly from the generated distribution will be less than. This is useful for replicating distributions from the literature. E.g. prob_rand_var_less_than=[3.7,5.8] will return the probability that a randomly chosen variable is less than 3.7 and 5.8 respectively.
- **num_bins** (*int*) – Number of bins to use in plot. Default is 0, in which case the number of bins chosen will be automatically selected.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

Returns

Tuple containing:

- **prob_dist** (*dict*): Probability distribution whose key-value pairs are random variable value-probability pairs.
- **rand_vars** (*list, optional*): Random variable values sampled from the generated probability distribution. To return, set return_data=True.
- **fig** (*matplotlib.figure.Figure, optional*): Probability density and cumulative distribution function plot. To return, set show_fig=True and/or plot_fig=True.

Return type

tuple

```
trafpy.generator.src.dists.val_dists.gen_named_val_dist(dist, params=None, interactive_plot=False,
size=150000, occurrence_multiplier=100,
return_data=False,
round_to_nearest=None,
num_decimal_places=2,
path_to_save=None, plot_fig=False,
show_fig=False, min_val=None,
max_val=None, xlim=None,
logscale=False, rand_var_name='Random
Variable', prob_rand_var_less_than=None,
num_bins=0, print_data=False)
```

Generates a 'named' (e.g. Weibull/exponential/log-normal/Pareto) distribution.

Parameters

- **dist** (*str*) – One of the valid named distributions (e.g. 'weibull', 'lognormal', 'pareto', 'exponential')
- **params** (*dict*) – Corresponding parameter arguments of distribution (e.g. for Weibull, params={'_alpha': 1.4, '_lambda': 7000}). See individual name distribution function generators for more information.
- **interactive_plot** (*bool*) – Whether or not you want to use the interactive functionality of this function in Jupyter notebook to visually shape your named distribution.

- **size** (*int*) – Number of values to sample from generated distribution when generating random variable data.
- **round_to_nearest** (*int/float*) – Value to round random variables to nearest. E.g. if `round_to_nearest=0.2`, will round each random variable to nearest 0.2.
- **num_decimal_places** (*int*) – Number of decimal places to random variable values. Need to explicitly state otherwise Python's floating point arithmetic will cause spurious unique random variable value errors when discretising.
- **occurrence_multiplier** (*int/float*) – When sampling random variables from distribution to create plot and random variable data, use this multiplier to determine number of data points to sample. A higher value will cause the random variable data to match the probability distribution more closely, but will take longer to generate.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. `path_to_save='data/dists/my_dist'`.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will return and display fig.
- **return_data** (*bool*) – from generated distribution.
- **min_val** (*int/float*) – Minimum random variable value.
- **max_val** (*int/float*) – Maximum random variable value.
- **xlim** (*list*) – X-axis limits of plot. E.g. `xlim=[0,10]` to plot random variable values between 0 and 10.
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **rand_var_name** (*str*) – Name of random variable to label plot's x-axis.
- **prob_rand_var_less_than** (*list*) – List of values for which to print the probability that a variable sampled randomly from the generated distribution will be less than. This is useful for replicating distributions from the literature. E.g. `prob_rand_var_less_than=[3.7,5.8]` will return the probability that a randomly chosen variable is less than 3.7 and 5.8 respectively.
- **num_bins** (*int*) – Number of bins to use in plot. Default is 0, in which case the number of bins chosen will be automatically selected.
- **print_data** (*bool*) – Whether or not to print extra information about the generated data.

Returns

Tuple containing:

- **prob_dist** (*dict*): Probability distribution whose key-value pairs are random variable value-probability pairs.
- **rand_vars** (*list, optional*): Random variable values sampled from the generated probability distribution. To return, set `return_data=True`.
- **fig** (*matplotlib.figure.Figure, optional*): Probability density and cumulative distribution function plot. To return, set `show_fig=True` and/or `plot_fig=True`.

Return type

tuple

```
trafpy.generator.src.dists.val_dists.gen_normal_dist(loc, scale, size, round_to_nearest=None,
                                                    num_decimal_places=2, min_val=None,
                                                    max_val=None, interactive_params=None,
                                                    logscale=False, transparent=True)
```

Generates a normal/gaussian distribution of random variable values.

Parameters

- **size** (*int*) – number of random variable values to sample from distribution.
- **interactive_params** (*dict*) – Dictionary of distribution parameter values (must provide if in interactive mode).
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.

Returns

random variable values generated by sampling from the distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.gen_pareto_dist(_alpha, _mode, size, round_to_nearest=None,
                                                    num_decimal_places=2, min_val=None,
                                                    max_val=None, interactive_params=None,
                                                    logscale=False, transparent=True)
```

Generates a pareto distribution of random variable values.

Pareto distributions often fit scenarios whose random variable values have high probability of having a small range of values, leading to a distribution that is heavily skewed (i.e. has a long tail).

E.g. real-world scenarios: A large portion of society's wealth being held by a small portion of its population, human settlement sizes, value of oil reserves in oil fields, size of sand particles, male dating success on Tinder, sizes of data centre demands, etc.

Parameters

- **_alpha** (*int/float*) – Shape parameter of Pareto distribution. Describes how 'stretched out' (i.e. how high variance) the distribution is.
- **_mode** (*int/float*) – Mode of the distribution, which is also the distribution's minimum possible value.
- **size** (*int*) – number of random variable values to sample from distribution.
- **interactive_params** (*dict*) – Dictionary of distribution parameter values (must provide if in interactive mode).
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.

Returns

random variable values generated by sampling from the distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.gen_rand_vars_from_discretised_dist(unique_vars,
                                                                           probabilities,
                                                                           num_demands,
                                                                           jensen_shannon_distance_threshold=None,
                                                                           show_fig=False,
                                                                           xlabel='Random
                                                                           Variable',
                                                                           font_size=20,
                                                                           figsize=(4, 3),
                                                                           marker_size=15,
                                                                           logscale=False,
                                                                           path_to_save=None)
```

Generates random variable values by sampling from a discretised distribution.

Parameters

- **unique_vars** (*list*) – Possible random variable values.
- **probabilities** (*list*) – Corresponding probabilities of each random variable value being chosen.
- **num_demands** (*int*) – Number of random variables to sample.
- **jensen_shannon_distance_threshold** (*float*) – Maximum jensen shannon distance required of generated random variables w.r.t. discretised dist they're generated from. Must be between 0 and 1. Distance of 0 -> distributions are exactly the same. Distance of 1 -> distributions are not at all similar. <https://medium.com/datalab-log/measuring-the-statistical-similarity-between-two-samples-using-jensen-shannon-and-kullback-leibler-8d05af514b>
N.B. To meet threshold, this function will keep doubling num_demands
- **show_fig** (*bool*) – Whether or not to generated sampled var dist plotted with the original distribution.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated data. E.g. path_to_save='data/my_data'

Returns

Random variable values sampled from dist.

Return type

numpy array

```
trafpy.generator.src.dists.val_dists.gen_skew_data(location, skew, scale, min_val, max_val,
                                                    num_skew_samples, xlim=None, logscale=False,
                                                    transparent=True, rand_var_name='Unknown',
                                                    num_bins=0, round_to_nearest=None,
                                                    num_decimal_places=2)
```

Generates and plots skewed data for interactive multimodal distributions.

Parameters

- **location** (*int/float*) – Position value of skewed distribution (mean shape parameter).
- **skew** (*int/float*) – Skew value of skewed distribution (skewness shape parameter).
- **scale** (*int/float*) – Scale value of skewed distribution (standard deviation
- **scale** – Scale value of skewed distribution (standard deviation shape parameter). shape parameter).

- **num_skew_samples** (*int*) – Number of random variables to sample from distribution to generate skew data and plot.
- **xlim** (*list*) – X-axis limits of plot. E.g. `xlim=[0,10]` to plot random variable values between 0 and 10.
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.
- **rand_var_name** (*str*) – Name of random variable to label plot's x-axis.
- **num_bins** (*int*) – Number of bins to use in plot. Default is 0, in which case the number of bins chosen will be automatically selected.
- **round_to_nearest** (*int/float*) – Value to round random variables to nearest. E.g. if `round_to_nearest=0.2`, will round each random variable to nearest 0.2.
- **num_decimal_places** (*int*) – Number of decimal places to random variable values. Need to explicitly state otherwise Python's floating point arithmetic will cause spurious unique random variable value errors when discretising.

Returns

Random variable values sampled from distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.gen_skew_dists(min_val, max_val, num_modes=2, xlim=None,
                                                    rand_var_name='Unknown',
                                                    round_to_nearest=None,
                                                    num_decimal_places=2)
```

```
trafpy.generator.src.dists.val_dists.gen_skewnorm_data(a, loc, scale, num_samples, min_val=None,
                                                       max_val=None, round_to_nearest=None,
                                                       num_decimal_places=2,
                                                       interactive_params=None, logscale=False,
                                                       transparent=True)
```

Generates skew data.

Parameters

- **a** (*int/float*) – Skewness shape parameter. When `a=0`, distribution is identical to a normal distribution.
- **loc** (*int/float*) – Position value of skewed distribution (mean shape parameter).
- **scale** (*int/float*) – Scale value of skewed distribution (standard deviation shape parameter).
- **min_val** (*int/float*) – Minimum random variable value.
- **max_val** (*int/float*) – Maximum random variable value.
- **num_samples** (*int*) – Number of values to sample from generated distribution to generate skew data.

Returns

List of random variable values sampled from skewed distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.gen_skewnorm_val_dist(location, skew, scale,
                                                         num_skew_samples=150000,
                                                         min_val=None, max_val=None,
                                                         return_data=False, xlim=None,
                                                         plot_fig=False, show_fig=False,
                                                         logscale=False, path_to_save=None,
                                                         transparent=True,
                                                         rand_var_name='Random Variable',
                                                         num_bins=0, round_to_nearest=None,
                                                         occurrence_multiplier=10,
                                                         prob_rand_var_less_than=None,
                                                         num_decimal_places=2)
```

Generates a skew norm distribution of random variable values.

Parameters

- **location** (*int/float*) – Position value of skewed distribution (mean shape parameter).
- **skew** (*int/float*) – Skew value of skewed distribution (skewness shape parameter).
- **scale** (*int/float*) – Scale value of skewed distribution (standard deviation
- **scale** – Scale value of skewed distribution (standard deviation shape parameter). shape parameter).
- **num_skew_samples** (*int*) – Number of random variables to sample from distribution to generate skew data and plot.
- **return_data** (*bool*) – from generated distribution.
- **xlim** (*list*) – X-axis limits of plot. E.g. xlim=[0,10] to plot random variable values between 0 and 10.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.
- **rand_var_name** (*str*) – Name of random variable to label plot's x-axis.
- **num_bins** (*int*) – Number of bins to use in plot. Default is 0, in which case the number of bins chosen will be automatically selected.
- **round_to_nearest** (*int/float*) – Value to round random variables to nearest. E.g. if round_to_nearest=0.2, will round each random variable to nearest 0.2.
- **prob_rand_var_less_than** (*list*) – List of values for which to print the probability that a variable sampled randomly from the generated distribution will be less than. This is useful for replicating distributions from the literature. E.g. prob_rand_var_less_than=[3.7,5.8] will return the probability that a randomly chosen variable is less than 3.7 and 5.8 respectively.
- **occurrence_multiplier** (*int/float*) – When sampling random variables from distribution to create plot and random variable data, use this multiplier to determine number of data points to sample. A higher value will cause the random variable data to match the probability distribution more closely, but will take longer to generate.
- **num_decimal_places** (*int*) – Number of decimal places to random variable values. Need to explicitly state otherwise Python's floating point arithmetic will cause spurious unique random variable value errors when discretising.

Returns**Tuple containing:**

- **prob_dist** (*dict*): Probability distribution whose key-value pairs are random variable value-probability pairs.
- **rand_vars** (*list, optional*): Random variable values sampled from the generated probability distribution. To return, set `return_data=True`.
- **fig** (*matplotlib.figure.Figure, optional*): Probability density and cumulative distribution function plot. To return, set `show_fig=True` and/or `plot_fig=True`.

Return type

tuple

```
trafpy.generator.src.dists.val_dists.gen_uniform_val_dist(min_val, max_val,
                                                           round_to_nearest=None,
                                                           num_decimal_places=2,
                                                           occurrence_multiplier=100,
                                                           path_to_save=None, plot_fig=False,
                                                           show_fig=False, return_data=False,
                                                           xlim=None, logscale=False,
                                                           rand_var_name='Random Variable',
                                                           prob_rand_var_less_than=None,
                                                           num_bins=0, print_data=False)
```

Generates a uniform distribution of random variable values.

Uniform distributions are the most simple distribution. Each random variable value in a uniform distribution has an equal probability of occurring.

Parameters

- **min_val** (*int/float*) – Minimum random variable value.
- **max_val** (*int/float*) – Maximum random variable value.
- **round_to_nearest** (*int/float*) – Value to round random variables to nearest. E.g. if `round_to_nearest=0.2`, will round each random variable to nearest 0.2.
- **num_decimal_places** (*int*) – Number of decimal places to random variable values. Need to explicitly state otherwise Python's floating point arithmetic will cause spurious unique random variable value errors when discretising.
- **occurrence_multiplier** (*int/float*) – When sampling random variables from distribution to create plot and random variable data, use this multiplier to determine number of data points to sample. A higher value will cause the random variable data to match the probability distribution more closely, but will take longer to generate.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated distribution. E.g. `path_to_save='data/dists/my_dist'`.
- **plot_fig** (*bool*) – Whether or not to plot fig. If True, will return fig.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will return and display fig.
- **return_data** (*bool*) – from generated distribution.
- **xlim** (*list*) – X-axis limits of plot. E.g. `xlim=[0,10]` to plot random variable values between 0 and 10.
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **rand_var_name** (*str*) – Name of random variable to label plot's x-axis.

- **prob_rand_var_less_than** (*list*) – List of values for which to print the probability that a variable sampled randomly from the generated distribution will be less than. This is useful for replicating distributions from the literature. E.g. `prob_rand_var_less_than=[3.7,5.8]` will return the probability that a randomly chosen variable is less than 3.7 and 5.8 respectively.
- **num_bins** (*int*) – Number of bins to use in plot. Default is 0, in which case the number of bins chosen will be automatically selected.
- **print_data** (*bool*) – whether or not to print extra information about the generated data.

Returns

Tuple containing:

- **prob_dist** (*dict*): Probability distribution whose key-value pairs are random variable value-probability pairs.
- **rand_vars** (*list, optional*): Random variable values sampled from the generated probability distribution. To return, set `return_data=True`.
- **fig** (*matplotlib.figure.Figure, optional*): Probability density and cumulative distribution function plot. To return, set `show_fig=True` and/or `plot_fig=True`.

Return type

tuple

```
trafpy.generator.src.dists.val_dists.gen_val_dist_data(val_dist, min_val, max_val,
                                                         num_vals_to_gen, path_to_save=None)
```

Generates values between `min_val` and `max_val` following `val_dist` distribution

```
trafpy.generator.src.dists.val_dists.gen_weibull_dist(_alpha, _lambda, size,
                                                       round_to_nearest=None,
                                                       num_decimal_places=2, min_val=None,
                                                       max_val=None, interactive_params=None,
                                                       logscale=False, transparent=True)
```

Generates a Weibull distribution of random variable values.

Weibull distributions often fit scenarios whose random variable values (e.g. ‘time until failure’) are modelled by ‘extreme value theory’ (EVT) in that the values being predicted are more extreme than any previously recorded and, similar to the log-normal distribution, have a low mean but high variance and therefore a long tail/positive skew. Often use to predict time until failure.

E.g. real-world scenarios: Particle sizes generated by grinding, milling & crushing operations, survival times after cancer diagnosis, light bulb failure times, divorce rates, data centre arrival times, etc.

Parameters

- **_alpha** (*int/float*) – Shape parameter. Describes slope of distribution.
`_alpha < 1`: Probability of random variable occurring decreases as values get higher. Occurs in systems with high ‘infant mortality’ in that e.g. defective items occur soon after `t=0` and are therefore weeded out of the population early on.
`_alpha == 1`: Special case of the Weibull distribution which reduces the distribution to an exponential distribution.
`_alpha > 1`: Probability of random variable value occurring increases with time (until peak passes). Occurs in systems with an ‘aging’ process whereby e.g. components are more likely to fail as time goes on.
- **_lambda** (*int/float*) – Weibull scale parameter. Use to shape distribution standard deviation.

- **size** (*int*) – Number of random variable values to sample from distribution.
- **interactive_params** (*dict*) – Dictionary of distribution parameter values (must provide if in interactive mode).
- **logscale** (*bool*) – Whether or not plot should have logscale x-axis and bins.
- **transparent** (*bool*) – Whether or not to make plot bins slightly transparent.

Returns

random variable values generated by sampling from the distribution.

Return type

list

```
trafpy.generator.src.dists.val_dists.x_round(x, round_to_nearest=1, num_decimal_places=2,
                                             print_data=False, min_val=None)
```

Rounds variable to nearest specified value.

Module contents**Submodules**

trafpy.generator.src module

trafpy.generator.src module

trafpy.generator.src module

trafpy.generator.src.builder module

trafpy.generator.src.demand module

trafpy.generator.src.flowcentric module

trafpy.generator.src.interactive module

trafpy.generator.src.jobcentric module

trafpy.generator.src.networks module

Module for generating and plotting networks.

```
trafpy.generator.src.networks.add_edge_capacity_attrs(network, edge, channel_names,
                                                    channel_capacity, bidirectional_links=True)
```

Adds channels and corresponding max channel bytes to single edge in network.

Parameters

- **network** (*networkx graph*) – Network containing edges to which attrs will be added.
- **edge** (*tuple*) – Node-node edge pair.
- **channel_names** (*list*) – List of channel names to add to edge.

- **channel_capacity** (*int, float*) – Capacity to allocate to each channel.
- **bidirectional_links** (*bool*) – If True, each link has capacity split equally between src and dst port. I.e. all links have a src and dst port which are treated separately to incoming and outgoing traffic to and from given node (switch or server).

`trafpy.generator.src.networks.add_edges_capacity_attrs(network, edges, channel_names, channel_capacity, bidirectional_links=True)`

Adds channels & max channel capacities to single edge in network.

To access e.g. the edge going from node 0 to node 1 (edge (0, 1)), you would index the network with `network[0][1]`

To access e.g. the `channel_1` attribute of this particular (0, 1) edge, you would do `network[0][1]['channels']['channel_1']` OR if `bidirectional_links`, you do `network[0][1]['0_to_1_port']['channels']['channel_1']` or `network[0][1]['1_to_0_port']['channels']['channel_1']` depending on which direction of the link you want to access.

Parameters

- **network** (*networkx graph*) – Network containing edges to which attrs will be added.
- **edges** (*list*) – List of node pairs in tuples.
- **channel_names** (*list of str*) – List of channel names to add to edge.
- **channel_capacity** (*int, float*) – Capacity to allocate to each channel.
- **bidirectional_links** (*bool*) – If True, each link has capacity split equally between src and dst port. I.e. all links have a src and dst port which are treated separately to incoming and outgoing traffic to and from given node (switch or server).

`trafpy.generator.src.networks.gen_arbitrary_network(num_eps, ep_label=None, ep_capacity=12500, num_channels=1, racks_dict=None, topology_type=None)`

Generates an arbitrary network with `num_eps` nodes labelled as `ep_label`.

Note that no edges are formed in this network; it is purely for ep name indexing purposes when using Demand class. This is useful where want to use the demand class but not necessarily with a carefully crafted networkx graph that accurately mimics the network you will use for the demands

Parameters

- **num_eps** (*int*) – Number of endpoints in network.
- **ep_label** (*str, int, float*) – Endpoint label (e.g. 'server'). All endpoints will have `ep_label` appended to the start of their label (e.g. 'server_0', 'server_1', ...).
- **ep_capacity** (*int, float*) – Byte capacity per end point channel.
- **num_channels** (*int, float*) – Number of channels on each link in network.
- **racks_dict** (*dict*) – Mapping of which end points are in which racks. Keys are rack ids, values are list of end points. If None, assume there is not clustering/rack system in the network where have different end points in different clusters/racks.

Returns

network object

Return type

networkx graph

`trafpy.generator.src.networks.gen_channel_names(num_channels)`

Generates channel names for channels on each link in network.

```
trafpy.generator.src.networks.gen_fat_tree(k=4, L=2, n=4, ep_label='server', rack_label='rack',
                                           edge_label='edge', aggregate_label='agg',
                                           core_label='core', num_channels=2,
                                           server_to_rack_channel_capacity=500,
                                           rack_to_edge_channel_capacity=1000,
                                           edge_to_agg_channel_capacity=1000,
                                           agg_to_core_channel_capacity=2000,
                                           rack_to_core_channel_capacity=2000, show_fig=False)
```

Generates a perfect fat tree (i.e. all layers have switches with same radix/number of ports).

Top layer is always core (spine) switch layer, bottom layer is always ToR (leaf) layer.

L must be either 2 (core, ToR) or 4 (core, agg, edge, ToR)

N.B. L=2 is commonly referred to as '2-layer Clos' or 'Clos' or 'spine-leaf' topology

Resource for building (scroll down to summary table with equations):

<https://packetpushers.net/demystifying-dcn-topologies-clos-fat-trees-part2/>

Another good resource for data centre topologies etc. in general:

<https://www.oreilly.com/library/view/bgp-in-the/9781491983416/ch01.html#:~:text=The%20most%20common%20routing%20protocol,single%20data%20center%2C%20as%20well.>

Parameters of network:

- number of core (spine) switches = $(k/2)^{(L/2)}$ (top layer)
- number of edge switches (if L=4) = $(k^2)/2$
- number of agg switches (if L=4) = $(k^2)/2$
- number of pods (if L=4) (pod is a group of agg and/or edge switches) = $2*(k/2)^{(L-2)}$
- number of ToR (leaf) switches (racks) = $2*(k/2)^{(L-1)}$ (bottom layer)
- number of server-facing ToR 'host' ports = $2*(k/2)^2$ (can have multiple servers connected to same host port, & can oversubscribe)
- number of servers = number ToR switches * n

Parameters

- **k** (*int*) – Number of ports (links) on each switch (both up and down).
- **L** (*int*) – Number of layers in the fat tree.
- **n** (*int*) – Number of server per rack.
- **ep_label** (*str, int, float*) – Endpoint label (e.g. 'server'). All endpoints will have ep_label appended to the start of their label (e.g. 'server_0', 'server_1', ...).
- **edge_label** (*str, int*) – Label to assign to edge switch nodes
- **aggregate_label** (*str, int*) – Label to assign to edge switch nodes
- **core_label** (*str, int*) – Label to assign to core switch nodes
- **num_channels** (*int, float*) – Number of channels on each link in network
- **server_to_edge_channel_capacity** (*int, float*) – Byte capacity per channel
- **edge_to_agg_channel_capacity** (*int, float*) – (if L==4) Byte capacity per channel
- **agg_to_core_channel_capacity** (*int, float*) – (if L==4) Byte capacity per channel

- **rack_to_core_channel_capacity** (*int, float*) – (if L==2) Byte capacity per channel

Returns

network object

Return type

networkx graph

```
trafpy.generator.src.networks.gen_nsfnet_network(ep_label='server', rack_label='rack', N=0,
                                                num_channels=2,
                                                server_to_rack_channel_capacity=1,
                                                rack_to_rack_channel_capacity=10,
                                                show_fig=False)
```

Generates the standard 14-node NSFNET topology (a U.S. core network).

Parameters

- **ep_label** (*str, int, float*) – Endpoint label (e.g. 'server'). All endpoints will have ep_label appended to the start of their label (e.g. 'server_0', 'server_1', ...).
- **N** (*int*) – Number of servers per rack. If 0, assume all nodes in nsfnet are endpoints
- **num_channels** (*int, float*) – Number of channels on each link in network.
- **server_to_rack_channel_capacity** (*int, float*) – Byte capacity per channel between servers and ToR switch.
- **rack_to_rack_channel_capacity** (*int, float*) – Byte capacity per channel between racks.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will display fig.

Returns

network object

Return type

networkx graph

```
trafpy.generator.src.networks.gen_simple_network(ep_label='server', num_channels=2,
                                                server_to_rack_channel_capacity=500,
                                                show_fig=False)
```

Generates very simple 5-node topology.

Parameters

- **ep_label** (*str, int, float*) – Endpoint label (e.g. 'server'). All endpoints will have ep_label appended to the start of their label (e.g. 'server_0', 'server_1', ...).
- **num_channels** (*int, float*) – Number of channels on each link in network.
- **channel_capacity** (*int, float*) – Byte capacity per channel.
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will display fig.

Returns

network object

Return type

networkx graph

```
trafpy.generator.src.networks.get_endpoints(network, ep_label)
```

Gets list of endpoints of network.

Parameters

- **network** (*networkx graph*) – Networkx object.
- **ep_label** (*str, int, float*) – Endpoint label (e.g. 'server'). All endpoints will have ep_label appended to the start of their label (e.g. 'server_0', 'server_1', ...).

Returns

List of endpoints.

Return type

eps (list)

`trafpy.generator.src.networks.get_fat_tree_positions(net, width_scale=500, height_scale=10)`

Gets networkx positions of nodes in fat tree network for plotting.

`trafpy.generator.src.networks.get_node_type_dict(network, node_types=[])`

Gets dict where keys are node types, values are list of nodes for each node type in graph.

`trafpy.generator.src.networks.init_global_network_attrs(network, max_nw_capacity, num_channels, ep_link_capacity, endpoint_label='server', topology_type='unknown', node_labels=['server'], racks_dict=None)`

Initialises the standard global network attributes of a given network.

Parameters

- **network** (*obj*) – NetworkX object.
- **max_nw_capacity** (*int/float*) – Maximum rate at which info can be reliably transmitted over the network (sum of all link capacities).
- **num_channels** (*int*) – Number of channels on each link in network.
- **topology_type** (*str*) – Label of network topology (e.g. 'fat_tree').
- **node_labels** (*list*) – Label classes assigned to network nodes (e.g. ['server', 'rack', 'edge']).
- **racks_dict** (*dict*) – Which servers/endpoints are in which rack. If None, assume do not have rack system where have multiple servers in one rack.

`trafpy.generator.src.networks.init_network_node_positions(net)`

Initialises network node positions for plotting.

`trafpy.generator.src.networks.plot_network(network, draw_node_labels=True, ep_label='server', network_node_size=2000, font_size=30, linewidths=1, fig_scale=2, path_to_save=None, show_fig=False)`

Plots networkx graph.

Recognises special fat tree network and applies appropriate node positioning, labelling, colouring etc.

Parameters

- **network** (*networkx graph*) – Network object to be plotted.
- **draw_node_labels** (*bool*) – Whether or not to draw node labels on plot.
- **ep_label** (*str, int, float*) – Endpoint label (e.g. 'server'). All endpoints will have ep_label appended to the start of their label (e.g. 'server_0', 'server_1', ...).
- **network_node_size** (*int, float*) – Size of plotted nodes.
- **font_size** (*int, float*) – Size of of font of plotted labels etc.
- **linewidths** (*int, float*) – Width of edges in network.

- **fig_scale** (*int, float*) – Scaling factor to apply to plotted network.
- **path_to_save** (*str*) – Path to directory (with file name included) in which to save generated plot. E.g. `path_to_save='data/my_plot'`
- **show_fig** (*bool*) – Whether or not to plot and show fig. If True, will return and display fig.

Returns

node distribution plotted as a 2d matrix.

Return type

matplotlib.figure.Figure

trafpy.generator.src.tools module

```
class trafpy.generator.src.tools.NumpyEncoder(*, skipkeys=False, ensure_ascii=True,
                                              check_circular=True, allow_nan=True,
                                              sort_keys=False, indent=None, separators=None,
                                              default=None)
```

Bases: JSONEncoder

Special json encoder for numpy types

default (*obj*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

`trafpy.generator.src.tools.calc_graph_diameter(graph)`

Calculate diameter of a single graph.

`trafpy.generator.src.tools.calc_graph_diameters(graphs, multiprocessing_type='none', print_times=False)`

Calculate diameters of a list of graphs.

`trafpy.generator.src.tools.compute_jensen_shannon_distance(p, q)`

`trafpy.generator.src.tools.gen_event_dict(demand_data, event_iter=None)`

Use demand data dict to generate dict for each event in demand data.

`trafpy.generator.src.tools.gen_event_times(interarrival_times, duration_times=None, path_to_save=None)`

Use event interarrival times to generate event times.

```
trafpy.generator.src.tools.get_network_params(eps, all_combinations=False)
```

Returns basic params of network.

If *all_combinations*, will consider all possible pair combinations (i.e. *src-dst* and *dst-src*). If *False*, will consider *src-dst==dst-src* -> get half number of node pairs returned.

```
trafpy.generator.src.tools.load_data_from_json(path_to_load, print_times=True)
```

```
trafpy.generator.src.tools.pickle_data(path_to_save, data, overwrite=False, zip_data=True,  
                                       print_times=True)
```

Save data as a pickle.

```
trafpy.generator.src.tools.save_data_as_csv(path_to_save, data, overwrite=False, print_times=True)
```

Saves data given as a csv file.

```
trafpy.generator.src.tools.save_data_as_json(path_to_save, data, overwrite=False, print_times=True)
```

```
trafpy.generator.src.tools.to_undirected_graph(directed_graph)
```

Converts directed graph to an undirected graph.

```
trafpy.generator.src.tools.unpickle_data(path_to_load, zip_data=True, print_times=True)
```

Re-load previously pickled data.

Module contents

Module contents

trafpy.manager package

Subpackages

trafpy.manager.src package

Subpackages

trafpy.manager.src.routers package

Submodules

trafpy.manager.src.routers.routers module

trafpy.manager.src.routers.rwa module

```
class trafpy.manager.src.routers.rwa.RWA(channel_names, num_k)
```

Bases: `object`

```
check_if_channel_space(graph, edges, channel, flow_size)
```

Takes list of edges to see if all of the edges have enough space for the given demand on a certain channel

Args: - *edges* (list of lists): edges we want to check if have enough space for a certain demand on a certain channel - *channel* (label): channel we want to check if has enough space for the given demand across all given edges - *flow_size*: demand size we want to check if there's space for on the given channel across all given edges

Returns: - True/False

check_if_channel_used(*graph, edges, channel*)

Takes list of edges to see if any one of the edges have used a certain channel

Args: - edges (list of lists): edges we want to check if have used certain channel - channel (label): channel we want to check if has been used by any of the edges

Returns: - True/False

ff_k_shortest_paths(*graph, k_shortest_paths, flow_size*)

Applies first fit algorithm, whereby path with lowest cost (shortest path) is looked at first when considering which route to select, then next etc. When route is considered, a wavelength (starting from lowest) is considered. If not available, move to next highest wavelength. If go through all wavelengths and none available, move to next shortest path and try again. I.e. this is a 'select route first, then select wavelength' k-shortest path first fit RWA algorithm. If no routes- wavelength pairs are available, message is blocked.

Uses this first fit process to allocate a given demand a path and a channel.

Args: - k_shortest_paths (list of lists): the k shortest paths from the source to destination node, with shortest path first etc - channel_names (list of strings): list of channel names that algorithm can consider assigning to each path - flow_size (int, float): size of demand

Returns: - path - channel

get_action(*observation*)

Gets an action (route+channel or blocked) for DCN simulation

get_path_edges(*path*)

Takes a path and returns list of edges in the path

Args: - path (list): path in which you want to find all edges

Returns: - edges (list of lists): all edges contained within the path

k_shortest_paths(*graph, source, target, num_k=None, weight='weight'*)

Uses Yen's algorithm to compute the k-shortest paths between a source and a target node. The shortest path is that with the lowest pathcost, defined by external path_cost() function. Paths are returned in order of path cost, with lowest path cost being first etc.

Args: - source (label): label of source node - target (label): label of destination node - num_k (int, float): number of shortest paths to compute - weight (dict key): dictionary key of value to be 'minimised' when finding 'shortest' paths

Returns: - A (list of lists): list of shortest paths between src and dst

path_cost(*graph, path, weight=None*)

Calculates cost of path. If no weight specified, 1 unit of cost is 1 link/edge in the path

Args: - path (list): list of node labels making up path from src to dst - weight (dict key): label of weight to be considered when evaluating path cost

Returns: - pathcost (int, float): total cost of path

Module contents

trafpy.manager.src.schedulers package

Submodules

trafpy.manager.src.schedulers.agent module

class trafpy.manager.src.schedulers.agent.**Agent**(*Graph, RWA, slot_size, max_F, epsilon, alpha, gamma, agent_type='sarsa_learning'*)

Bases: *SchedulerToolbox*

check_if_end_of_time_slot_decisions(*action, chosen_flows*)

If one of the following is true, agent should stop making decisions this time slot and this method will return True:

- 1) The action chosen is the 'null' action, which is the agent's way of explicitly stating that it doesn't want to schedule anymore flows for this time slot
- 2) The action chosen is a flow that has already been chosen this time slot
- 3) The action chosen is invalid since, due to previously selected actions, there are no paths or channels (lightpaths) available

Args: - action: The action to be checked chosen by the agent - chosen_flows: A list of flows already chosen by the agent

gen_state_from_agent_queues(*agent_queues*)

Uses agent queues to generate state

get_action(*observation*)

get_agent_action(*state*)

get_agent_state_representation(*observation*)

make_epsilon_greedy_policy(*Q_table, epsilon, action_space*)

merge_agent_flow_dict(*agent_flow_dict*)

Merges flow dict of agent into single array

process_reward(*reward, next_observation*)

Take reward from environment that resulted in action from prev time step and use to learn

update_agent_state(*agent_queues, action*)

Updates flow=action in agent_queues to having scheduled = 1, returns updated state

trafpy.manager.src.schedulers.basrpt module

```
class trafpy.manager.src.schedulers.basrpt.BASRPT(Graph, RWA, slot_size, V, packet_size=300,
                                                  time_multiplexing=True, debug_mode=False,
                                                  scheduler_name='basrpt')
```

Bases: *SchedulerToolbox*

```
display_get_action_processing_time(num_decimals=8)
```

```
find_contending_flow(chosen_flow, chosen_flows)
```

Goes through chosen flow possible path & channel combinations & compares to path-channel combinations in chosen flows. Saves all contentions that arise. When all possible contentions have been checked, finds the ‘most contentious’ (i.e. shortest flow completion time) in chosen_flows and returns this as the contending flow (since other flows in contending_flows will have a higher cost than this most contentious flow and therefore if the chosen flow has a lower cost than the most contentious flow, it will also have a lower cost than all competing flows and therefore should replace all contending flows)

```
get_action(observation, print_processing_time=False)
```

```
get_scheduler_action(observation)
```

```
class trafpy.manager.src.schedulers.basrpt.BASRPT_v2(Graph, RWA, slot_size, V, packet_size=300,
                                                      time_multiplexing=True, debug_mode=False,
                                                      scheduler_name='BASRPT')
```

Bases: object

```
cost_function(flow)
```

BASRPT cost function.

```
get_action(observation, print_processing_time=False)
```

```
get_scheduler_action(observation)
```

trafpy.manager.src.schedulers.fair_share module

```
class trafpy.manager.src.schedulers.fair_share.FairShare(Graph, RWA, slot_size, packet_size=300,
                                                         time_multiplexing=True,
                                                         debug_mode=False,
                                                         scheduler_name='FS')
```

Bases: object

```
get_action(observation, print_processing_time=False)
```

```
get_scheduler_action(observation, path_channel_assignment_strategy='fair_share_num_flows')
```

trafpy.manager.src.schedulers.first_fit module

```
class trafpy.manager.src.schedulers.first_fit.FirstFit(Graph, RWA, slot_size, packet_size=300,
                                                    time_multiplexing=True,
                                                    debug_mode=False, scheduler_name='FF')
```

Bases: object

```
get_action(observation, print_processing_time=False)
```

```
get_scheduler_action(observation)
```

trafpy.manager.src.schedulers.lambda_share module

```
class trafpy.manager.src.schedulers.lambda_share.LambdaShare(Graph, RWA, slot_size,
                                                            _lambda=0.5, packet_size=300,
                                                            time_multiplexing=True,
                                                            debug_mode=False,
                                                            scheduler_name='S')
```

Bases: object

```
get_action(observation, print_processing_time=False)
```

```
get_scheduler_action(observation)
```

trafpy.manager.src.schedulers.parametric_agent module**trafpy.manager.src.schedulers.random_agent module**

```
class trafpy.manager.src.schedulers.random_agent.RandomAgent(Graph, RWA, slot_size,
                                                            packet_size=300,
                                                            time_multiplexing=True,
                                                            debug_mode=False,
                                                            scheduler_name='Rand')
```

Bases: object

```
get_action(observation, print_processing_time=False)
```

```
get_scheduler_action(observation)
```

trafpy.manager.src.schedulers.schedulers module**trafpy.manager.src.schedulers.schedulertoolbox module**

```
class trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox
```

Bases: object

```
class trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2(Graph, RWA,
                                                                           slot_size,
                                                                           packet_size=300,
                                                                           time_multiplexing=True,
                                                                           debug_mode=False)
```

Bases: object

allocate_available_bandwidth(*flow_info*, *resolution_strategy*)

Goes through each edge and allocates bandwidth available on that edge to requesting flows until either no bandwidth left to allocate or all requesting flows would be completed this time slot.

If *flow_id_to_cost* is not None, will allocate bandwidth to flows in order of cost (prioritising low cost flows first). If *flow_id_to_cost* is None, must specify a valid *resolution_strategy* (e.g. 'random', 'fair_share', etc.).

Parameters

- **flow_id_to_cost** (*dict*) – Dict mapping *flow_id* to corresponding flow cost.
- **resolution_strategy** (*str*) – Which resolution strategy to use if *flow_id_to_cost* is None to allocate available bandwidth and resolve conflicts

Returns:

check_connection_valid(*flow*, *num_decimals*=6)

Returns False if setting up connection would result in -ve bandwidth on at least one link in network.

check_edge_valid(*flow*, *edge*, *num_decimals*=6)

collect_flow_info_dicts(*path_channel_assignment_strategy*='random', *cost_function*=None)

Goes through network and collects useful dictionaries for making scheduling decisions.

Parameters

- **path_channel_assignment_strategy** (*str*) – If 'random', allocates flows a randomly chosen path and channel. If None, does not allocate any path or channel; just uses whichever path and channel has already been assigned (if nothing assigned, will get an error later). If 'fair_share_num_flows', distributes number of flows across channels and paths equally. If 'fair_share_num_packets', distributes number of flow packets across channels and paths equally.
- **cost_function** (*function*) – If not None, uses *cost_function* to assign a cost to each flow and stores this in a dictionary. *cost_function* must take a single flow dictionary argument.

Returns

Maps *flow_id* to corresponding flow dictionary. *requested_edges* (dict): Maps links (edges) in network being requested

to corresponding flow ids requesting them.

edge_to_flow_ids (dict): Maps edge to the list of flow ids requesting it.

flow_id_to_cost (dict): Maps *flow_id* to corresponding cost of flow.

Return type

queued_flows (dict)

estimate_time_to_completion(*flow*)

filter_unavailable_flows()

Takes a network and filters out any flow that is not ready to be scheduled yet i.e. has incomplete parent flow dependencies. Use this method to get network representation for 'job-agnostic' flow scheduling systems.

find_flow_idx(*flow, flows*)

Finds flow idx in a list of flows. Assumes the following flow features are unique and unchanged properties of each flow: - flow size - source - destination - time arrived - flow_id - job_id

Args: - flow (dict): flow dictionary - flows (list of dicts) list of flows in which to find flow idx

find_flow_queue(*flow*)

Finds queue of flow in network

gen_flow_packets(*flow_size*)**get_channel_bandwidth(*edge, channel*)**

Gets current channel bandwidth left on a given edge in the network.

get_edge_to_bandwidth_dict(*requested_edges, max_bw=True*)

Goes through network and maps each edge to its maximum bandwidth.

If max_bw, gets maximum possible bandwidth on each edge. If not max_bw, gets available bandwidth on each edge ASSUMES ONE CHANNEL.

get_lowest_edge_bandwidth(*path, max_bw=True, channel=None*)

Goes through path edges and finds bandwidth of lowest bandwidth edge port.

If max_bw, will return maximum possible bandwidth of lowest max bandwidth edge. If not, will return available bandwidth of lowest available bandwidth edge.

N.B. if not max_bw, MUST given channel (since checking bandwidth available on edge)

get_path_edges(*path*)

Takes a path and returns list of edges in the path

Args: - path (list): path in which you want to find all edges

Returns: - edges (list of lists): all edges contained within the path

init_paths_and_packets(*flow_dict*)**remove_flow_from_queue(*flow_dict, network*)**

Given flow dict and network that flow is in, will locate flow in network and remove from queue

reset()**reset_channel_capacities_of_edges()**

Takes edges and resets their available capacities back to their maximum capacities.

resolve_contentions_and_set_up_flow(*flow, chosen_flows, flow_info, scheduling_info, cost_info, resolution_strategy*)

If contention found, will resolve contention using resolution strategy.

Cost resolution strategy -> choose flow with lowest cost. Random resolution strategy -> choose random flow.

set_up_connection(*flow, num_decimals=6*)

Sets up connection between src-dst node pair by removing capacity from all edges in path connecting them. Also updates graph's global curr network capacity used property

Args: - flow (dict): flow dict containing flow info to set up

take_down_connection(*flow*, *num_decimals*=6)

Removes established connection by adding capacity back onto all edges in the path connecting the src-dst node pair. Also updates graph's global curr network capacity used property

Args: - flow (dict): flow dict containing info of flow to take down

update_network_state(*observation*, *hide_child_dependency_flows*=True, *reset_channel_capacities*=True)

If *hide_child_dependency_flows* is True, will only update scheduler network to see flows that are ready to be scheduled i.e. all parent flow dependencies have been completed. This is used for 'job-agnostic' scheduling systems which, rather than considering the job that each flow is part of, only consider the flow.

If False, will just update network with all flows (even those that cannot yet be scheduled). This is used for 'job- & network- aware' scheduling systems.

trafpy.manager.src.schedulers.srpt module

```
class trafpy.manager.src.schedulers.srpt.SRPT(Graph, RWA, slot_size, packet_size=300,
                                             time_multiplexing=True, debug_mode=False,
                                             scheduler_name='srpt')
```

Bases: [SchedulerToolbox](#)

display_get_action_processing_time(*num_decimals*=8)

find_contending_flow(*chosen_flow*, *chosen_flows*)

Goes through chosen flow possible path & channel combinations & compares to path-channel combinations in chosen flows. Saves all contentions that arise. When all possible contentions have been checked, finds the 'most contentious' (i.e. shortest flow completion time) in chosen_flows and returns this as the contending flow (since other flows in contending_flows will have a higher FCT than this most contentious flow and therefore if the chosen flow has a lower FCT than the most contentious flow, it will also have a lower FCT than all competing flows and therefore should replace all contending flows)

get_action(*observation*, *print_processing_time*=False)

get_scheduler_action(*observation*)

Uses observation and chosen rwa action(s) to construct schedule for this timeslot

```
class trafpy.manager.src.schedulers.srpt.SRPT_v2(Graph, RWA, slot_size, packet_size=300,
                                                  time_multiplexing=True, debug_mode=False,
                                                  scheduler_name='SRPT')
```

Bases: object

cost_function(*flow*)

SRPT cost function.

get_action(*observation*, *print_processing_time*=False)

get_scheduler_action(*observation*, *reset_channel_capacities*=True, *path_channel_assignment_strategy*='fair_share_num_flows')

Module contents

trafpy.manager.src.simulators package

Submodules

trafpy.manager.src.simulators.analysers module

trafpy.manager.src.simulators.dcn module

trafpy.manager.src.simulators.env_analyser module

```
class trafpy.manager.src.simulators.env_analyser.EnvAnalyser(env, time_units='a.u.',
                                                            info_units='a.u.',
                                                            subject_class_name=None)
```

Bases: object

```
compute_metrics(measurement_start_time=None, measurement_end_time=None,
                  env_analyser_database_path=None, overwrite=False, print_summary=False)
```

measurement_start_time (int, float): Simulation time at which to begin recording
metrics etc.; is the warm-up time

measurement_end_time (int, float): Simulation time at which to stop recording
metrics etc.; is the cool-down time

If overwrite is False and an analyser object exists in env_analyser_database_path, will load previously saved analyser object rather than re-computing everything. To overwrite this previously saved analyser, set overwrite=True.

If tmp_database_path is not None, will store data in tmp_database_path str specified. This can help with memory errors as avoids holding everything in RAM memory.

trafpy.manager.src.simulators.env_plotter module

trafpy.manager.src.simulators.plotters module

trafpy.manager.src.simulators.simulators module

Module contents

Module contents

Module contents

3.7.2 Module contents

INDEX

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

- trafpy, 71
- trafpy.benchmark, 38
- trafpy.benchmark.config, 38
- trafpy.benchmark.main_gen_benchmark_data,
38
- trafpy.benchmark.test, 38
- trafpy.benchmark.versions, 38
- trafpy.benchmark.versions.benchmark, 36
- trafpy.benchmark.versions.benchmark_importer,
37
- trafpy.benchmark.versions.benchmark_v001,
36
- trafpy.benchmark.versions.old_benchmark_importer,
37
- trafpy.generator.src, 63
- trafpy.generator.src.dists, 57
- trafpy.generator.src.dists.node_dists, 38
- trafpy.generator.src.dists.val_dists, 45
- trafpy.generator.src.interactive, 57
- trafpy.generator.src.networks, 57
- trafpy.generator.src.tools, 62
- trafpy.manager.src, 71
- trafpy.manager.src.routers, 65
- trafpy.manager.src.routers.routers, 63
- trafpy.manager.src.routers.rwa, 63
- trafpy.manager.src.schedulers, 71
- trafpy.manager.src.schedulers.agent, 65
- trafpy.manager.src.schedulers.basrpt, 66
- trafpy.manager.src.schedulers.fair_share, 66
- trafpy.manager.src.schedulers.first_fit, 67
- trafpy.manager.src.schedulers.lambda_share,
67
- trafpy.manager.src.schedulers.random_agent,
67
- trafpy.manager.src.schedulers.schedulertoolbox,
67
- trafpy.manager.src.schedulers.srpt, 70
- trafpy.manager.src.simulators, 71
- trafpy.manager.src.simulators.analysers, 71
- trafpy.manager.src.simulators.env_analyser,
71

INDEX

A

add_edge_capacity_attrs() (in module *trafpy.generator.src.networks*), 57

add_edges_capacity_attrs() (in module *trafpy.generator.src.networks*), 58

adjust_node_dist_for_rack_prob_config() (in module *trafpy.generator.src.dists.node_dists*), 38

adjust_node_dist_from_multinomial_exp_for_rack_prob_config() (in module *trafpy.generator.src.dists.node_dists*), 39

adjust_probability_array_sum() (in module *trafpy.generator.src.dists.node_dists*), 39

adjust_probability_dict_sum() (in module *trafpy.generator.src.dists.node_dists*), 39

Agent (class in *trafpy.manager.src.schedulers.agent*), 65

allocate_available_bandwidth() (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox* method), 68

assign_matrix_to_probs() (in module *trafpy.generator.src.dists.node_dists*), 39

assign_probs_to_matrix() (in module *trafpy.generator.src.dists.node_dists*), 39

B

BASRPT (class in *trafpy.manager.src.schedulers.basrpt*), 66

BASRPT_v2 (class in *trafpy.manager.src.schedulers.basrpt*), 66

Benchmark (class in *trafpy.benchmarker.versions.benchmark*), 36

BenchmarkImporter (class in *trafpy.benchmarker.versions.benchmark_importer*), 37

BenchmarkImporter (class in *trafpy.benchmarker.versions.old_benchmark_importer*), 37

C

calc_graph_diameter() (in module *trafpy.generator.src.tools*), 62

calc_graph_diameters() (in module *trafpy.generator.src.tools*), 62

check_connection_valid() (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox* method), 68

check_edge_valid() (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox* method), 68

check_if_channel_space() (*trafpy.manager.src.routers.rwa.RWA* method), 63

check_if_channel_used() (*trafpy.manager.src.routers.rwa.RWA* method), 64

check_if_end_of_time_slot_decisions() (*trafpy.manager.src.schedulers.agent.Agent* method), 65

collect_flow_info_dicts() (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox* method), 68

combine_multiple_mode_dists() (in module *trafpy.generator.src.dists.val_dists*), 45

combine_skews() (in module *trafpy.generator.src.dists.val_dists*), 45

compute_jensen_shannon_distance() (in module *trafpy.generator.src.tools*), 62

compute_metrics() (*trafpy.manager.src.simulators.env_analyser.EnvAnalyser* method), 71

convert_data_to_key_occurrences() (in module *trafpy.generator.src.dists.val_dists*), 45

convert_key_occurrences_to_data() (in module *trafpy.generator.src.dists.val_dists*), 46

convert_pair_prob_dist_dict_to_matrix_pair_prob_dist_dict() (in module *trafpy.generator.src.dists.node_dists*), 40

convert_sampled_pairs_into_node_dist() (in module *trafpy.generator.src.dists.node_dists*), 40

cost_function() (*trafpy.manager.src.schedulers.basrpt.BASRPT_v2* method), 66

cost_function() (*trafpy.manager.src.schedulers.srpt.SRPT_v2* method), 70

D

`default()` (*trafpy.generator.src.tools.NumpyEncoder* method), 62

`display_get_action_processing_time()` (*trafpy.manager.src.schedulers.basrpt.BASRPT* method), 66

`display_get_action_processing_time()` (*trafpy.manager.src.schedulers.srpt.SRPT* method), 70

`do_work()` (in module *trafpy.benchmark.test*), 38

E

`EnvAnalyser` (class in *trafpy.manager.src.simulators.env_analyser*), 71

`estimate_time_to_completion()` (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2* method), 68

F

`FairShare` (class in *trafpy.manager.src.schedulers.fair_share*), 66

`ff_k_shortest_paths()` (*trafpy.manager.src.routers.rwa.RWA* method), 64

`filter_unavailable_flows()` (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2* method), 68

`find_contending_flow()` (*trafpy.manager.src.schedulers.basrpt.BASRPT* method), 66

`find_contending_flow()` (*trafpy.manager.src.schedulers.srpt.SRPT* method), 70

`find_flow_idx()` (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2* method), 68

`find_flow_queue()` (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2* method), 69

`FirstFit` (class in *trafpy.manager.src.schedulers.first_fit*), 67

G

`gen_arbitrary_network()` (in module *trafpy.generator.src.networks*), 58

`gen_channel_names()` (in module *trafpy.generator.src.networks*), 58

`gen_demand_nodes()` (in module *trafpy.generator.src.dists.node_dists*), 40

`gen_discrete_prob_dist()` (in module *trafpy.generator.src.dists.val_dists*), 46

`gen_event_dict()` (in module *trafpy.generator.src.tools*), 62

`gen_event_times()` (in module *trafpy.generator.src.tools*), 62

`gen_exponential_dist()` (in module *trafpy.generator.src.dists.val_dists*), 46

`gen_fat_tree()` (in module *trafpy.generator.src.networks*), 58

`gen_flow_packets()` (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2* method), 69

`gen_lognormal_dist()` (in module *trafpy.generator.src.dists.val_dists*), 47

`gen_multimodal_node_dist()` (in module *trafpy.generator.src.dists.node_dists*), 40

`gen_multimodal_node_pair_dist()` (in module *trafpy.generator.src.dists.node_dists*), 41

`gen_multimodal_val_dist()` (in module *trafpy.generator.src.dists.val_dists*), 48

`gen_named_val_dist()` (in module *trafpy.generator.src.dists.val_dists*), 49

`gen_node_demands()` (in module *trafpy.generator.src.dists.node_dists*), 42

`gen_normal_dist()` (in module *trafpy.generator.src.dists.val_dists*), 50

`gen_nsfnet_network()` (in module *trafpy.generator.src.networks*), 60

`gen_pareto_dist()` (in module *trafpy.generator.src.dists.val_dists*), 51

`gen_rand_vars_from_discretised_dist()` (in module *trafpy.generator.src.dists.val_dists*), 51

`gen_simple_network()` (in module *trafpy.generator.src.networks*), 60

`gen_skew_data()` (in module *trafpy.generator.src.dists.val_dists*), 52

`gen_skew_dists()` (in module *trafpy.generator.src.dists.val_dists*), 53

`gen_skewnorm_data()` (in module *trafpy.generator.src.dists.val_dists*), 53

`gen_skewnorm_val_dist()` (in module *trafpy.generator.src.dists.val_dists*), 53

`gen_state_from_agent_queues()` (*trafpy.manager.src.schedulers.agent.Agent* method), 65

`gen_uniform_multinomial_exp_node_dist()` (in module *trafpy.generator.src.dists.node_dists*), 43

`gen_uniform_node_dist()` (in module *trafpy.generator.src.dists.node_dists*), 43

`gen_uniform_val_dist()` (in module *trafpy.generator.src.dists.val_dists*), 55

`gen_val_dist_data()` (in module *trafpy.generator.src.dists.val_dists*), 56

`gen_weibull_dist()` (in module *trafpy.generator.src.dists.val_dists*), 56

`get_action()` (*trafpy.manager.src.routers.rwa.RWA* method), 64

`get_action()` (*trafpy.manager.src.schedulers.agent.Agent* method), 65

[get_action\(\)](#) ([traffy.manager.src.schedulers.basrpt.BASRPT](#) method), 66
[get_action\(\)](#) ([traffy.manager.src.schedulers.basrpt.BASRPT_v2](#) method), 66
[get_action\(\)](#) ([traffy.manager.src.schedulers.fair_share.FairShare](#) method), 66
[get_action\(\)](#) ([traffy.manager.src.schedulers.first_fit.FirstFit](#) method), 67
[get_action\(\)](#) ([traffy.manager.src.schedulers.lambda_share.LambdaShare](#) method), 67
[get_action\(\)](#) ([traffy.manager.src.schedulers.random_agent.RandomAgent](#) method), 67
[get_action\(\)](#) ([traffy.manager.src.schedulers.srpt.SRPT](#) method), 70
[get_action\(\)](#) ([traffy.manager.src.schedulers.srpt.SRPT_v2](#) method), 70
[get_agent_action\(\)](#) ([traffy.manager.src.schedulers.agent.Agent](#) method), 65
[get_agent_state_representation\(\)](#) ([traffy.manager.src.schedulers.agent.Agent](#) method), 65
[get_benchmark_dists\(\)](#) ([traffy.benchmarker.versions.benchmark_importer.BenchmarkImporter](#) method), 37
[get_benchmark_dists\(\)](#) ([traffy.benchmarker.versions.old_benchmark_importer.BenchmarkImporter](#) method), 37
[get_channel_bandwidth\(\)](#) ([traffy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2](#) method), 69
[get_dist_and_path\(\)](#) ([traffy.benchmarker.versions.benchmark.Benchmark](#) method), 36
[get_edge_to_bandwidth_dict\(\)](#) ([traffy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2](#) method), 69
[get_endpoints\(\)](#) (in module [traffy.generator.src.networks](#)), 60
[get_fat_tree_positions\(\)](#) (in module [traffy.generator.src.networks](#)), 61
[get_flow_size_dist\(\)](#) ([traffy.benchmarker.versions.benchmark.Benchmark](#) method), 36
[get_inter_intra_rack_pair_prob_dicts\(\)](#) (in module [traffy.generator.src.dists.node_dists](#)), 44
[get_interarrival_time_dist\(\)](#) ([traffy.benchmarker.versions.benchmark.Benchmark](#) method), 36
[get_lowest_edge_bandwidth\(\)](#) ([traffy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2](#) method), 69
[get_network_pair_mapper\(\)](#) (in module [traffy.generator.src.dists.node_dists](#)), 44
[get_network_params\(\)](#) (in module [traffy.generator.src.tools](#)), 62
[get_node_dist\(\)](#) ([traffy.benchmarker.versions.benchmark.Benchmark](#) method), 36
[get_node_type_dict\(\)](#) (in module [traffy.generator.src.networks](#)), 61
[get_num_ops_dist\(\)](#) ([traffy.benchmarker.versions.benchmark.Benchmark](#) method), 37
[get_pair_prob_dict_of_node_dist_matrix\(\)](#) (in module [traffy.generator.src.dists.node_dists](#)), 44
[get_path_edges\(\)](#) ([traffy.manager.src.routers.rwa.RWA](#) method), 64
[get_path_edges\(\)](#) ([traffy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2](#) method), 69
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.basrpt.BASRPT](#) method), 66
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.basrpt.BASRPT_v2](#) method), 66
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.fair_share.FairShare](#) method), 66
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.first_fit.FirstFit](#) method), 67
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.lambda_share.LambdaShare](#) method), 67
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.random_agent.RandomAgent](#) method), 67
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.srpt.SRPT](#) method), 70
[get_scheduler_action\(\)](#) ([traffy.manager.src.schedulers.srpt.SRPT_v2](#) method), 70
[get_suitable_destination_node_for_rack_config\(\)](#) (in module [traffy.generator.src.dists.node_dists](#)), 44
[init_global_network_attrs\(\)](#) (in module [traffy.generator.src.networks](#)), 61
[init_network_node_positions\(\)](#) (in module [traffy.generator.src.networks](#)), 61
[init_paths_and_packets\(\)](#) ([traffy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2](#) method), 69
[k_shortest_paths\(\)](#) ([traffy.manager.src.routers.rwa.RWA](#) method), 64

method), 64

L

LambdaShare (class in *trafpy.manager.src.schedulers.lambda_share*), 67

load_data_from_json() (in module *trafpy.generator.src.tools*), 63

load_dist() (*trafpy.benchmark.versions.benchmark.Benchmark* *method*), 37

M

make_epsilon_greedy_policy() (*trafpy.manager.src.schedulers.agent.Agent* *method*), 65

merge_agent_flow_dict() (*trafpy.manager.src.schedulers.agent.Agent* *method*), 65

module

trafpy, 71

trafpy.benchmark, 38

trafpy.benchmark.config, 38

trafpy.benchmark.main_gen_benchmark_data, 38

trafpy.benchmark.test, 38

trafpy.benchmark.versions, 38

trafpy.benchmark.versions.benchmark, 36
trafpy.benchmark.versions.benchmark_importer, 37

trafpy.benchmark.versions.benchmark_v001, 36

trafpy.benchmark.versions.old_benchmark_importer, 37

trafpy.generator.src, 57, 63

trafpy.generator.src.dists, 57

trafpy.generator.src.dists.node_dists, 38

trafpy.generator.src.dists.val_dists, 45

trafpy.generator.src.interactive, 57

trafpy.generator.src.networks, 57

trafpy.generator.src.tools, 62

trafpy.manager.src, 71

trafpy.manager.src.routers, 65

trafpy.manager.src.routers.routers, 63

trafpy.manager.src.routers.rwa, 63

trafpy.manager.src.schedulers, 71

trafpy.manager.src.schedulers.agent, 65

trafpy.manager.src.schedulers.basrpt, 66

trafpy.manager.src.schedulers.fair_share, 66

trafpy.manager.src.schedulers.first_fit, 67

trafpy.manager.src.schedulers.lambda_share_scheduler_toolbox_v2, 67

trafpy.manager.src.schedulers.random_agent, 67

trafpy.manager.src.schedulers.scheduler_toolbox, 67

trafpy.manager.src.schedulers.srpt, 70

trafpy.manager.src.simulators, 71

trafpy.manager.src.simulators.analysers, 71

trafpy.manager.src.simulators.env_analyser, 71

N

NumpyEncoder (class in *trafpy.generator.src.tools*), 62

P

path_cost() (*trafpy.manager.src.routers.rwa.RWA* *method*), 64

pickle_data() (in module *trafpy.generator.src.tools*), 63

plot_network() (in module *trafpy.generator.src.networks*), 61

process_reward() (*trafpy.manager.src.schedulers.agent.Agent* *method*), 65

R

RandomAgent (class in *trafpy.manager.src.schedulers.random_agent*), 67

remove_flow_from_queue() (*trafpy.manager.src.schedulers.scheduler_toolbox.SchedulerToolbox* *method*), 69

reset() (*trafpy.manager.src.schedulers.scheduler_toolbox.SchedulerToolbox* *method*), 69

reset_channel_capacities_of_edges() (*trafpy.manager.src.schedulers.scheduler_toolbox.SchedulerToolbox* *method*), 69

resolve_contentions_and_set_up_flow() (*trafpy.manager.src.schedulers.scheduler_toolbox.SchedulerToolbox* *method*), 69

RWA (class in *trafpy.manager.src.routers.rwa*), 63

S

save_data_as_csv() (in module *trafpy.generator.src.tools*), 63

save_data_as_json() (in module *trafpy.generator.src.tools*), 63

save_dist() (*trafpy.benchmark.versions.benchmark.Benchmark* *method*), 37

SchedulerToolbox (class in *trafpy.manager.src.schedulers.scheduler_toolbox*), 67

SchedulerToolbox_v2 (class in *trafpy.manager.src.schedulers.scheduler_toolbox*), 67

set_up_connection()
 (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2*
 method), 69
 SRPT (*class in trafpy.manager.src.schedulers.srpt*), 70
 SRPT_v2 (*class in trafpy.manager.src.schedulers.srpt*), 70
T
 take_down_connection()
 (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2*
 method), 69
 to_undirected_graph() (in *module*
 trafpy.generator.src.tools), 63
 trafpy
 module, 71
 trafpy.benchmarkmarker
 module, 38
 trafpy.benchmarkmarker.config
 module, 38
 trafpy.benchmarkmarker.main_gen_benchmark_data
 module, 38
 trafpy.benchmarkmarker.test
 module, 38
 trafpy.benchmarkmarker.versions
 module, 38
 trafpy.benchmarkmarker.versions.benchmark
 module, 36
 trafpy.benchmarkmarker.versions.benchmark_importer
 module, 37
 trafpy.benchmarkmarker.versions.benchmark_v001
 module, 36
 trafpy.benchmarkmarker.versions.old_benchmark_importer
 module, 37
 trafpy.generator.src
 module, 57, 63
 trafpy.generator.src.dists
 module, 57
 trafpy.generator.src.dists.node_dists
 module, 38
 trafpy.generator.src.dists.val_dists
 module, 45
 trafpy.generator.src.interactive
 module, 57
 trafpy.generator.src.networks
 module, 57
 trafpy.generator.src.tools
 module, 62
 trafpy.manager.src
 module, 71
 trafpy.manager.src.routers
 module, 65
 trafpy.manager.src.routers.routers
 module, 63
 trafpy.manager.src.routers.rwa
 module, 63
 trafpy.manager.src.schedulers
 module, 71
 trafpy.manager.src.schedulers.agent
 module, 65
 trafpy.manager.src.schedulers.basrpt
 module, 66
 trafpy.manager.src.schedulers.fair_share
 module, 66
 trafpy.manager.src.schedulers.first_fit
 module, 67
 trafpy.manager.src.schedulers.lambda_share
 module, 67
 trafpy.manager.src.schedulers.random_agent
 module, 67
 trafpy.manager.src.schedulers.schedulertoolbox
 module, 67
 trafpy.manager.src.schedulers.srpt
 module, 70
 trafpy.manager.src.simulators
 module, 71
 trafpy.manager.src.simulators.analysers
 module, 71
 trafpy.manager.src.simulators.env_analyser
 module, 71
U
 unpickle_data() (in *module*
 trafpy.generator.src.tools), 63
 update_agent_state()
 (*trafpy.manager.src.schedulers.agent.Agent*
 method), 65
 update_network_state()
 (*trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox_v2*
 method), 70
 update_pbar() (in *module trafpy.benchmarkmarker.test*), 38
X
 x_round() (in *module*
 trafpy.generator.src.dists.val_dists), 57