TafPy Documentation

version

C W F Parsonson

December 21, 2020

Contents

Overview of TrafPy

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

TrafPy provides:

- a pre-built **interactive Jupyter Notebook** tool for visually building distributions and data which accurately mimic traffic characteristics of real networks (e.g. data centres);
- a **generator** package for generating network traffic which can be flexibly integrated into custom Python projects; and
- a **manager** package which can be used to simulate network management (scheduling, routing etc.) following the standard OpenAI Gym reinforcement learning framework.

TrafPy can be used to quickly and easily replicate traffic distributions from the literature even in the absense 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 TrafPy, then have a look at the tutorial.

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

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/cwfparsonson/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.generator as tpg
>>> from trafpy.manager import Demand, RWA, SRPT, DCN
```

Tutorial

This guide can help you start with TrafPy.

Note

TrafPy users who are not familiar with Python can read the 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 (CSVs can be generated and imported into e.g. MATLAB scripts; no Python knowledge is required).

TrafPy Generator

Import the `trafpy.generator` package.

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

Network traffic patterns can be characterised by probability distributions. By accurately describing a probability distribution, one can sample from it to generate arbitrary amounts of realistic network traffic.

Value Distributions

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_ne
```

TrafPy probability distributions are defined as Python dictionaries with value-probability key-value pairs

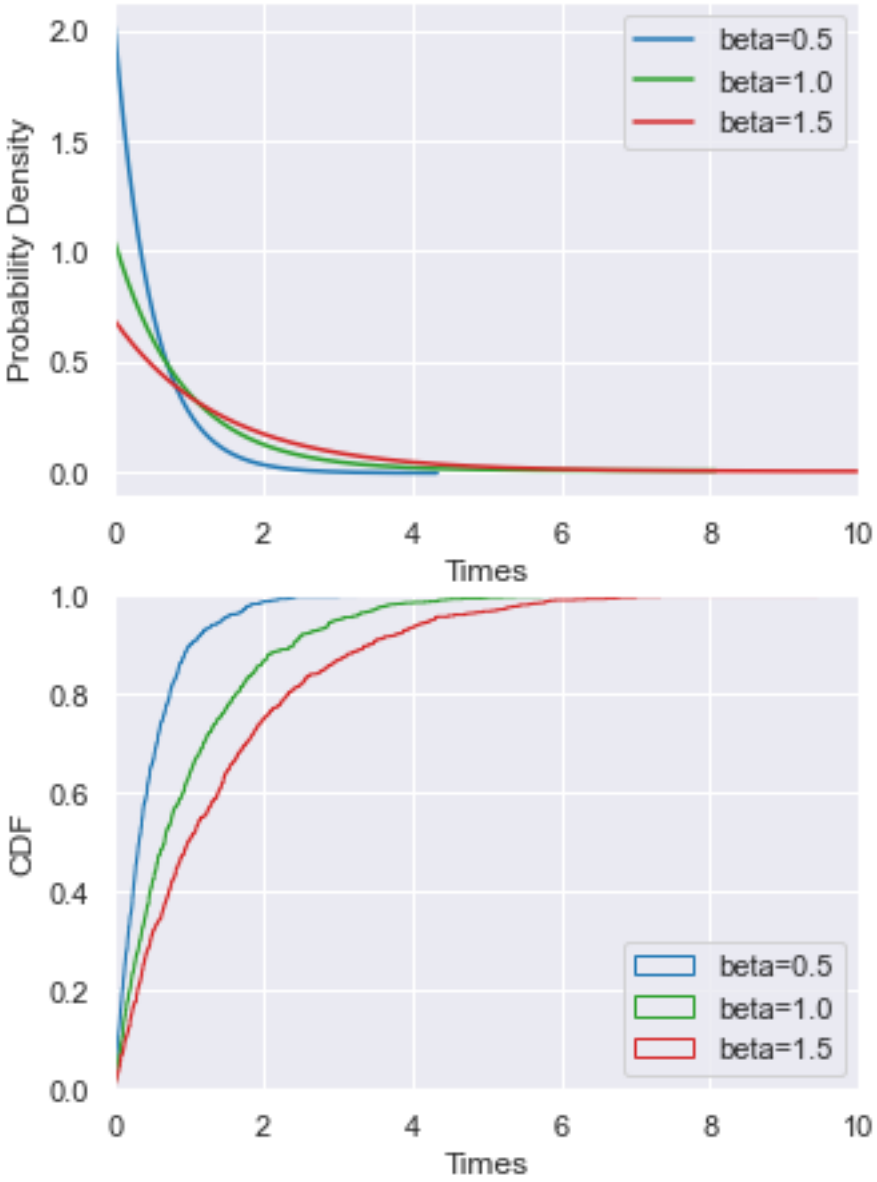
```
>>> print('Uniform probability distribution:\n{}'.format(prob_dist))
Uniform probability distribution:
{1: 0.01, 2: 0.01, 3: 0.01, 4: 0.01, 5: 0.01, 6: 0.01, 7: 0.01, 8: 0.01, 9: 0.01, 10: 0.01,
```

and the probability density plot is constructed by sampling random variables from the discrete probability distribution.

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.

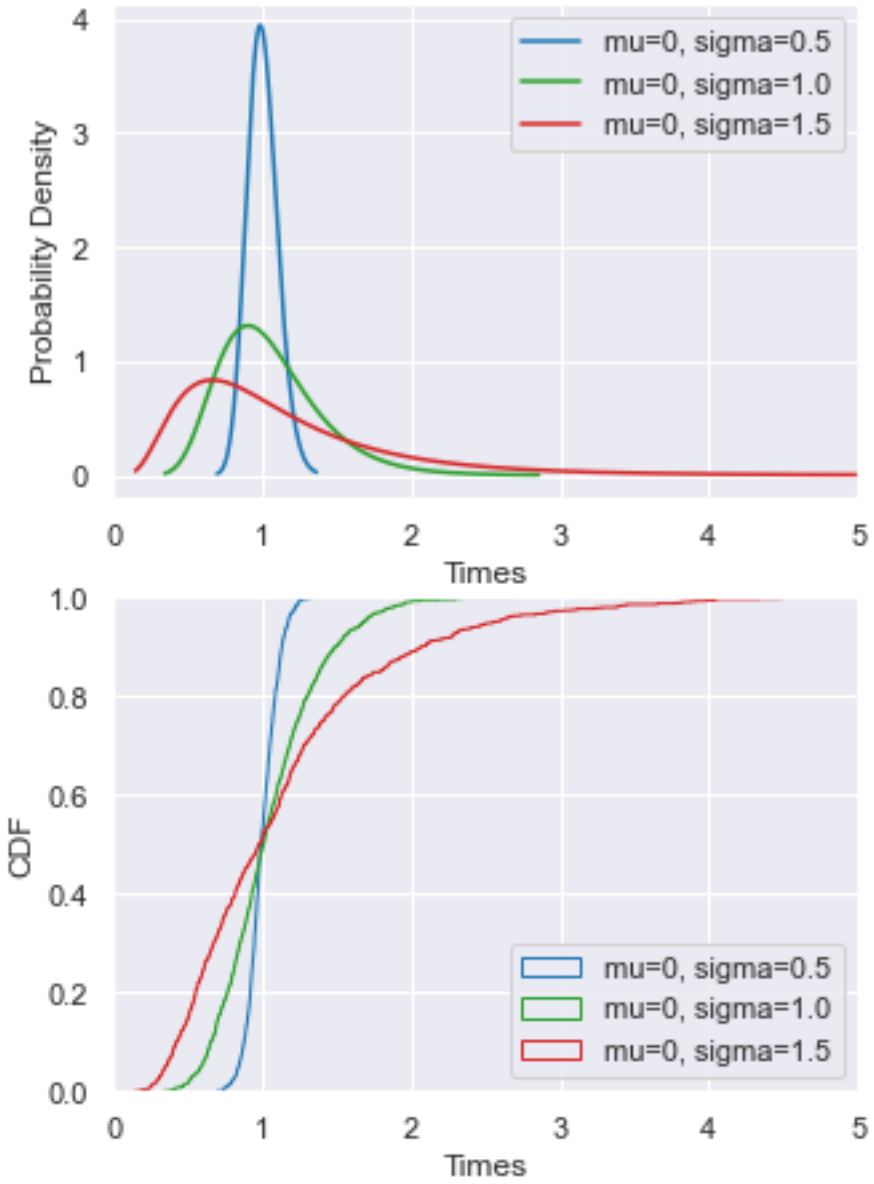
Named distributions supported by TrafPy include the *exponential distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='exponential', params={'_beta':
```



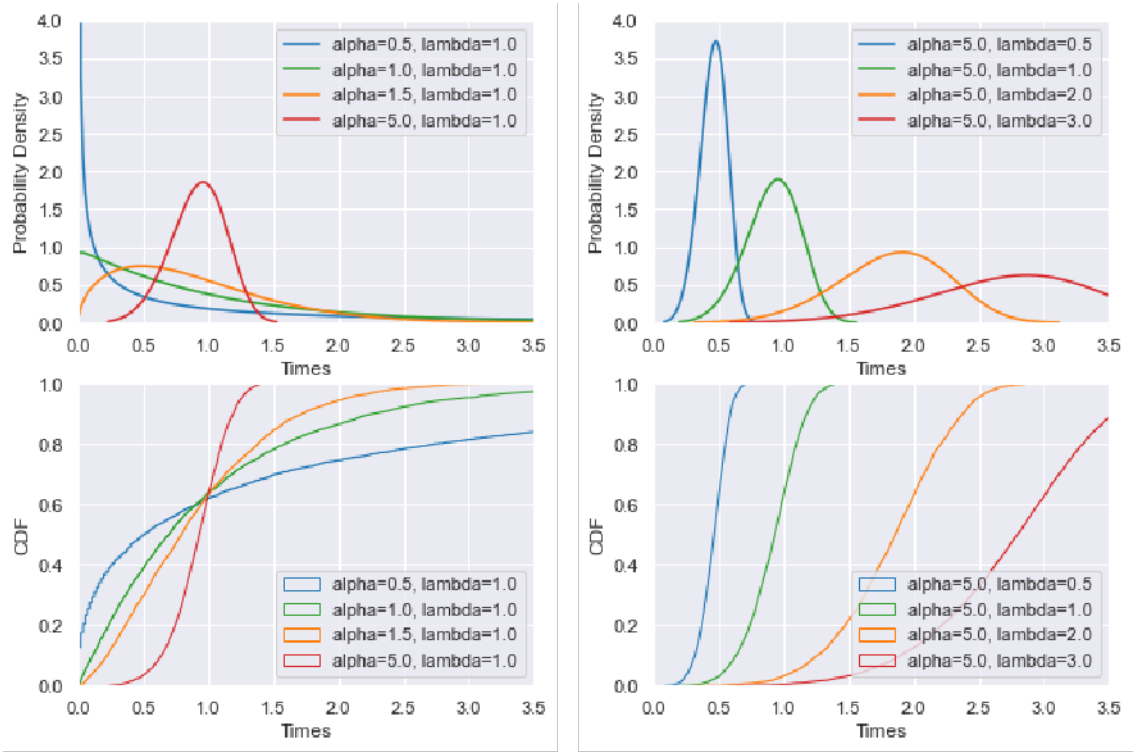
the *log-normal distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='lognormal', params={'_mu': 0, '
```



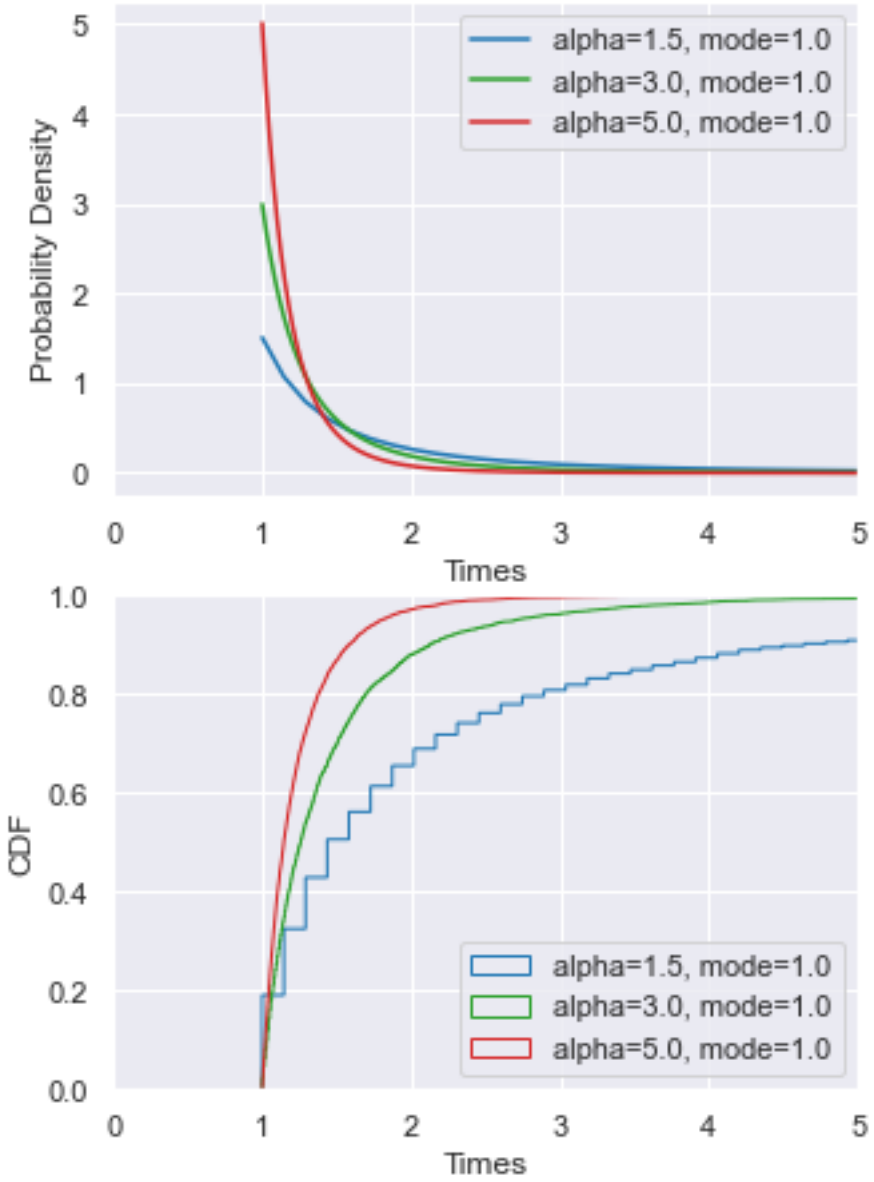
the *Weibull distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='weibull', params={'_alpha': 1.5
```



and the *Pareto distribution*

```
>>> prob_dist, rand_vars, fig = tpg.gen_named_val_dist(dist='pareto', params={'_alpha': 3.0,
```



However, some demand characteristics cannot be accurately described by these named distributions. Instead, they are described by arbitrary **multimodal distributions**, which are distributions with more than one **mode** which may or may not have some amount of **skewness** and/or **kurtosis**

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

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 probability 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()),p
```

Node Distributions

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 probability matrix**. 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'],sh
```

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_
```

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)
```

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

```
>>> sn, dn = tpg.gen_node_demands(eps=endpoints,node_dist=node_dist,num_demands=1000)
```

Networks

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 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.

Flow-Centric Traffic Demands

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

```
>>> num_demands = 1000
>>> 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, '_lambd
```

then the flow interarrival time distribution

```
>>> interarrival_time_dist, _ = tpg.gen_named_val_dist(dist='lognormal',params={'_mu': 7.4,
```

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)
```

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

```
>>> flow_centric_demand_data = tpg.create_demand_data(num_demands=num_demands,eps=endpoints,
```

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.pi
```

or as a csv:

```
tpg.save_data_as_csv(data=flow_centric_demand_data,path_to_save='data/flow_centric_demand_da
```

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
```

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],
    'establish': [event_establish_flow_0, ..., event_establish_flow_1],
    'index': [index_flow_0, ..., index_flow_1]
}
```

Where `'establish'` keys' values are binary values indicating whether the demand is a connection establishment request (1) or a take-down request (0) for a given event. Specifying take-down requests is optional in TrafPy. If take-downs have been speified, then there will be `2 * num_demands` events in the demand data dictionary, otherwise there will be `num_demands` events.

Job-Centric Traffic Demands

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 >= 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).

```
>>> num_demands = 10
>>> tpg.gen_simple_network(ep_label='endpoint')
```

You could start by definiing 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=
```

then the job interarrival time distribution

```
>>> interarrival_time_dist = tpg.gen_multimodal_val_dist(min_val=1,max_val=1e8,locations=[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=
```

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(num_demands=num_demands,eps=endpoints,n
```

Don't forget to save your data:

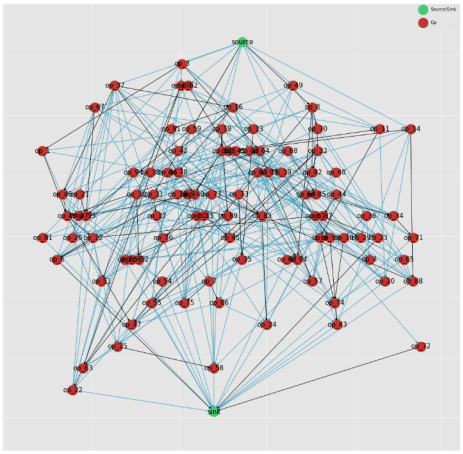
```
tpg.pickle_data(data=job_centric_demand_data,path_to_save='data/job_centric_demand_data.pick
```

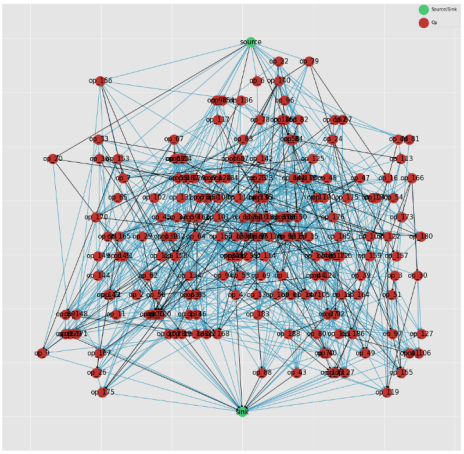
TrafPy job-centric demand data dictionaries are organised as:

```
{
    '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)
```





Additional Functionality

Up to now, you have used TrafPy to create some number of demands. However, it is often more useful to create an arbitrary number of demands such that a certain **network load** is generated for an arbitrary network. Here, the **network capacity** is the total *rate* at which information can be reliably transmitted over the network, and the network load is the fraction of this capacity being requested during the session.

To specify the load and have the number of demands automatically generated, the same functions you've seen above can be used, but now specifying the `network_load_config` argument rather than `num_demands`. E.g. to generate demands that will request a 0.75 load of a network with a network capacity of 6,000 Gbps:

TEMPORARY COMMENT OUT OF BELOW CODE - BRING DEMO IN LATER WHEN FINALISED THIS FUNCTIONALITY #.. nbplot:

```
#     >>> network_load_config = {'network_rate_capacity': 6000, 'target_load_fraction': 0.75}
#     >>> flow_centric_demand_data = tpg.create_demand_data(network_load_config=network_load_
```

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 probabilites. 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 20% of traffic inter-rack in a fat-tree topology:

```
>>> net = tpg.gen_fat_tree(k=3, N=2, num_channels=1)
>>> 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'], 'rack_1': ['server_2', 'server_3'],
'rack_2': ['server_4', 'server_5'], 'rack_3': ['server_6', 'server_7'],
'rack_4': ['server_8', 'server_9'], 'rack_5': ['server_10', 'server_11']}
```

```
>>> rack_prob_config = {'racks_dict': net.graph['rack_to_ep_dict'], 'prob_inter_rack': 0.20}
>>> node_dist, _ = tpg.gen_uniform_node_dist(net.graph['endpoints'], rack_prob_config=rack_p
```

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_p
```

Visually Shaping TrafPy Distributions

Up until now you have assumed you already knew all the parameters of each distribution you have generated with TrafPy. But what if you want to replicate a distribution which has either not been produced in TrafPy before or has not had open-access data provided? TrafPy has a useful interactive Jupyter-Notebook which integrates with all of the above functions, allowing distributions to be visually shaped. Crucially, once a distribution has been shaped, it can be easily replicated with TrafPy so long as the set of parameters used to shape the distribution are shared.

Academic papers present network traffic distribution information in many forms. It could be e.g. a plot, an analyticaly described named distribution (e.g. ‘the connection duration times followed a log-normal distribution with mu -3.8 and sigma 6.4’), an analytically described unnamed distribution (e.g. ‘the flow sizes followed a distribution with minimum 8, maximum 33,000, mean 6,450, skewness 1.23, and kurtosis 2.03’) etc.

The TrafPy Jupyter Notebook tool enables distributions to be tuned visually and analytically to reproduce literature distributions. Distribution plots are live-updated as slide bars, text boxes etc. are adjusted, with analytical characteristics 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

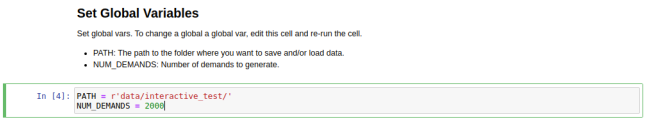
To run a Jupyter Notebook cell, click on the cell and click ‘Run’ on the top ribbon. If you are running a cell with a TrafPy interactive graph, some configurable parameters will appear. Adjust these parameters and click the `Run Interact` button to update your plot (and the returned values).

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.

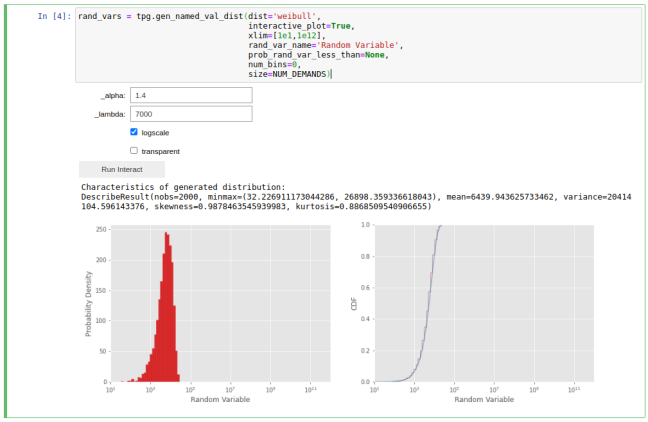
Set Global Variables

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.



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()`**.

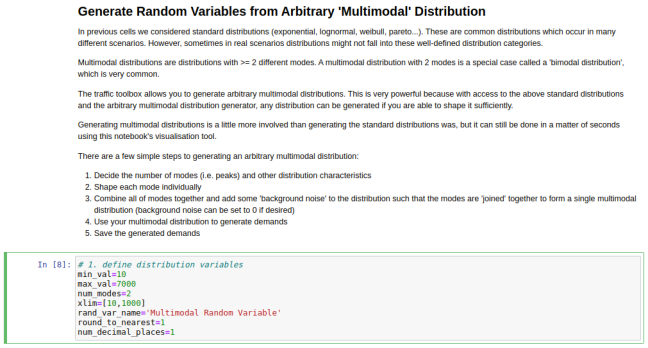


Generate Random Variables from Arbitrary ‘Multimodal’ Distribution

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.

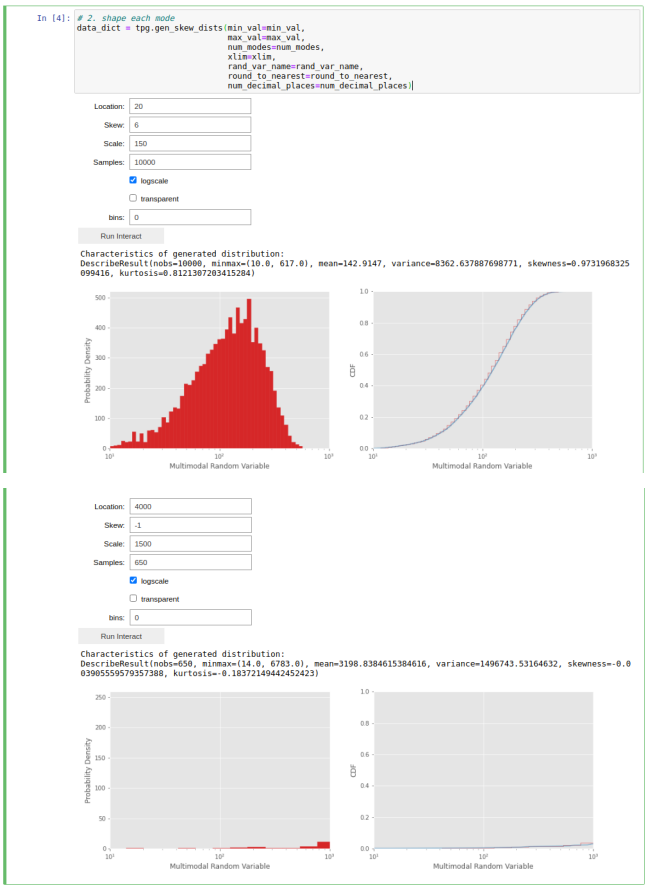
Generating multimodal distributions is a little more involved than generating the standard distributions was, but it can still be done in a matter of seconds using this notebook's visualisation tool.

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

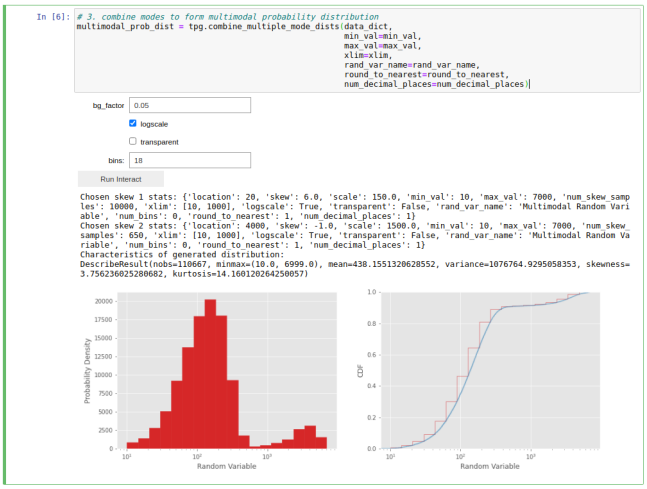
1. Decide the number of modes (i.e. peaks) and other distribution characteristics
2. Shape each mode individually
3. Combine all of modes together and add some 'background noise' to the distribution such that the modes are 'joined' together to form a single multimodal distribution (background noise can 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_nodes` 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).



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



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.

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

Generate Distributions in Sets

Note

This is an extension of the interactive toolbox primarily for TrafPy users who are not familiar with Python and want to simply run the Notebook to generate distribution and/or random variable data CSV files to import into their own e.g. MATLAB scripts. Users familiar with Python are encouraged to shape their distributions and then implement their own TrafPy scripts, and may therefore omit this final cell.

The final cell in the TrafPy Jupyter Notebook tool allows users to generate distributions without visualisation (i.e. the above cells are needed to first shape the distributions, or some prior shaping parameters are needed). This is useful for generating large amounts of data in 'sets', where some sets may have different demand characteristics/distributions from others.

Simply configure the variables under `# set vars` (e.g. the number of sets `num_sets` and the number of demands in each set `num_demands`). Any distributions to keep constant across all sets should be defined outside the for loop, and those that should change should be defined within.

Note

This is a basic script written for a specific use-case. Adjusting it to your specific needs may require some basic Python knowledge.

TrafPy Manager

Note

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.

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 requried 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
```

Initialise the `trafpy.manager` objects:

```
network = tpg.gen_simple_network(ep_label=config.ENDPOINT_LABEL,num_channels=config.NUM_CHAN
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
```

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))
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 ar
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
```

Contribute

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

Development Workflow

1. If you are a first-time contributor:
    - Go to https://github.com/cwfparsonson/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:cwfparsonson/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 -
#
# 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.

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
```

License

TrafPy is distributed with the Apache License 2.0.

Citing

setup module

trafpy package

Subpackages

trafpy.benchmarker package

Subpackages

trafpy.benchmarker.versions package

Subpackages

trafpy.benchmarker.versions.benchmark_v001 package

Submodules

trafpy.benchmarker.versions.benchmark_v001.benchmark_plot_script module

trafpy.benchmarker.versions.benchmark_v001.config module

trafpy.benchmarker.versions.benchmark_v001.distribution_generator module

class trafpy.benchmarker.versions.benchmark_v001.distribution_generator.**DistributionGenerator** (load_prev_dists=True)

Bases: **object**

conv_fig_to_image (fig, dpi=300)

Takes matplotlib figure and converts into numpy array of RGB pixel values

get_flow_size_dist (benchmark, save_data=True)

get_interarrival_time_dist (benchmark, save_data=True)

get_node_dist (benchmark, racks_dict, eps, save_data=True)

init_dir (benchmark)

load_dist (benchmark, dist_name)

plot_benchmark_dists (benchmarks)

Plots dist info of all benchmark(s).
e.g. benchmarks = [‘uniform’, ‘university’]

Module contents

Submodules

trafpy.benchmarker.versions.benchmark_importer module

class trafpy.benchmarker.versions.benchmark_importer.**BenchmarkImporter** (benchmark_version, load_prev_dists=True)
Bases: **object**

get_benchmark_dists (benchmark, racks_dict, eps)

Module contents

Submodules

trafpy.benchmarker.config module

trafpy.benchmarker.main_gen_benchmark_data module

trafpy.benchmarker.main_testbed_benchmark_data module

class trafpy.benchmarker.main_testbed_benchmark_data.**TestBed** (path_to_benchmark_data)
Bases: **object**

load_benchmark_data (demand_file_path)

reset ()

run_test (scheduler, env, envs, path_to_save)

run_tests (config, path_to_save)

save (path, overwrite=False, conv_back_to_mp_manager_list=False)

trafpy.benchmarker.ray_main_testbed_benchmark_data module

class trafpy.benchmarker.ray_main_testbed_benchmark_data.**TestBed** (path_to_benchmark_data)
Bases: **object**

load_benchmark_data (demand_file_path)

reset ()

run_test = <*ray.remote_function.RemoteFunction object*>

run_tests (config, path_to_save)

save (path, overwrite=False, conv_back_to_mp_manager_list=False)

trafpy.benchmarker.tools module

trafpy.benchmarker.tools.**gen_benchmark_demands** (path_to_save=None, save_format='json', load_prev_dists=True)

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_cor`**
(rack_prob_config, eps, node_dist, num_exps_factor=2, 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_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_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=[['server_8', 'server_28'], ['server_5', 'server_45'], ['server_35', 'server_46'], ['server_20', 'server_42'], ['server_18', 'server_42'], ['server_32', 'server_50'], ['server_6', 'server_47'], ['server_0', 'server_46'], ['server_13', 'server_29'], ['server_17', 'server_24'], ['server_42', 'server_43'], ['server_49', 'server_56'], ['server_16', 'server_21'], ['server_9', 'server_61'], ['server_8', 'server_21'], ['server_11', 'server_34'], ['server_4', 'server_57'], ['server_32', 'server_58'], ['server_20', 'server_56'], ['server_56', 'server_58'], ['server_18', 'server_33'], ['server_2', 'server_25'], ['server_18', 'server_22'], ['server_0', 'server_59'], ['server_33', 'server_49'], ['server_5', 'server_47'], ['server_26', 'server_30'], ['server_7', 'server_46'], ['server_31', 'server_54'], ['server_35', 'server_50'], ['server_38', 'server_45'], ['server_4', 'server_25'], ['server_12', 'server_23'], ['server_9', 'server_27'], ['server_51', 'server_52'], ['server_2', 'server_41'], ['server_12', 'server_13'], ['server_31', 'server_50'], ['server_6', 'server_17'], ['server_3', 'server_30'], ['server_9', 'server_18'], ['server_9', 'server_21'], ['server_8', 'server_59'], ['server_17', 'server_40'], ['server_24', 'server_27'], ['server_7', 'server_40'], ['server_44', 'server_54'], ['server_17', 'server_33'], ['server_6', 'server_25'], ['server_48', 'server_61'], ['server_30', 'server_55'], ['server_4', 'server_59'], ['server_8', 'server_54'], ['server_38', 'server_59'], ['server_26', 'server_59'], ['server_28', 'server_62'], ['server_20', 'server_63'], ['server_45', 'server_54'], ['server_42', 'server_55'], ['server_16', 'server_44'], ['server_10', 'server_49'], ['server_45', 'server_62'], ['server_7', 'server_15'], ['server_3', 'server_37'], ['server_38', 'server_43'], ['server_13', 'server_34'], ['server_4', 'server_53'], ['server_19', 'server_53'], ['server_8', 'server_32'], ['server_29', 'server_63'], ['server_24', 'server_42'], ['server_50', 'server_56'], ['server_29', 'server_53'], ['server_9', 'server_20'], ['server_11', 'server_22'], ['server_22', 'server_62'], ['server_14', 'server_22'], ['server_4', 'server_52'], ['server_22', 'server_50'], ['server_33', 'server_37'], ['server_0', 'server_10'], ['server_2', 'server_35'], ['server_11', 'server_51'], ['server_23', 'server_33'], ['server_1', 'server_42'], ['server_32', 'server_51'], ['server_4', 'server_26'], ['server_39', 'server_63'], ['server_24', 'server_32'], ['server_22', 'server_41'], ['server_46', 'server_59'], ['server_23', 'server_27'], ['server_16', 'server_28'], ['server_20', 'server_36'], ['server_8', 'server_41'], ['server_23', 'server_35'], ['server_3', 'server_28'], ['server_20', 'server_32'], ['server_20', 'server_24'], ['server_49', 'server_63'], ['server_19', 'server_26'], ['server_28', 'server_40'], ['server_20', 'server_41'], ['server_33', 'server_61'], ['server_27', 'server_62'], ['server_13', 'server_41'], ['server_9', 'server_24'], ['server_17', 'server_52'], ['server_19', 'server_31'], ['server_0', 'server_22'], ['server_23', 'server_29'], ['server_13', 'server_23'],

['server_5', 'server_34'], ['server_17', 'server_29'], ['server_45', 'server_48'], ['server_3', 'server_16'], ['server_12', 'server_42'], ['server_20', 'server_62'], ['server_24', 'server_34'], ['server_48', 'server_57'], ['server_0', 'server_31'], ['server_29', 'server_60'], ['server_1', 'server_10'], ['server_44', 'server_49'], ['server_0', 'server_36'], ['server_36', 'server_39'], ['server_42', 'server_60'], ['server_33', 'server_56'], ['server_20', 'server_51'], ['server_31', 'server_55'], ['server_6', 'server_30'], ['server_4', 'server_47'], ['server_35', 'server_42'], ['server_16', 'server_25'], ['server_47', 'server_51'], ['server_40', 'server_49'], ['server_6', 'server_12'], ['server_13', 'server_45'], ['server_41', 'server_58'], ['server_42', 'server_54'], ['server_4', 'server_27'], ['server_9', 'server_25'], ['server_7', 'server_29'], ['server_19', 'server_58'], ['server_8', 'server_37'], ['server_0', 'server_52'], ['server_3', 'server_53'], ['server_8', 'server_39'], ['server_25', 'server_27'], ['server_15', 'server_40'], ['server_3', 'server_47'], ['server_14', 'server_43'], ['server_28', 'server_51'], ['server_39', 'server_50'], ['server_13', 'server_24'], ['server_36', 'server_56'], ['server_7', 'server_62'], ['server_1', 'server_50'], ['server_31', 'server_60'], ['server_10', 'server_26'], ['server_30', 'server_57'], ['server_14', 'server_17'], ['server_11', 'server_36'], ['server_7', 'server_10'], ['server_17', 'server_30'], ['server_2', 'server_10'], ['server_3', 'server_42'], ['server_42', 'server_45'], ['server_23', 'server_59'], ['server_55', 'server_58'], ['server_13', 'server_50'], ['server_15', 'server_62'], ['server_30', 'server_49'], ['server_19', 'server_59'], ['server_7', 'server_45'], ['server_37', 'server_60'], ['server_8', 'server_14'], ['server_6', 'server_32'], ['server_9', 'server_42'], ['server_26', 'server_56'], ['server_3', 'server_31'], ['server_4', 'server_21'], ['server_38', 'server_55'], ['server_14', 'server_55'], ['server_28', 'server_29'], ['server_0', 'server_47'], ['server_4', 'server_19'], ['server_15', 'server_59'], ['server_50', 'server_53'], ['server_0', 'server_2'], ['server_16', 'server_31'], ['server_8', 'server_31'], ['server_19', 'server_37'], ['server_20', 'server_27'], ['server_34', 'server_37'], ['server_1', 'server_27'], ['server_31', 'server_52'], ['server_14', 'server_58'], ['server_3', 'server_54'], ['server_6', 'server_50'], ['server_59', 'server_61'], ['server_5', 'server_11'], ['server_0', 'server_39'], ['server_6', 'server_59'], ['server_28', 'server_44'], ['server_6', 'server_42'], ['server_5', 'server_22'], ['server_27', 'server_41'], ['server_28', 'server_63'], ['server_22', 'server_48'], ['server_10', 'server_15'], ['server_53', 'server_55'], ['server_5', 'server_24'], ['server_38', 'server_61'], ['server_24', 'server_39'], ['server_54', 'server_62'], ['server_46', 'server_63'], ['server_33', 'server_59'], ['server_21', 'server_53'], ['server_38', 'server_42'], ['server_5', 'server_38'], ['server_57', 'server_58'], ['server_24', 'server_53'], ['server_43', 'server_45'], ['server_11', 'server_32'], ['server_8', 'server_50'], ['server_58', 'server_59'], ['server_7', 'server_39'], ['server_0', 'server_17'], ['server_13', 'server_18'], ['server_40', 'server_50'], ['server_5', 'server_60'], ['server_28', 'server_59'], ['server_34', 'server_43'], ['server_36', 'server_60'], ['server_29', 'server_51'], ['server_8', 'server_51'], ['server_6', 'server_23'], ['server_51', 'server_53'], ['server_58', 'server_61'], ['server_11', 'server_63'], ['server_19', 'server_33'], ['server_8', 'server_31'], ['server_15', 'server_48'], ['server_19', 'server_39'], ['server_3', 'server_49'], ['server_33', 'server_50'], ['server_2', 'server_37'], ['server_7', 'server_56'], ['server_25', 'server_58'], ['server_8', 'server_46'], ['server_46', 'server_55'], ['server_16', 'server_20'], ['server_0', 'server_6'], ['server_9', 'server_60'], ['server_15', 'server_28'], ['server_16', 'server_56'], ['server_25', 'server_60'], ['server_32', 'server_48'], ['server_7', 'server_49'], ['server_40', 'server_49'], ['server_0', 'server_43'], ['server_4', 'server_37'], ['server_21', 'server_60'], ['server_49', 'server_50'], ['server_1', 'server_35'], ['server_16', 'server_43'], ['server_14', 'server_56'], ['server_57', 'server_63'], ['server_18', 'server_52'], ['server_24', 'server_61'], ['server_0', 'server_58'], ['server_7', 'server_20'], ['server_12', 'server_43'], ['server_3', 'server_52'], ['server_21', 'server_48'], ['server_32', 'server_60'], ['server_55', 'server_59'], ['server_5', 'server_25'], ['server_14', 'server_60'], ['server_21', 'server_33'], ['server_9', 'server_35'], ['server_30', 'server_52'], ['server_44', 'server_52'], ['server_39', 'server_59'], ['server_22', 'server_51'], ['server_37', 'server_53'], ['server_33', 'server_53'], ['server_6', 'server_32'], ['server_12', 'server_14'], ['server_40', 'server_44'], ['server_15', 'server_54'], ['server_6', 'server_22'], ['server_22', 'server_24'], ['server_43', 'server_59'], ['server_21', 'server_23'], ['server_16', 'server_50'], ['server_45', 'server_53'], ['server_30', 'server_48'], ['server_44', 'server_63'], ['server_21', 'server_34'], ['server_2', 'server_24'], ['server_14', 'server_21'], ['server_3', 'server_13'], ['server_15', 'server_60'], ['server_19', 'server_29'], ['server_19', 'server_40'], ['server_27', 'server_46'], ['server_13', 'server_17'], ['server_36', 'server_43'], ['server_4', 'server_6'], ['server_20', 'server_52'], ['server_45', 'server_60'], ['server_49', 'server_59'], ['server_29', 'server_34'], ['server_3', 'server_7'], ['server_14', 'server_48'], ['server_3', 'server_34'], ['server_20', 'server_23'], ['server_16', 'server_46'], ['server_5', 'server_32'], ['server_3', 'server_19'], ['server_19', 'server_52'], ['server_9', 'server_13'], ['server_20', 'server_45'], ['server_39', 'server_55'], ['server_10', 'server_49'], ['server_0', 'server_25'], ['server_34', 'server_42'], ['server_35', 'server_51'], ['server_5', 'server_35'], ['server_37', 'server_46'], ['server_39', 'server_40'], ['server_36', 'server_47'], ['server_11', 'server_33'], ['server_17', 'server_35'], ['server_4', 'server_39'], ['server_0', 'server_50'], ['server_34', 'server_45'], ['server_51', 'server_59'], ['server_6', 'server_38'], ['server_22', 'server_38'], ['server_0', 'server_34'], ['server_37', 'server_51'], ['server_60', 'server_62'], ['server_44', 'server_53'], ['server_30', 'server_44'], ['server_5', 'server_62'], ['server_17', 'server_55'], ['server_23', 'server_55'], ['server_24', 'server_44'], ['server_7', 'server_12'], ['server_37', 'server_42'], ['server_2', 'server_10'], ['server_21', 'server_63'], ['server_41', 'server_56'], ['server_34', 'server_52'], ['server_6', 'server_34'], ['server_35', 'server_48'], ['server_48', 'server_49'], ['server_48', 'server_58'], ['server_5', 'server_18'], ['server_31', 'server_63'], ['server_0', 'server_35'], ['server_6', 'server_39'], ['server_16', 'server_59'], ['server_4', 'server_50'], ['server_13', 'server_19'], ['server_12', 'server_30'], ['server_24', 'server_54'], ['server_36', 'server_45'], ['server_5', 'server_9'], ['server_13', 'server_57'], ['server_17', 'server_58'], ['server_25', 'server_38'], ['server_19', 'server_61'], ['server_26', 'server_61'], ['server_5', 'server_40'], ['server_55', 'server_63'], ['server_50', 'server_52'], ['server_5', 'server_42'], ['server_7', 'server_55'], ['server_12', 'server_58'], ['server_22', 'server_28'], ['server_0', 'server_51'], ['server_10', 'server_39'], ['server_29',

'server_58'], ['server_16', 'server_35'], ['server_26', 'server_33'], ['server_36', 'server_48'], ['server_15', 'server_23'], ['server_59', 'server_62'], ['server_11', 'server_35'], ['server_14', 'server_46'], ['server_3', 'server_51'], ['server_14', 'server_62'], ['server_24', 'server_46'], ['server_16', 'server_27'], ['server_24', 'server_50'], ['server_8', 'server_16'], ['server_20', 'server_48'], ['server_39', 'server_57'], ['server_11', 'server_17'], ['server_31', 'server_57'], ['server_15', 'server_51'], ['server_44', 'server_46'], ['server_0', 'server_15'], ['server_48', 'server_60'], ['server_52', 'server_58'], ['server_6', 'server_46'], ['server_0', 'server_11'], ['server_4', 'server_35'], ['server_33', 'server_42'], ['server_1', 'server_52'], ['server_33', 'server_34'], ['server_0', 'server_29'], ['server_24', 'server_37'], ['server_25', 'server_57'], ['server_45', 'server_63'], ['server_12', 'server_17'], ['server_23', 'server_32'], ['server_6', 'server_9'], ['server_13', 'server_37'], ['server_33', 'server_39'], ['server_47', 'server_61'], ['server_30', 'server_38'], ['server_13', 'server_25'], ['server_1', 'server_21'], ['server_43', 'server_62'], ['server_35', 'server_59'], ['server_28', 'server_56'], ['server_6', 'server_24'], ['server_13', 'server_39'], ['server_16', 'server_54'], ['server_34', 'server_56'], ['server_31', 'server_51'], ['server_23', 'server_46'], ['server_1', 'server_58'], ['server_48', 'server_51'], ['server_10', 'server_29'], ['server_3', 'server_36'], ['server_7', 'server_11'], ['server_38', 'server_58'], ['server_4', 'server_15'], ['server_24', 'server_63'], ['server_8', 'server_22'], ['server_58', 'server_62'], ['server_2', 'server_31'], ['server_62', 'server_63'], ['server_22', 'server_47'], ['server_6', 'server_51'], ['server_36', 'server_49'], ['server_18', 'server_51']], 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 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_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 is 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 souce-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 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.`**`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)

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

`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

Module for plotting node and value distributions.

`trafpy.generator.src.dists.plot_dists.`**`plot_demand_slot_colour_grid`**(grid_demands, title=None, xlim=None, show_fig=False)

`trafpy.generator.src.dists.plot_dists.`**`plot_multiple_kdes`**(plot_dict={}, plot_hist=False, xlabel='Random Variable', ylabel='Density', logscale=False, show_fig=False)

`trafpy.generator.src.dists.plot_dists.`**`plot_node_dist`**(node_dist, eps=None, node_to_index_dict=None, add_labels=False, add_ticks=False, show_fig=False)

Plots network node demand probability distribution as a 2D matrix.

Parameters:

- **node_dist** (*list or 2d numpy array*) – Source-destination pair probabilities of being chosen. Must be either a 2d numpy matrix of probabilities or a 1d list/array of node pair probabilities.
- **eps** (*list*) – List of node endpoint labels.
- **node_to_index_dict** (*dict*) – Maps node labels (keys) to integer indices (values).
- **add_labels** (*bool*) – Whether or not to node labels to plot.
- **add_ticks** (*bool*) – Whether or not to add ticks to x- and y-axis.
- **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.dists.plot_dists.`**`plot_val_bar`** (x_values, y_values, ylabel='Random Variable', ylim=None, xlabel=None, plot_x_ticks=True, bar_width=0.35, show_fig=False)

Plots standard bar chart.

`trafpy.generator.src.dists.plot_dists.`**`plot_val_cdf`** (plot_dict={}, xlabel='Random Variable', ylabel='CDF', logscale=False, plot_points=True, complementary_cdf=False, show_fig=False)

Plots CDF plot.

plot_dict= {'class_1': {'rand_vars': [0.1, 0.1, 0.3],

'class_2': {'rand_vars': [0.2, 0.2, 0.3]}}

`trafpy.generator.src.dists.plot_dists.`**`plot_val_dist`** (rand_vars, dist_fit_line=None, xlim=None, logscale=False, transparent=False, rand_var_name='Random Variable', prob_rand_var_less_than=None, num_bins=0, plot_cdf=True, plot_horizontally=True, fig_scale=1, font_size=20, show_fig=False)

Plots (1) probability distribution and (2) cumulative distribution function.

Parameters:

- **rand_vars** (*list*) – Random variable values.
- **dist_fit_line** (*str*) – Line to fit to named distribution. E.g. 'exponential'. If not plotting a named distribution, leave as None.
- **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.
- **plot_cdf** (*bool*) – Whether or not to plot the CDF as well as the probability distribution.
- **plot_horizontally** (*bool*) – Wheter to plot PDF and CDF horizontally (True) or vertically (False).
- **fig_scale** (*int/float*) – Scale by which to multiply figure size.
- **font_size** (*int*) – Size of axes ticks and titles.
- **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.dists.plot_dists.`**`plot_val_line`** (plot_dict={}, xlabel='Random Variable', ylabel='Random Variable Value', ylim=None, linewidth=1, alpha=1, vertical_lines=[], show_fig=False)

Plots line plot.

plot_dict= {'class_1': {'x_values': [0.1, 0.2, 0.3], 'y_values': [20, 40, 80]},

'class_2': {'x_values': [0.1, 0.2, 0.3], 'y_values': [80, 60, 20]}}

`trafpy.generator.src.dists.plot_dists.`**`plot_val_scatter`** (plot_dict={}, xlabel='Random Variable', ylabel='Random Variable Value', alpha=1, logscale=False, show_fig=False)

Plots scatter plot.

plot_dict= {'class_1': {'x_values': [0.1, 0.2, 0.3], 'y_values': [20, 40, 80]},

'class_2': {'x_values': [0.1, 0.2, 0.3], 'y_values': [80, 60, 20]}}

`trafpy.generator.src.dists.plot_dists.`**`plot_val_stacked_bar`** (plot_dict={}, ylabel='Random Variable', ylim=None, bar_width=0.35, show_fig=False)

Plots stacked bar chart.

E.g. plot_dict given should be of the form:
plot_dict= {'class_1': {'x_values': ['Uni DCN', 'Private DCN', 'Cloud DCN'], 'y_values': [20, 40, 80]}, 'class_2': {'x_values': ['Uni DCN', 'Private DCN', 'Cloud DCN'], 'y_values': [80, 60, 20]}}
ylim=[0,100]

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=False, 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, 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
- **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 explitly 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** (_beta, size, round_to_nearest=None, num_decimal_places=2, min_val=None, max_val=None, interactive_params=None, logscale=False, transparent=False)

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=False)

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 notmally 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=30000, 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=False)

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=False)

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, 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.
- **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=False, 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, min_val, max_val, num_samples)

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_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=False)

Generates a Weibull distribution of random variable values.

Weibull distributions often fir 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: Paricle 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)

Rounds variable to nearest specified value.

Module contents

Submodules

trafpy.generator.src.builder module

Module for building demand data dictionaries (flow- and job-centric).

`trafpy.generator.src.builder.`**`construct_demand_slots_dict`** (demand_data, slot_size=0.1, include_empty_slots=False, print_info=False)

Takes demand data (job-centric or flow-centric) and generates time-slot demand dictionaries.

Often when simulating networks, it is useful to divide the arriving demands into time slots. This function uses the generated demand data event times and the specified slot size to divide when each demand arrives in the simulation into specific time slots.

Returned dict keys are time slot boundary times and values are any demands which arrive in the time slot.

Parameters:

- **demand_data** (*dict*) – Generated demand data (either flow-centric or job-centric).
- **slot_size** (*float*) – Time period of each time slot. MUST BE FLOAT!!
- **include_empty_slots** (*bool*) – Whether or not to include empty (i.e. no flows arriving) slots in slots_dict values. If True, will have keys for all slots of simulation, but will larger memory usage, making the slots_dict less scalable.

Returns: Dictionary containing the original demand data organised into time slots.

Return type: dict

`trafpy.generator.src.builder.`**`create_demand_data`** (eps, node_dist, flow_size_dist, interarrival_time_dist, num_demands=None, network_load_config=None, duration_time_dist=None, num_ops_dist=None, c=None, use_multiprocessing=True, num_demands_factor=50, min_last_demand_arrival_time=None, print_data=False, path_to_save=None)

Create demand data dictionary using given distributions.

If num_ops_dist and c are left as None, return flow-centric demand data. Otherwise, return job-centric demand data.

Parameters:

- **eps** (*list*) – List of network endpoints.
- **node_dist** (*numpy array*) – 2d matrix of source-destination probabilities of occurring
- **flow_size_dist** (*dict*) – Probability distribution whose key-value pairs are flow size value-probability pairs.
- **interarrival_time_dist** (*dict*) – Probability distribution whose key-value pairs are interarrival time value-probability pairs.
- **num_demands** (*int*) – Number of demands to generate. If None, must specify network_load_config
- **network_load_config** (*dict*) – Dict of form {'network_rate_capacity': <int/float>, 'target_load_fraction': <float>, 'disable_timeouts': <bool>, 'return_new_interarrival_time_dist': <bool>}, where network_rate_capacity is the maximum rate (in e.g. Gbps) at which information can be reliably transmitted over the communication network which the demand data will be inserted into, and where target_load_fraction is the fraction of the network rate capacity being requested by the demands (e.g. target_load_fraction=0.75 would generate demands which request a load that is 75% of the network rate capacity from the first to the last demand arriving). disable_timeouts defines whether or not to stop looping when trying to meet specified network load. return_new_interarrival_time_dist defines whether or not to return the new interarrival time dist which was adjusted to meet the network node requested. If network_load_config is None, must specify num_demands
- **duration_time_dist** (*dict*) – Probability distribution whose key-value pairs are duration time value-probability pairs. If specified, half events returned will be 'take-down' events (establish==0). If left as None, all returned events will be 'connection establishment' events (establish==1).
- **num_ops_dist** (*dict*) – Probability distribution whose key-value pairs are number of operations (in a job) value-probability pairs.
- **c** (*int/float*) – Coefficient which determines job graph connectivity and therefore the number of edges in the job graph. Use this because, for large enough c and n (number of nodes), edge formation probability when using Erdos-Renyi random graph creation scales with the number of edges such that p=c*ln(n)/n, where graph diameter (and number of edges) scales with O(ln(n)). See https://www.cs.cmu.edu/~avrim/598/chap4only.pdf for more information.
- **use_multiprocessing** (*bool*) – Whether or not to use multiprocessing when generating data. For generating large numbers of big job computation graphs, it is recommended to use multiprocessing.
- **num_demands_factor** (*int*) – Factor by which to multipl number of network endpoint pairs by to get the number of demands.
- **min_last_demand_arrival_time** (*int, float*) – Minimum last time of arrival for final demand (helps user specify a minimum simulation time). Will keep doubling number of demands until get >= min_last_demand_arrival_time.
- **print_data** (*bool*) – whether or not to print extra information about the generated data (such as time 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'.

Returns: Generated demand data (either flow-centric or job-centric demanddemand data depending on the args given to the function). N.B. If network_load_config is not None, will return tuple of (demand_data, interarrival_time_dist) to give updated interarrival time dist needed to meet network_load_config requested.

Return type: dict

trafpy.generator.src.demand module

class trafpy.generator.src.demand.**Demand** (demand_data, eps, name='demand')
Bases: **object**

get_num_demands (demand_data)

get_num_deps (demand_data)

get_slots_dict (slot_size, print_info=False)

reset (demand_data)

class trafpy.generator.src.demand.**DemandAnalyser** (demand, bidirectional_links=True, subject_class_name=None)
Bases: **object**

compute_metrics (bidirectional_links=True, print_summary=False)

class trafpy.generator.src.demand.**DemandPlotter** (demand)
Bases: **object**

find_index_of_int_in_str (string)

plot_flow_size_dist (logscale=True, num_bins=20)

plot_interarrival_time_dist (logscale=True, num_bins=20)

plot_link_loads_vs_time (net, slot_size, demand, mean_period=None, logscale=False)

plot_node_dist (eps, logscale=True, num_bins=20)

class trafpy.generator.src.demand.**DemandsAnalyser** (*demands)
Bases: **object**

compute_metrics (print_summary=False)

class trafpy.generator.src.demand.**DemandsPlotter** (*demands)
Bases: **object**

plot_flow_size_dists (logscale=False)

plot_interarrival_time_dists (logscale=False)

trafpy.generator.src.flowcentric module

trafpy.generator.src.flowcentric.**adjust_demand_load** (demand_data, network_load_config, num_demands, eps, node_dist, flow_size_dist, interarrival_time_dist, duration_time_dist, print_data=False)

trafpy.generator.src.flowcentric.**adjust_demand_load_to_ep_link_margin** (demand_data, new_num_demands, interarrival_time_dist, eps, node_dist, flow_size_dist, network_load_config, ep_link_margin=0.95, increment_factor=1.001, print_data=False)
Decrease ep link loads of each ep link until <= ep link margin load (e.g. 0.95). If after this decrease the overall load is below target load, increase lowest ep link loads until get to target load. Therefore as target load tends to 1, node distribution tends towards uniform distribution.

trafpy.generator.src.flowcentric.**create_flow_centric_demand_data** (num_demands, eps, node_dist, flow_size_dist, interarrival_time_dist, duration_time_dist=None, print_data=False)

trafpy.generator.src.flowcentric.**decrease_demand_load_to_target** (demand_data, num_demands, interarrival_time_dist, eps, node_dist, flow_size_dist, network_load_config, increment_factor=1.001, print_data=False)

trafpy.generator.src.flowcentric.**drop_random_flow_from_demand_data** (demand_data)

trafpy.generator.src.flowcentric.**duplicate_demands_in_demand_data_dict** (demand_data, method='all_eps', **kwargs)
If method == 'all_eps', will duplicate all demands by adding final event time over all endpoints to each event time
if method == 'per_ep', will duplicate all demands by adding final even time for each endpoint's final event time

trafpy.generator.src.flowcentric.**find_index_of_int_in_str** (string)

trafpy.generator.src.flowcentric.**get_first_last_flow_arrival_times** (demand_data)

trafpy.generator.src.flowcentric.**get_flow_centric_demand_data_ep_load_rate** (demand_data, ep, eps, method='all_eps')
If method=='all_eps', duration is time_last_flow_arrived-time_first_flow_arrived across all endpoints. If method=='per_ep', duration is time_last_flow_arrived-time_first_flow_arrived for this specific ep.

trafpy.generator.src.flowcentric.**get_flow_centric_demand_data_overall_load_rate** (demand_data, bidirectional_links=True)
If flow connections are bidirectional_links, 1 flow takes up 2 endpoint links (the source link and the destination link), therefore effecitvely takes up load rate 2*flow_size*duration bandwidth. If not bidriectional, only takes up 1*flow_size*duration since only occupies bandwidth for 1 of these links.
If method == 'mean_per_ep', will calculate the total network load as being the mean average load on each endpoint link (i.e. sum info requests for each link -> find load of each link -> find mean of ep link loads)
If method == 'mean_all_eps', will calculate the total network load as being the average load over all endpoint links (i.e. sum info requests for all links -> find overall load of network)

trafpy.generator.src.flowcentric.**get_flow_centric_demand_data_total_info_arrived** (demand_data)

trafpy.generator.src.flowcentric.**group_demand_data_into_ep_info** (demand_data, eps)

trafpy.generator.src.flowcentric.**increase_demand_load_to_target** (demand_data, num_demands, interarrival_time_dist, eps, node_dist, flow_size_dist, network_load_config, increment_factor=0.5, print_data=False)

trafpy.generator.src.interactive module

trafpy.generator.src.jobcentric module

trafpy.generator.src.jobcentric.**allocate_job_flow_attrs** (job, job_idx, job_ids, flow_size_dist, jobs=None)
If doing multi processing i.e. generating multiple jobs in parallel, must give multiprocessing.Manager().list() object as jobs attr of this function so that function can append to multiprocessing manager list. If not doing multiprocessing, leave attr as jobs=None

trafpy.generator.src.jobcentric.**allocate_job_ops_to_machines** (job, eps, node_dist, jobs=None)
If doing multi processing i.e. generating multiple jobs in parallel, must give multiprocessing.Manager().list() object as jobs attr of this function so that function can append to multiprocessing manager list. If not doing multiprocessing, leave attr as jobs=None

trafpy.generator.src.jobcentric.**create_job_centric_demand_data** (num_demands, eps, node_dist, flow_size_dist, interarrival_time_dist, num_ops_dist, duration_time_dist=None, c=0.8, run_time_gaussian_noise_mean=0, run_time_gaussian_noise_sd=0.1, round_op_run_time_to_nearest=1e-05, use_multiprocessing=True, print_data=False)

trafpy.generator.src.jobcentric.**draw_job_graph** (job, node_size=500, font_size=15, linewidths=1, fig_scale=1, draw_labels=True, show_fig=False, directed_graph=True)
Draws single job graph

trafpy.generator.src.jobcentric.**draw_job_graphs** (demand_data=None, job_graphs=[], node_size=500, font_size=15, linewidths=1, fig_scale=1.25, draw_labels=True, show_fig=False, path_to_save=None)
Draws list of specified job graphs. If no job graphs specified, plots all job graphs

trafpy.generator.src.jobcentric.**gen_job_event_dict** (demand_data, event_iter)

trafpy.generator.src.jobcentric.**gen_job_graph** (num_ops, c, prob_data_dependency=0.8, jobs=None, print_data=False)

If doing multi processing i.e. generating multiple jobs in parallel, must give multiprocessing.Manager().list() object as jobs attr of this function so that function can append to multiprocessing manager list. If not doing multiprocessing, leave attr as jobs=None & func will return a single job.

`trafpy.generator.src.jobcentric.`**`gen_job_graphs`** (num_jobs, num_ops_dist, c=1.5, prob_data_dependency=0.8, use_multiprocessing=True, print_data=False)

`trafpy.generator.src.jobcentric.`**`get_job_demand_data_dependency_stats`** (demand_data)
Gets stats of all dependencies of each job in demand_data. Returns these stats as a dict {job_id: dependency_stats}

`trafpy.generator.src.jobcentric.`**`get_job_dependency_stats`** (job)
Get stats of all dependencies in a single job and returns as a list.

`trafpy.generator.src.jobcentric.`**`set_job_op_run_times`** (job, run_time_gaussian_noise_mean, run_time_gaussian_noise_sd, round_op_run_time_to_nearest, jobs=None)
If doing multi processing i.e. generating multiple jobs in parallel, must give multiprocessing.Manager().list() object as jobs attr of this function so that function can append to multiprocessing manager list. If not doing multiprocessing, leave attr as jobs=None

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

Parameters:
- **network** (*networkx graph*) – Network containing edges to whiich 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.

`trafpy.generator.src.networks.`**`add_edges_capacity_attrs`** (network, edges, channel_names, channel_capacity)
Adds channels & max channel capacitys 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’]

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.

`trafpy.generator.src.networks.`**`gen_arbitrary_network`** (ep_label=None, server_to_rack_channel_capacity=12500, num_channels=1, num_eps=10)
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:
- **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_eps** (*int*) – Number of endpoints in network.

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, 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=1, rack_to_edge_channel_capacity=10000000000, edge_to_agg_channel_capacity=40000000000, agg_to_core_channel_capacity=40000000000, show_fig=False)

Generates a data centre network with a 3-layer fat tree topology.
Resource for building: https://blogchinmaya.blogspot.com/2017/04/what-is-fat-tree
Parameters of network:

- number of core switches = (k/2)^2
- number of pods = k
- number of aggregate switches = (k^2)/2
- number of edge switches = (k^2)/2
- number of racks = (k^3)/4
- number of servers = N*(k^3)/4

Parameters:

- **k** (*int*) – Number of ports/links on each switch
- **N** (*int*) – Number of servers 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*) – Byte capacity per channel
- **agg_to_core_channel_capacity** (*int,float*) – 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: **`json.encoder.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.`**`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)

Returns basic params of network.

`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 past 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 module

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: **trafpy.manager.src.schedulers.schedulertoolbox.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: **trafpy.manager.src.schedulers.schedulertoolbox.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)

trafpy.manager.src.schedulers.parametric_agent module

class trafpy.manager.src.schedulers.parametric_agent.**ParametricAgent** (obs_space, action_space, num_outputs, model_config={}, name='agent', true_obs_shape=4, action_embed_size=2, env=None, Graph=None, RWA=None, slot_size=None, **kw)
Bases: **ray.rllib.agents.dqn.distributional_q_tf_model.DistributionalQTFModel**

conv_chosen_action_index_to_chosen_flow (action, chosen_flows=None)

forward (input_dict, state=None, seq_lens=None)
Call the model with the given input tensors and state.
Any complex observations (dicts, tuples, etc.) will be unpacked by __call__ before being passed to forward(). To access the flattened observation tensor, refer to input_dict["obs_flat"].
This method can be called any number of times. In eager execution, each call to forward() will eagerly evaluate the model. In symbolic execution, each call to forward creates a computation graph that operates over the variables of this model (i.e., shares weights).
Custom models should override this instead of __call__.

Parameters:
- **input_dict** (*dict*) – dictionary of input tensors, including "obs", "obs_flat", "prev_action", "prev_reward", "is_training", "eps_id", "agent_id", "infos", and "t".
- **state** (*list*) – list of state tensors with sizes matching those returned by get_initial_state + the batch dimension
- **seq_lens** (*Tensor*) – 1d tensor holding input sequence lengths

Returns: The model output tensor of size[BATCH, num_outputs], and the new RNN state.

Return type: (outputs, state)

Examples

```
>>> def forward(self, input_dict, state, seq_lens):
>>>     model_out, self._value_out = self.base_model(
...         input_dict["obs"])
>>>     return model_out, state
```

get_indices_of_available_actions (action_dict)

register_env (env)

update_avail_actions (*chosen_actions)

trafpy.manager.src.schedulers.random module

class trafpy.manager.src.schedulers.random.**RandomAgent** (Graph, RWA, slot_size, packet_size=300, env=None, scheduler_name='random')

Bases: **trafpy.manager.src.schedulers.schedulertoolbox.SchedulerToolbox**

conv_chosen_action_index_to_chosen_flow (action, chosen_flows=None)

get_action (obs)

get_indices_of_available_actions (action_dict)

get_scheduler_action (obs, choose_multiple_actions=True)

register_env (env)

update_avail_actions (*chosen_actions)

trafpy.manager.src.schedulers.schedulers module

trafpy.manager.src.schedulers.schedulertoolbox module

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

Bases: **object**

binary_encode_num_flows_in_queue (num_flows_in_queue, max_num_flows_in_queue)

binary_encode_paths (paths)

binary_encode_time_in_queue (time_in_queue, max_record_time_in_queue)

calc_basrpt_cost (flow, V, N)

check_if_lightpath_available (path, channel, chosen_flows)

Checks if chosen flow already has edges which have been assigned to channel. If it does and if time_multiplexing, checks if any space left on contentious path channels.

choose_channel_and_path_using_contending_flows (contending_flows)

choose_num_packets_this_time_slot (flow, path, channel)

Given a flow, path and channel, finds the number of packets that this flow can have scheduled this time slot given the channel capacities of the link channels in the path.

estimate_time_to_completion (flow_dict**)**

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_all_contending_flows (chosen_flow**,** chosen_flows**,** cost_metric**,** **kwargs**)**

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
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

find_shortest_flow_in_queue (queued_flows**,** completion_times**)**
Allocates shortest lightpaths and finds flow in queue w/ shortest completion time

gen_flow_packets (flow_size**)**

get_channel_bandwidth (edge**,** channel**)**
Gets current channel bandwidth left on a given edge in the network.

get_curr_queue_states (network**)**
Returns all queues (empty and full) in network

get_flow_from_network (server**,** queue**,** flow_idx**)**
Given the server that the flow is at, the queue of the server that the flow is in, and the flow idx of the flow in the queue, this method returns the flow dict

get_max_flow_info_transferred_per_slot (flow**,** path**,** channel**)**
Returns maximum possible flow information & number of packets transferred per timeslot given the flow's path (i.e. in point-to-point circuit switched network, max info transferred per slot is the bandwidth of the lowest bw link in the path * the slot size)

get_maximum_packets_available_if_all_edges_empty (chosen_flow**,** chosen_path**)**

get_maximum_packets_requestable_by_flow (chosen_flow**,** max_packets_available_if_all_edges_empty**,** packets_available_outside_contending_edges**)**

get_packets_available_if_drop_all_contending_flows (chosen_flow**,** chosen_path**,** chosen_channel**,** contending_flows_list**)**

get_packets_available_outside_contending_edges (chosen_flow**,** chosen_path**,** chosen_channel**,** contending_flows_list**)**

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**)**

look_for_available_lightpath (chosen_flow**,** chosen_flows**,** search_k_shortest=True**)**
If search_k_shortest, will look at all k shortest paths available in flow['k_shortest_paths']. If False, will only consider flow['path'] already assigned.

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.

select_minimum_number_of_contending_flows_to_drop (chosen_flow**,** chosen_path**,** chosen_channel**,** contending_flows_list**,** max_packets_requested_by_chosen_flow**,** max_packets_available_if_all_edges_empty**,** cost_metric**,** **kwargs**)**

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: **trafpy.manager.src.schedulers.schedulertoolbox.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

Module contents

trafpy.manager.src.simulators package

Submodules

trafpy.manager.src.simulators.analysers module

trafpy.manager.src.simulators.dcn module

class trafpy.manager.src.simulators.dcn.**DCN** (Network, Demand, Scheduler, num_k_paths, slot_size, sim_name='dcn_sim', max_flows=None, max_time=None, time_multiplexing=True, track_grid_slot_evolution=False, track_queue_length_evolution=False, track_link_utilisation_evolution=True, track_link_concurrent_demands_evolution=True, gen_machine_readable_network=False)

Bases: **gym.core.Env**

add_flow_to_queue (flow_dict**)**
Adds a new flow to the appropriate src-dst virtual queue in the simulator's network. Also updates arrived flows record

add_flows_to_queues (observation**)**
Takes observation of current time slot and updates virtual queues in network

add_job_to_queue (job_dict**,** print_times=False**)**
Adds a new job with its respective flows to the appropriate src-dst virtual queue in the simulator's network. Aslo updates arrived flows record

calc_num_queued_flows_num_full_queues ()
Calc num queued flows and full queues in network

calc_num_queued_jobs ()
Calc num queued jobs in network.

calc_queue_length (src**,** dst**)**
Calc queue length in bytes at a given src-dst queue

calc_reward ()

check_chosen_flows_valid (chosen_flows**)**

check_flow_present (flow**,** flows**)**
Checks if flow is present 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 check if flow is present

check_if_any_flows_arrived ()

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

check_if_done ()
Checks if all flows (if flow centric) or all jobs (if job centric) have arrived & been completed &/or dropped

check_if_pairs_valid (slots_dict**)**
Since the network and the demand for a simulation are created separately, an easy mistake to fall into is to name the network nodes in the network differently from the src-dst pairs in the demand. This can lead to infinite loops since the flows never get added to appropriate queues! This function loops through all the src-dst pairs in the first slot of the slots dict to try to catch this error before the simulation is ran

check_num_channels_used (graph**,** edge**)**
Checks number of channels currently in use on given edge in graph

conv_fig_to_image (fig**)**
Takes matplotlib figure and converts into numpy array of RGB pixel values

display_env_memory_usage (obs**)**

display_step_processing_time (num_decimals=8**)**

draw_network_state (draw_flows=True**,** draw_ops=True**,** draw_node_labels=False**,** ep_label='server'**,** appended_node_size=300**,** network_node_size=2000**,** appended_node_x_spacing=5**,** appended_node_y_spacing=0.75**,** font_size=15**,** linewidths=1**,** fig_scale=2**)**
Draws network state as matplotlib figure

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

get_channel_bandwidth (edge**,** channel**)**
Gets current channel bandwidth left on a given edge in the network.

get_current_queue_states ()
Returns list of all queues in network

get_max_flow_info_transferred_per_slot (flow_dict**)**
Returns maximum possible flow information & number of packets transferred per timeslot given the flow's path (i.e. in point-to-point circuit switched network, max info transferred per slot is the bandwidth of the lowest bw link in the path * the slot size)

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_grid_slot_evolution (Graph**)**

init_link_concurrent_demands_dict (net**)**

init_link_utilisation_evolution (net**)**

init_queue_evolution (Graph**)**

init_virtual_queues (Graph**)**

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 past 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

next_observation ()
Compiles simulator data and returns observation

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

register_arrived_flow (flow_dict**)**

register_completed_flow (flow_dict**,** print_times=False**)**
Takes a completed flow, appends it to list of completed flows, records time at which it was completed, and removes it from queue. If 'flow' in fact never become a flow (i.e. had src == dst or was control dependency with size == 0), will update dependencies but won't append to completed flows etc.

register_completed_job (job_dict**)**

remove_flow_from_queue (flow_dict**)**

remove_job_from_queue (job_dict**)**

render (action=None**,** dpi=300**,** fig_scale=1**)**
Renders current network state for final animation at end of scheduling session. If action is None, will only render network queue state(s) rather than also rendering selected lightpaths.

render_network (action=None**,** fig_scale=1**)**
Renders network state as matplotlib figure and, if specified, renders chosen action(s) (lightpaths) on top of figure

reset (pickled_demand_path=None**,** return_obs=True**)**
Resets DCN simulation environment

reset_channel_capacities_of_edges (edges**)**
Takes edges and resets their available capacities back to their maximum capacities.

save_rendered_animation (path_animation**,** fps=1**,** bitrate=1800**,** animation_name='anim'**)**

save_sim (path**,** name=None**,** overwrite=False**,** zip_data=True**,** print_times=True**)**
Save self (i.e. object) using pickle

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

step (action**,** print_memory_usage=False**,** print_processing_time=False**)**
Performs an action in the DCN simulation environment, moving simulator to next step

take_action (action**)**

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_completed_flow_job (completed_flow**)**
Updates dependencies of other flows in job of completed flow. Checks if all flows in job of completed flow have been completed. If so, will update stat trackers & remove from tracking list

update_curr_time (slot_dict**)**
Updates current time of simulator using slot dict

update_flow_attrs (chosen_flows**)**

update_flow_packets (flow_dict**)**
Takes flow dict that has been schedueled to be activated for curr time slot and removes corresponding number of packets flow in queue

update_grid_slot_evolution (chosen_flows**)**

update_job_flow_dependencies (completed_flow**)**
Go through flows in job and update any flows that were waiting for completed flow to arrive before being able to be scheduled

update_link_concurrent_demands_evolution (link**,** num_concurrent_demands_to_add=1**)**
Adds num_concurrent_demands_to_add to current number of concurrent demands on a given link.

update_link_utilisation_evolution ()

update_queue_evolution ()

update_running_op_dependencies (observation**)**
Takes observation of current time slot and updates dependencies of any ops that are running

class trafpy.manager.src.simulators.dcn.**RepresentationGenerator** (env)
Bases: **object**

conv_human_readable_flow_to_machine_readable_flow (flow**,** return_onehot_vectors=False**,** dtype=tf.float32**)**

gen_machine_readable_network_observation (network_observation**,** return_onehot_vectors=False**,** dtype=tf.float32**)**
If return_onehot_vectors is False, rather than returning one hot encodings, will return discrete indices of variables. This is useful for gym.spaces.Discrete() observation spaces which automatically one-hot encode Discrete observation space variables.

onehot_encode_endpoints ()

onehot_encode_paths ()

trafpy.manager.src.simulators.env_analyser module

class trafpy.manager.src.simulators.env_analyser.**EnvAnalyser** (env, subject_class_name=None)
Bases: **object**

compute_metrics (measurement_start_time=None**,** measurement_end_time=None**,** bidirectional_links=True**,** 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

bidirectional_links (bool): If True, 1 flow occupies bandwidth of 2 endpoint
links therefore is requesting load of 2*flow_size

trafpy.manager.src.simulators.env_plotter module

class trafpy.manager.src.simulators.env_plotter.**EnvPlotter**
Bases: **object**

class trafpy.manager.src.simulators.env_plotter.**EnvsPlotter**
Bases: **object**

find_index_of_int_in_str (string**)**

plot_99th_percentile_fct_vs_basrpt_v (*analysers**)**

plot_99th_percentile_fct_vs_load (*analysers**)**

plot_average_fct_vs_basrpt_v (*analysers**)**

plot_average_fct_vs_load (*analysers**)**
*analysers (*args): Analyser objects whose metrics you wish to plot.

plot_demand_completion_time_vs_size_for_different_loads (*analysers**)**

Index

D

E

F

G

R

U

X

Python Module Index

s

t