ASQ Documentation

Release 0.0.1

Geraint Palmer

December 14, 2015

Contents

1 Basics **3**
- 1.1 Getting Started . . . 3
- 1.2 The Parameters Dictionary . . . 4
- 1.3 The Output Data . . . 5
- 1.4 Going Deeper . . . 5
- 1.5 The Command Line Tool . . . 6
- 1.6 The Parameters File . . . 7

2 Features **9**
- 2.1 Service Time Distributions . . . 9
- 2.2 Custom Discrete PDFs for Service Times . . . 11
- 2.3 Assign Work Schedules for Servers . . . 12
- 2.4 Dynamic Customer Classes . . . 13
- 2.5 Deadlock Detection Capability . . . 14

3 Examples **15**
- 3.1 Example - Simulate M/M/1 Queue . . . 15

4 Full List of Objects and Attributes **17**

5 Glossary **19**

6 Indices and tables **21**

ASQ is a recursive acronym that stands for ASQ Simulates Queues. This package is a simulation framework for open queueing networks.

Contents:

CHAPTER 1

Basics

Contents:

1.1 Getting Started

Consider the following 2 node queueing network:

This queueing network contains 2 nodes:

- **Node 1 (Bottom)**
  - Poisson arrivals at rate 6.0
  - Exponential service rate 8.5
  - Single server
  - Infinite queueing capacity
  - Probability 0.2 of joining Node 2 after service
- **Node 2 (Top)**
  - Poisson arrivals at rate 2.5
  - Exponential service rate 5.5
  - Single server
  - Maximum queueing capacity of 4
  - Probability 0.1 of joining Node 1 after service

We wish to simulate this system for 1000 time units. This system is defined by the following parameters dictionary:

```
>>> params = {
... 'Arrival_rates': {'Class 0': [6.0, 8.5]},
... 'Number_of_nodes': 2,
... 'detect_deadlock': False,
... 'Simulation_time': 1000,
... 'Number_of_servers': [1, 1],
... 'Queue_capacities': ['Inf', 4],
... 'Number_of_classes': 1,
... 'Service_rates': {'Class 0': [['Exponential', 8.5], ['Exponential', 5.5]]},
```

```
... 'Transition_matrices': {'Class 0': [[0.0, 0.2], [0.1, 0.0]]}
... }
```

Please see *The Parameters Dictionary* for a fuller explaination of this. ASQ can then use this parameters dictionary to run the simulation:

```
>>> import asq
>>> Q = Simulation(params)
>>> Q.simulate_until_max_time()
```

Once this simulation has been run, *The Output Data* can be written to file through:

```
>>> Q.write_records_to_file(<path_to_file>)
```

1.2 The Parameters Dictionary

In order to fully define a queueing network simulation, the following need to be defined:

- Number of nodes (service stations)
- Number of customer classes
- Simulation run time

Every node must have the following defined globally (independent of customer class):

- Number of servers
- Queue capacity

Then, for every node and every class the following must be defined:

- Arrival rates
- Service distribution

And then each customer class requires:

- Transition matrix

A full example of the parameters dictionary for a three node network with two classes of customer is shown below:

```
>>> {'Arrival_rates': {'Class 1': [1.0, 1.8, 7.25],
                       'Class 0': [6.0, 4.5, 2.0]},
...  'Number_of_nodes': 3,
...  'detect_deadlock': False,
...  'Simulation_time': 2500,
...  'Number_of_servers': [2, 1, 1],
...  'Queue_capacities': ['Inf', 'Inf', 10],
...  'Number_of_classes': 2,
...  'Service_rates': {'Class 1': [['Exponential', 8.5], ['Triangular', 0.1, 0.8, 0.95], ['Exponentia
...                    'Class 0': [['Exponential', 7.0], ['Exponential', 5.0], ['Gamma', 0.4, 0.6]]},
...  'Transition_matrices': {'Class 1': [[0.7, 0.05, 0.05], [0.5, 0.1, 0.4], [0.2, 0.2, 0.2]],
...                          'Class 0': [[0.1, 0.6, 0.2], [0.0, 0.5, 0.5], [0.3, 0.1, 0.1]]}}
```

Notice that:

- `Queue_capacities` can be set to `"Inf"`.
- When `Queue_capacities` aren't set to `"Inf"` blocking rules apply. Type I (blocked after service) blocking applies here.

- `Number_of_servers` may be set to *"Inf"* also.
- To obtain no arrivals, set `Arrival_rates` to 0.
- There are many service distributions available, see *Service Time Distributions*.
- The `Transition_matrices` for `Class 0` section represents the following transition matrix:

```
    [[0.1, 0.6, 0.2],
     [0.0, 0.5, 0.5],
     [0.3, 0.1, 0.1]]
```

In this transition matrix the *(i,j)* th element corresponds to the probability of transitioning to node *j* after service at node *i*.

There are numerous other features, please see *Features* for more information.

The keys of this dictionary may also be used as keyword arguments when defining a simulation. ASQ features a function that will load in these parameters from a file, please read *The Parameters File*.

1.3 The Output Data

Once a simulation has been run, the following method may be called to write a data file:

```
>>> Q.write_records_to_file('data/file/location.csv')
```

This file contains does not contain summary statistics, but all the information that happened during the simulation in raw format. Each time an individual completes service at a service station, a data record of that service is kept. This file contains all these data records for all services of all customers at all nodes during the simulation run time.

The following table summarises the columns:

I.D number	Class	Node	Arrival Date	Waiting Time	Service Start Date	Service Time	Service End Date	Time Blocked	Exit Date
227592	1	0	245.601	0.0	245.601	0.563	246.164	0.0	246.164
411239	0	0	245.633	0.531	246.164	0.608	246.772	0.0	246.772
001195	0	2	247.821	0.0	247.841	1.310	249.151	0.882	250.033
...	...	...	...	...	...	...	...	...	...

The `write_records_to_file` method writes a header as default. To disable this feature, input `headers=False`:

```
>>> Q.write_records_to_file(<path_to_file>, header=False)
```

1.4 Going Deeper

In *Getting Started* you saw how to run a simple simulation. This page lets you access the simulation by exploring its attributed and methods. First, set up a parameters file as described in *The Parameters File*.

Now importing ASQ and the parameters file as a dictionary is simple:

```
>>> import asq
>>> params = asq.load_parameters("path/to/parameters/file/")
>>> params["Number_of_servers"]
[2, 1, 1]
```

Set up a Simulation object, from which all parameters can also be accessed:

```
>>> Q = asq.Simulation(params)
>>> Q.number_of_nodes
3
>>> Q.queue_capacities
['Inf', 'Inf', 10]
>>> Q.lmbda    # The arrival rates of the system
[[1.0, 1.8, 7.25], [6.0, 4.5, 2.0]]
>>> Q.lmbda[0]    # Arrival rates of the 0th class
[1.0, 1.8, 7.2]
```

A full list of ASQ's objects and attributes can be found here: *Full List of Objects and Attributes* Now to run a simulation simply run the following method:

```
>>> Q.simulate_until_max_time()
```

Individuals' data records can be accessed directly using the following methods:

```
>>> all_individuals = Q.get_all_individuals()    # Creates a list of all individuals in the system
>>> all_individuals[0]
Individual 13
>>> all_individuals[0].data_records.values()[0][0].wait    # Time Individual 13 was waiting for this
0.39586652218275364
>>> all_individuals[0].data_records.values()[0][0].arrival_date # Time Individual 13 arrived for this
0.5736475797750542
```

The full list data records can be written to a csv file:

```
>>> Q.write_records_to_file(<path_to_file>)
```

Please see *The Output Data* for an explanation of the data contained here.

1.5 The Command Line Tool

This page will describe how to run the experiment via the command line. Set up a new folder alongside ASQ that will contain your parameters file:

```
.
-- my_simulation_instance
|   -- parameters.yml
|
-- ASQ
```

The parameters.yml file is a yaml file containing all the parameters that describe the queueing network you would like to simulate. See *The Parameters File* for how to set up this file.

To run the simulation go to the directory which contains both `ASQ` and `my_simultion_instance`. Then run the following command:

```
$ python ASQ/scripts/run_simulation.py my_simulation_instance/
```

This will create a `data.csv`, positioned here:

```
.
-- my_simulation_instance
|   -- parameters.yml
|   -- data.csv
-- ASQ
```

Please see *The Output Data* for an explanation of the data contained here.

1.6 The Parameters File

ASQ features a `load_parameters` function that imports a parameters file as a dictionary. This file take is in `.yml` format. A `parameters.yml` file of this same format is required for use with *The Command Line Tool*.

A full example of the parameters file for a three node network with two classes of customer is shown below:

```
Arrival_rates:
  Class 0:
  - 6.0
  - 4.5
  - 2.0
  Class 1:
  - 1.0
  - 1.8
  - 7.25
detect_deadlock: False
Number_of_classes: 2
Number_of_nodes: 3
Number_of_servers:
- 2
- 1
- 1
Queue_capacities:
- "Inf"
- "Inf"
- 10
Service_rates:
  Class 0:
  - - Exponential
    - 7.0
  - - Exponential
    - 5.0
  - - Gamma
    - 0.4
    - 0.6
  Class 1:
  - - Exponential
    - 8.5
  - - Triangular
    - 0.1
    - 0.8
    - 0.95
  - - Exponential
    - 3.0
Simulation_time: 2500
Transition_matrices:
  Class 0:
  - - 0.1
    - 0.6
    - 0.2
  - - 0.0
    - 0.5
    - 0.5
  - - 0.3
```

```
    - 0.1
    - 0.1
  Class 1:
- - 0.7
    - 0.05
    - 0.05
- - 0.5
    - 0.1
    - 0.4
- - 0.2
    - 0.2
    - 0.2
```

The variable names are identical to the keys of the parameters dictionary (*The Parameters Dictionary*), and the keyword arguments that may also be used.

CHAPTER 2

Features

Contents:

2.1 Service Time Distributions

ASQ currently allows the following continuous service rate distributions:

- *The Uniform Distribution*
- *The Deterministic Distribution*
- *The Triangular Distribution*
- *The Exponential Distribution*
- *The Gamma Distribution*
- *The Lognormal Distribution*
- *The Weibull Distribution*

See *Custom Discrete PDFs for Service Times* for how to define custom discrete service time distributions. Note that when choosing parameters for these distributions, ensure that no negative numbers may be sampled.

2.1.1 The Uniform Distribution

The uniform distribution samples a random number between two numbers a and b. In the `parameters.yml` file, write a uniform distribution between *4* and *9* as follows:

```
- - Uniform
  - 4
  - 9
```

2.1.2 The Deterministic Distribution

The deterministic distribution is non-stachastic, and produces the same service time repeatedly. In the `parameters.yml` file, write a deterministic distribution that repeatedly gives a value of *18.2* as follows:

```
- - Deterministic
  - 18.2
```

2.1.3 The Triangular Distribution

The triangular distribution samples a continuous pdf that rises linearly from its minimum value *low* to its mode value *mode*, and then decreases linearly to its highest attainable value *high*. In the `parameters.yml` file, write a triangular distribution between *2.1* and *7.6* with mode of ‘3.4 as follows:

```
- - Triangular
  - 2.1
  - 7.6
  - 3.4
```

2.1.4 The Exponential Distribution

The exponential distribution samples a random number from the negative exponential distribution with *1 / lambda*. In the `parameters.yml` file, write an exponential distribution with mean *0.2* as follows:

```
- - Exponential
  - 5
```

2.1.5 The Gamma Distribution

The gamma distribution samples a random number from the gamma distribution with shape parameter *alpha* and scale parameter *beta*. In the `parameters.yml` file, write a gamma distribution with parameters *alpha = 0.6* and *beta = 1.2* as follows:

```
- - Gamma
  - 0.6
  - 1.2
```

2.1.6 The Lognormal Distribution

The lognormal distribution samples a random number from the log of the normal distribution with mean *mu* and standard deviation *sigma*. In the `parameters.yml` file, write a lognomal distribution of the normal distribution with mean *4.5* and standard deviation *2.0* as follows:

```
- - Lognormal
  - 4.5
  - 2.0
```

2.1.7 The Weibull Distribution

The Weibull distribution samples a random number from the Weibull distribution with scale parameter *alpha* and shape parameter *beta*. In the `parameters.yml` file, write a Weibull distribution with *alpha = 0.9* and *beta = 0.8* as follows:

```
- - Weibull
  - 0.9
  - 0.8
```

2.2 Custom Discrete PDFs for Service Times

ASQ allows users to define their own discrete service time distributions. An example distribution may look like this:

P(X)	0.1	0.1	0.3	0.2	0.2	0.1
X	9.5	10.2	10.6	10.9	11.7	12.1

In order to define this probability density function, it must be given a name. Let's call it `my_special_distribution_01`.

In order to implement this in the parameters dictionary, simply state that that class and node's service distribution is `Custom`, and the name the distribution as a parameter:

```
'Service_rates':{'Class 0':[['Custom', 'my_special_distribution_01'], ['Exponential', 0.1]], 'Class 1
```

In the `parameters.yml` file, under `Serivce_rates`, for the given class and node enter `Custom` and the name of the distribution below it. An example is shown:

```
Service_rates:
  Class 0:
  - - Custom
    - my_special_distribution_01
  - - Exponential
    - 0.1
  Class 1:
  - - Exponential
    - 0.3
  - - Exponential
    - 0.1
```

This tells ASQ that at Node 1 all Class 0 customers will have their service time drawn from the custom distribution `my_special_distribution_01`. This distribution hasn't been defined yet.

To define the distribution, add the following to your parameters dictionary:

```
'my_special_distribution':[[0.1, 9.5], [0.1, 10.2], [0.3, 10.6], [0.2, 10.9], [0.2, 11.7], [0.2, 12.1
```

To define the distribution in the `parameters.yml` file, add the following lines to the end:

```
my_special_distribution_01:
  - - 0.1
    - 9.5
  - - 0.1
    - 10.2
  - - 0.3
    - 10.6
  - - 0.2
    - 10.9
  - - 0.2
    - 11.7
  - - 0.1
    - 12.1
```

Here we are saying that the value 9.5 will be sampled with probability 0.1, the value 10.2 will be samples with probability 0.1, etc. This fully defines the custom discrete PDF.

Note:

- For each distribution, probabilities must sum to 1.
- You may add as many custom probability distributions as you like.

- **Service times** are sampled from the custom probability distribution, even though they are placed under the heading `Service_rates`.

2.3 Assign Work Schedules for Servers

ASQ allows users to assign cyclic work schedules to servers at each service centre. An example cyclic work schedule may look like this:

Shift Times	0-40	40-100	100-120	120-180	180-220	220-250
Number of Servers	2	3	1	2	4	0

This schedule is cyclic, therefore after the last shift (220-250), schedule begins again with the shift (0-40). The cycle length for this schedule is 250.

In order to define this work schedule, it must be given a name. Let's call it `my_special_schedule_01`.

In the `parameters.yml` file, under `Number_of_servers`, for the given node enter the name of the schedule. The `cycle_length` must also be given. An example is shown:

```
cycle_length: 250
Number_of_servers:
  - 'my_special_schedule_01'
  - 3
```

The equivalent way to add this to the parameters dictionary is by first adding the cycle length:

```
'cycle_length':250
```

And then under number of servers, add the schedule name:

```
'Number_of_servers':['my_special_schedule_01', 3]
```

This tells ASQ that at Node 1 the number of servers will vary over time according to the work schedule `my_special_schedule_01`. This schedule hasn't been defined yet. To define the work schedule, add the following lines to the end of the `parameters.yml` file:

```
my_special_schedule_01:
  - - 0
    - 2
  - - 40
    - 3
  - - 100
    - 1
  - - 120
    - 2
  - - 180
    - 4
  - - 220
    - 0
```

And equivalently, adding the following to the parameters dictionary:

```
'my_special_schedule_01':[[0, 2], [40, 3], [100, 1], [120, 2], [180, 4], [220, 0]]
```

Here we are saying that there will be 2 servers scheduled between times 0 and 40, 3 between 40 and 100, etc. The final shift denotes 0 servers between times 220 and `cycle_length`, and then the schedule cycles to the beginning. This fully defines the cyclic work schedule.

Note:

- If more than one work schedule is defined, the same `cycle_length` must be used for the entire system.

2.4 Dynamic Customer Classes

ASQ allows customers to probabilistically change their class after service. That is after service at node *k* a customer of class *i* will become class *j* with probability *P(J=j | I=i, K=k)*. These probabilities are input into the system through the `Class_change_matrices`.

Below is an example of a class change matrix for a given node:

Node 1		
0.3	0.4	0.3
0.1	0.9	0.0
0.5	0.1	0.4

Node 2		
1.0	0.0	0.0
0.4	0.5	0.1
0.2	0.2	0.6

In this example a customer of class 0 finishing service at node 1 will become class 0 again with probability 0.3, will become class 1 with probability 0.4, and will become class 2 with probability 0.3. A different matrix is given for customers finishing service at node 2.

This is input into the simulation model by including the following to the parameters dictionary:

```
'Class_change_matrices':{'Node 0':[[0.3, 0.4, 0.3], [0.1, 0.9, 0.0], [0.5, 0.1, 0.4]], 'Node 1':[[1.0
```

This is equivalent to adding the following code to the parameters file:

```
Class_change_matrices:
  Node 0:
  - - 0.3
    - 0.4
    - 0.3
  - - 0.1
    - 0.9
    - 0.0
  - - 0.5
    - 0.1
    - 0.4
  Node 1:
  - - 1.0
    - 0.0
    - 0.0
  - - 0.4
    - 0.5
    - 1.0
  - - 0.2
    - 0.2
    - 0.6
```

Omitting this code will simply assume that customers cannot change class after service.

2.5 Deadlock Detection Capability

ASQs has built in deadlock detection capability. With ASQ, a queueing network can be simulated until it reaches deadlock. ASQ then records the time until deadlock from each state.

In order to take advantage of this feature, set deadlock detection option to True in the parameters file:

```
detect_deadlock: False
```

Then use the `simulate_until_deadlock` method to return the times to deadlock from each state:

```
>>> import asq
>>> Q = asq.Simulation(deadlock_params)
>>> times = Q.simulate_until_deadlock()
```

where `times` is a dictionary with states as keys and times to deadlock as values. Note that `Simulation_time` is ingnored in this case.

2.5.1 Example - Deadlock

Consider the M/M/1/3 queue where customers have probability 0.5 of rejoining the queue after service. If the queue is full then that customer gets blocked, and hence the system deadlocks.

Parameters:

```
>>> params = {'Arrival_rates': {'Class 0': [6.0]},
...           'Number_of_nodes': 1,
...           'detect_deadlock': True,
...           'Simulation_time': 2500,
...           'Number_of_servers': [1],
...           'Queue_capacities': [3],
...           'Number_of_classes': 1,
...           'Service_rates': {'Class 0': [['Exponential', 5.0]]},
...           'Transition_matrices': {'Class 0': [[0.5]]}}
```

Running until deadlock:

```
>>> import asq
>>> seed(99)
>>> Q = asq.Simulation(params)
>>> times = Q.simulate_until_deadlock()
>>> times
{((1, 0),): 1.0845416939916719, ((3, 0),): 0.5436399978272065, ((0, 0),): 1.1707879982560288, ((4, 0)
```

Here the state `((i, j),)` denotes the state where there are i customers at the node, j of which are blocked.

CHAPTER 3

Examples

Contents:

3.1 Example - Simulate M/M/1 Queue

Here, an example of an M/M/1 queue will be given, and results compared to to those obtained using standard queueing theory. This will walk through an example of an M/M/1 queue with Poisson arrivals of rate 3 and Exponential service times of rate 5.

Standard queueing theory gives the expected wait in an M/M/1 queue as $E[W] = \frac{\rho}{\mu(1-\rho)}$. With arrival rate $\lambda = 3$ and service rate $\mu = 5$, we get traffic intensity $\rho = \frac{\lambda}{\mu} = 0.6$, and so $E[W] = 0.3$.

We set up the parameters in ASQ:

```
>>> params_dict = {'Queue_capacities': ['Inf'],
...                'Number_of_classes': 1,
...                'Arrival_rates': {'Class 0': [3.0]},
...                'Number_of_nodes': 1,
...                'Service_rates':{'Class 0': [['Exponential', 5.0]]},
...                'Simulation_time': 250,
...                'Transition_matrices': {'Class 0': [[0.0]]},
...                'detect_deadlock': False,
...                'Number_of_servers': [1]}
```

The following code repeats the experiment 100 times, only recording waits for those that arrived after a warm-up time of 50. It then returns the average wait in the system:

```
>>> def interation(warmup):
...     Q = asq.Simulation(params_dict)
...     Q.simulate_until_max_time()
...     inds = Q.get_all_individuals()
...     waits = [ind.data_records[1][0].wait for ind in inds if ind.data_records[1][0].arrival_date >
...     return sum(waits)/len(waits)

>>> seed(27)
>>> ws = []
>>> for i in range(100):
...     ws.append(interation(50))

>>> print sum(ws)/len(ws)
0.292014274888
```

We see that the results of the simulation are in agreement with those of standard queueing theory.

CHAPTER 4

Full List of Objects and Attributes

CHAPTER 5

Glossary

CHAPTER 6

Indices and tables

- genindex
- modindex
- search