Ciw Documentation

Release 1.1.6

Geraint Palmer

Oct 22, 2018

Contents

Etymology: Ciw *noun* Welsh word for a queue. *pl.* ciwiau.

Ciw is a discrete event simulation library for open queueing networks. Its core features include the capability to simulate networks of queues, multiple customer classes, and implementation of Type I blocking for restricted networks. A number of other features are also implemented, including priorities, baulking, schedules, and deadlock detection.

Please note that Ciw is currently supported for and regularly tested on Python versions 2.7, 3.4, 3.5 and 3.6. (However, for the documentation we assume Python 3 is used.)

Contents:

CHAPTER 1

Installing Ciw

The simplest way to install Ciw is:

```
$ pip install ciw
```

However, if you would like to install it from source:

```
$ git clone https://github.com/CiwPython/Ciw.git
$ cd Ciw
$ pip install -r requirements.txt
$ python setup.py install
```

CHAPTER 2

Tutorials - Part 1

Part 1 of the Ciw tutorial will cover what is simulation and why we simulate.

Contents:

2.1 Tutorial I: Defining & Running a Simulation

Assume you are a bank manager and would like to know how long customers wait in your bank. Customers arrive randomly, roughly 12 per hour, regardless of the time of day. Service time is random, but on average lasts roughly 10 minutes. The bank is open 24 hours a day, 7 days a week, and has three servers who are always on duty. If all servers are busy, newly arriving customers form a queue and wait for service. On average how long do customers wait?

We can use computer simulation to find out. Here we will simulate this system, that is make the computer pretend customers arrive and get served at the bank. We can then look at all the virtual customers that passed through the bank, and gain an understanding of how that system would behave if it existed in real life.

This is where Ciw comes in. Let's import Ciw:

```
>>> import ciw
```

Now we need to tell Ciw what our system looks like and how it behaves. We do this by creating a Network object. It takes in keywords containing the following information about the system:

- **Number of servers (`Number_of_servers`)**
  - How many servers are on duty at the bank.
  - In this case, 3 servers.
- **Distribution of inter-arrival times (`Arrival_distributions`)**
  - The distribution of times between arrivals.
  - In this case 12 per hour would mean an average of 5 mins between arrivals.
- **Distribution of service times (`Service_distributions`)**

  - The distribution of times spent in service with a server.
  - In this case an average of 10 mins.

For our bank system, create the Network:

```
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 0.2]],
...     Service_distributions=[['Exponential', 0.1]],
...     Number_of_servers=[3]
... )
```

This fully defines our bank. Notice the distributions; `'Exponential'` here means the inter-arrival and service times are derived from an exponential distribution. The parameters `0.2` and `0.1` imply an average of 5 and 10 time units respectively. Therefore this Network object defines our system in minutes.

First we will *set a seed*. This is *good practice*, and also ensures your results and our results are identical.

```
>>> ciw.seed(1)
```

Now we can create a `Simulation` object. This is the engine that will run the simulation:

```
>>> Q = ciw.Simulation(N)
```

Let's run the simulation for a day of bank time (don't worry, this won't take a day of real time!). As we parametrised the system with time units of minutes, a day will be 1440 time units:

```
>>> Q.simulate_until_max_time(1440)
```

Well done! We've now defined and simulated the bank for a day. In the next tutorial, we will explore the `Simulation` object some more.

2.2 Tutorial II: Exploring the Simulation Object

In the previous tutorial, we defined and simulated our bank for a week:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 0.2]],
...     Service_distributions=[['Exponential', 0.1]],
...     Number_of_servers=[3]
... )
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(1440)
```

Let's explore the Simulation object `Q`. Although out queueing system consisted of one node (the bank), the object `Q` is made up of three, accessed using:

```
>>> Q.nodes
[Arrival Node, Node 1, Exit Node]
```

- **The Arrival Node:** This is where customers are created. They are spawned here, and can *baulk*, *be rejected* or sent to a service node. Accessed using:

```
>>> Q.nodes[0]
Arrival Node
```

- **Service Node:** This is where customers queue up and receive service. This can be thought of as the bank itself. Accessed using:

```
>>> Q.nodes[1]
Node 1
```

- **The Exit Node:** When customers leave the system, they are collected here. Then, when we wish to find out what happened during the simulation run, we can find the customers here. Accessed using:

```
>>> Q.nodes[-1]
Exit Node
```

Once the simulation is run, the simulation object remains in exactly the same state as it reached at the end of the simulation run. Therefore, the simulation object itself can give some information about what went on during the run. The `Exit Node` contains all customers who had completed service in the bank, in order of when they left the system:

```
>>> Q.nodes[-1].all_individuals
[Individual 2, Individual 3, Individual 5, ..., Individual 272]
```

The service node will also contain customers, those who were still waiting or still receiving service at the time the simulation run ended. During this run, there was one customer left in the bank at the end of the day:

```
>>> Q.nodes[1].all_individuals
[Individual 274]
```

Let's look at the first individual to finish service, `Individual 2`. Individuals carry data records, that contain information such as arrival date, waiting time, and exit date:

```
>>> ind = Q.nodes[-1].all_individuals[0]
>>> ind
Individual 2
>>> len(ind.data_records)
1

>>> ind.data_records[0].arrival_date
7.936299...
>>> ind.data_records[0].waiting_time
0.0
>>> ind.data_records[0].service_start_date
7.936299...
>>> ind.data_records[0].service_time
2.944637...
>>> ind.data_records[0].service_end_date
10.88093...
>>> ind.data_records[0].exit_date
10.88093...
```

This isn't a very efficient way to look at results. In the next tutorial we will look at generating lists of records to gain some summary statistics.

2.3 Tutorial III: Collecting Results

In the previous tutorials, we defined and simulated our bank for a week, and saw how to access parts of the simulation engine:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 0.2]],
...     Service_distributions=[['Exponential', 0.1]],
...     Number_of_servers=[3]
... )
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(1440)
```

We can quickly get a list of all data records collected by all customers have have finished at least one service, using the `get_all_records` method of the Simulation object:

```
>>> recs = Q.get_all_records()
```

This returns a list of named tuples. Each named tuple contains the following information:

- `id_number`
- `customer_class`
- `node`
- `arrival_date`
- `waiting_time`
- `service_start_date`
- `service_time`
- `service_end_date`
- `time_blocked`
- `exit_date`
- `destination`
- `queue_size_at_arrival`
- `queue_size_at_departure`

More information on each of these is given in *List of Available Results*.

Using list comprehension, we can get lists on whichever statistic we like:

```
>>> # A list of service times
>>> servicetimes = [r.service_time for r in recs]
>>> servicetimes
[2.94463..., 5.96912..., 0.28757..., ..., 10.46244...]

>>> # A list of waits
>>> waits = [r.waiting_time for r in recs]
>>> waits
[0.0, 0.0, 0.20439..., ..., 5.63781...]
```

Now we can get summary statistics simply by manipulating these lists:

```
>>> mean_service_time = sum(servicetimes) / len(servicetimes)
>>> mean_service_time
9.543928...
```

(continues on next page)

(continued from previous page)

```
>>> mean_waiting_time = sum(waits) / len(waits)
>>> mean_waiting_time
1.688541...
```

We now know the mean waiting time of the customers! In next tutorial we will show how to get more representative results (as we have only simulated one given day here).

Further summary statistics can be obtained using external libraries. We recommend numpy, pandas and matplotlib. Using these further statistics and plots can be explored. The histogram of waits below was created using matplotlib, using the following code:

```
>>> import matplotlib.pyplot as plt
>>> plt.hist(waits);
```

If we'd like to see how busy or idle the servers have been throughout the simulation run, we can look at the `server_utilisation` of a Node. This is the average utilisation of each server, which is the amount of time a server was busy (with a customer), divided by the total amount of time the server was on duty:

```
>>> Q.transitive_nodes[0].server_utilisation
0.60517...
```

Thus in our bank, on average the servers were busy 60.5% of the time.

The next tutorial will show how to use Ciw to get trustworthy results, and finally find out the average waiting time at the bank.

2.4 Tutorial IV: Trials, Warm-up & Cool-down

In Tutorials I-III we investigated one run of a simulation of a bank. Before we draw any conclusions about the behaviour of the bank, there are three things we should consider:

1. Our original description of the bank described it as open 24/7. This means the bank would never start from an empty state, as our simulation does.
2. Those customers left inside the system at the end of the run: their records were not collected, despite spending time in the system during the observation period.
3. Does that single run of our simulation really reflect reality? Were our results simply a fluke? An extreme case?

These three concerns can be addressed with warm-up, cool-down, and trials respectively. A full explanation of these can be found *here*.

1. **Warm-up:** Simulate the system for some time before the beginning of the observation period, such that the system is non-empty and 'in the swing of it' by the time the observation period begins. Only collect results from the beginning of the observation period.
2. **Cool-down:** Simulate the system for some time after the end of the observation period, such that no concerned customers are stuck in the simulation when results collection happens. Only collect results until the end of the observation period.
3. **Trials:** Simulate the system many times with different random number streams (different *seeds*). Keep all interested results from all trials, so that we may take averages and confidence intervals.

Let's define our bank by creating our Network object:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 0.2]],
...     Service_distributions=[['Exponential', 0.1]],
...     Number_of_servers=[3]
... )
```

For simplicity, we will be concerned with finding the mean waiting time only. We'll run 10 simulations in a loop, and take a warm-up time and a cool-down time of 100 time units. Therefore each trial we will run for 1 day + 200 minutes (1640 minutes):

```
>>> average_waits = []
>>> for trial in range(10):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(1640)
...     recs = Q.get_all_records()
...     waits = [r.waiting_time for r in recs if r.arrival_date > 100 and r.arrival_
↪date < 1540]
...     mean_wait = sum(waits) / len(waits)
...     average_waits.append(mean_wait)
```

The list `average_waits` will now contain ten numbers, the mean waiting time from each of the trials. Notice that we set a different seed every time, so each trial will yield different results:

```
>>> average_waits
[3.23439..., 1.67172..., 3.81397..., 2.49197..., 4.53303..., 5.08311..., 3.64789...,␣
↪7.46596..., 7.12032..., 3.28304...]
```

We can see that the range of waits are quite high, between 1.6 and 7.5. This shows that running a single trial wouldn't have given us a very confident answer. We can take the mean result over the trials to get a more confident answer:

```
>>> sum(average_waits) / len(average_waits)
4.23454...
```

Using Ciw, we have found that on average customers wait 4.23 minutes in the queue at the bank. What would happen if we added an extra server? Let's repeat the analysis with 4 servers:

```
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 0.2]],
...     Service_distributions=[['Exponential', 0.1]],
...     Number_of_servers=[4]
... )

>>> average_waits = []
>>> for trial in range(10):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(1640)
...     recs = Q.get_all_records()
...     waits = [r.waiting_time for r in recs if r.arrival_date > 100 and r.arrival_
↪date < 1540]
...     mean_wait = sum(waits) / len(waits)
...     average_waits.append(mean_wait)

>>> sum(average_waits) / len(average_waits)
0.87878...
```

By adding an extra server, we can reduce waits from an average 4.23 minutes to 0.88 minutes! Well done, you have just run your first what-if scenario! What-if scenarios allow us to use simulation to see if expensive changes to the system are beneficial, without actually implementing those expensive changes.

CHAPTER 3

Tutorials - Part 2

Part 2 of the Ciw tutorial will cover some of the other core features of the library. It covers networks of queues, the transition matrix, blocking, and multiple classes of customer.

Contents:

3.1 Tutorial V: A Network of Queues

Ciw's real power comes when modelling networks of queues. That is many service nodes, such that when customers finish service, there is a probability of joining another node, rejoining the current node, or leaving the system.

Imagine a café that sells both hot and cold food. Customers arrive and can take a few routes:

- Customers only wanting cold food must queue at the cold food counter, and then take their food to the till to pay.
- Customers only wanting hot food must queue at the hot food counter, and then take their food to the till to pay.
- Customers wanting both hot and cold food must first queue for cold food, then hot food, and then take both to the till and pay.

In this system there are three nodes: Cold food counter (Node 1), Hot food counter (Node 2), and the till (Node 3):

- Customers wanting hot food only arrive at a rate of 12 per hour.
- Customers wanting cold food arrive at a rate of 18 per hour.
- 30% of all customer who buy cold food also want to buy hot food.
- On average it takes 1 minute to be served cold food, 2 and a half minutes to be served hot food, and 2 minutes to pay.
- There is 1 server at the cold food counter, 2 servers at the hot food counter, and 2 servers at the till.

A diagram of the system is shown below:

This system can be described in one Network object. Arrival and Service distributions are listed in the order of the nodes. So are number of servers. We do however require a *transition matrix*.

A transition matrix is an $n \times n$ matrix (where n is the number of nodes in the network) such that the (i, j)th element corresponds to the probability of transitioning to node j after service at node i. In Python, we write this matrix as a list of lists. The transition matrix for the café system looks like this:

$$\begin{pmatrix} 0.0 & 0.3 & 0.7 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

That is 30% of cold food customers then go to hot food, while the remaining 70% go to the till, and 100% of hot food customers go to the till. This is included when creating a network, with the keyword `Transition_matrices`. So, our Network for the café looks like this:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 0.3],
...                            ['Exponential', 0.2],
...                            'NoArrivals'],
...     Service_distributions=[['Exponential', 1.0],
...                            ['Exponential', 0.4],
...                            ['Exponential', 0.5]],
...     Transition_matrices=[[0.0, 0.3, 0.7],
...                          [0.0, 0.0, 1.0],
...                          [0.0, 0.0, 0.0]],
...     Number_of_servers=[1, 2, 2]
... )
```

Notice the Arrival distributions: 18 cold food arrivals per hour is equivalent to `0.3` per minute; 12 hot food arrivals per hour is equivalent to `0.2` per minute; and we want no arrivals to occur at the Till.

Notice the Service distributions: an average cold food service time of 1 minute is equivalent to a rate of 1/1 = `1` service per minute; an average hot food service time of 2.5 minutes is equivalent to 1/2.5 = `0.4` services per minute; and an average till service time of 2 minutes is equivalent to `0.5` services per minute.

Let's simulate this for one shift of lunchtime of 3 hours (180 mins). At the beginning of lunchtime the café opens, and thus begins from an empty system. Therefore no warm-up time is required. We'll use 20 minutes of cool-down time. We'll run 10 trials, to get a measure of the average number of customers that pass through the system. To find the average number of customers that pass through the system, we can count the number of data records that have passed through Node 3 (the Till):

```
>>> completed_custs = []
>>> for trial in range(10):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(200)
...     recs = Q.get_all_records()
...     num_completed = len([r for r in recs if r.node==3 and r.arrival_date < 180])
...     completed_custs.append(num_completed)
```

We can now get the average number of customers that have passed through the system:

```
>>> sum(completed_custs) / len(completed_custs)
83.0
```

So we've now used Ciw to find out that this café can serve an average 83 customers in a three hour lunchtime.

3.2 Tutorial VI: Restricted Networks

Imagine a manufacturing plant that produces stools:

- Every 4 seconds a seat arrives on a conveyor-belt.
- The belt contains three workstations.
- At the each workstation a leg is connected.
- Connecting a leg takes a random amount of time between 3 seconds and 5 seconds.
- Between workstations (and before the first workstation) the conveyor-belt is only long enough to hold 3 stools.
- If the belt before the first workstation is full then new stools fall to the floor and break.
- If a stool finishes 'service' at a workstation, but there is no space on the conveyor-belt, that stool must remain at the workstation until room becomes available on the conveyor-belt. While this blockage happens, that workstation cannot begin assembling any more stools. (Full details on blocking available *here*.)

Each broken stool costs the factory 10p in wasted wood. We wish to know how many stools will fall to the floor and break per hour of operation, and thus the average cost per hour. First let's define the Network. A restricted network such as this is represented by nearly the same Network object as an unrestricted network, but we now include the keyword `Queue_capacities`:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Deterministic', 4.0],
...                            'NoArrivals',
...                            'NoArrivals'],
...     Service_distributions=[['Uniform', 3, 5],
...                            ['Uniform', 3, 5],
...                            ['Uniform', 3, 5]],
...     Transition_matrices=[[0.0, 1.0, 0.0],
...                          [0.0, 0.0, 1.0],
...                          [0.0, 0.0, 0.0]],
...     Number_of_servers=[1, 1, 1],
...     Queue_capacities=[3, 3, 3]
... )
```

The time taken to attach a leg to the stool (service time) is sampled using the uniform distribution. This samples values equally likely between an upper and lower limit. Note the time units here are in seconds.

If we simulate this, we have access to information about the blockages, for example the amount of time a stool was spent blocked at each node. To illustrate, let's simulate for 20 minutes:

```
>>> ciw.seed(2)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(1200)
>>> recs = Q.get_all_records()

>>> blockages = [r.time_blocked for r in recs]
>>> max(blockages)
1.503404...
```

Here we see that in 20 minutes the maximum time a stool was blocked at a workstation for was 1.5 seconds.

We can get information about the stools that fell off the conveyor-belt using the Simulation's `rejection_dict` attribute. This is a dictionary, that maps node numbers to dictionaries. These dictionaries map customer class numbers to a list of dates at which customers where rejected:

```
>>> Q.rejection_dict
{1: {0: [740.0, 960.0, 1140.0]}, 2: {0: []}, 3: {0: []}}
```

In this run 3 stools were rejected (fell to the floor as there was no room on the conveyor-belt) at Node 1, at times 740, 960, and 1140. To get the number of stools rejected, take the length of this list:

```
>>> len(Q.rejection_dict[1][0])
3
```

Now we'll run 8 trials, and get the average number of rejections in an hour. We will take a warm-up time of 10 minutes. A cool-down will be unnecessary as we are recording rejections, which happen at the time of arrival:

```
>>> broken_stools = []
>>> for trial in range(8):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(4200)
...     num_broken = len([r for r in Q.rejection_dict[1][0] if r > 600])
...     broken_stools.append(num_broken)

>>> broken_stools
[7, 4, 7, 8, 5, 8, 9, 6]

>>> sum(broken_stools)/len(broken_stools)
6.75
```

On average the system gets 6.75 broken stools per hour; costing and average of 67.5p per hour of operation.

A new stool assembly system, costing £2500, can reduce the variance in the leg assembly time, such that it takes between 3.5 and 4.5 seconds to attach a leg. How many hours of operation will the manufacturing plant need to run for so that the new system has saved the plant as much money as it costed?

First, under the new system how many broken stools per hour do we expect?:

```
>>> N = ciw.create_network(
...     Arrival_distributions=[['Deterministic', 4.0],
...                            'NoArrivals',
...                            'NoArrivals'],
...     Service_distributions=[['Uniform', 3.5, 4.5],
...                            ['Uniform', 3.5, 4.5],
...                            ['Uniform', 3.5, 4.5]],
...     Transition_matrices=[[0.0, 1.0, 0.0],
...                          [0.0, 0.0, 1.0],
...                          [0.0, 0.0, 0.0]],
...     Number_of_servers=[1, 1, 1],
...     Queue_capacities=[3, 3, 3]
... )

>>> broken_stools = []
>>> for trial in range(8):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(4200)
...     num_broken = len([r for r in Q.rejection_dict[1][0] if r > 600])
...     broken_stools.append(num_broken)

>>> sum(broken_stools) / len(broken_stools)
0.875
```

Thus the new system saves an average of 5.875 stools per hour, around 58.75p per hour. Therefore it would take $2500/0.5875 \approx 4255.32$ hours of operation for the system to begin paying off.

3.3 Tutorial VII: Multiple Classes of Customer

Imagine a 24 hour paediatricians clinic:

- Two types of patient arrive, babies and children.
- When patients arrive they must register at the reception desk.
- Registration time is random, but for babies lasts an average of 15 minutes, and for children and average of 10 minutes.
- After registration the two types of patients go to separate waiting rooms where they wait to be seen by separate specialists.
- Appointments with specialists take a random amount of time, but on average last one hour.
- There is one receptionist on duty, two baby specialists on duty, and three children's specialists on duty.
- Babies arrive randomly at a rate of one per hour, children at a rate two per hour.

In this set-up we have a scenario where two different types of customer are accessing the same resources, but may use them in different ways. Ciw handles this by assigning **customer classes** to customers. In this set-up:

- Babies are assigned customer `'Class 0'`.
- Children are assigned customer `'Class 1'`.
- The receptionist's desk is Node 1.
- The baby specialist clinic is Node 2.
- The children's specialist clinic is Node 3.

We assign different behaviour for different customer classes by replacing the values of the keywords of the Network object with dictionaries, with customer classes as keys and the required behaviour as values:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions={'Class 0': [['Exponential', 1.0],
...                                        'NoArrivals',
...                                        'NoArrivals'],
...                            'Class 1': [['Exponential', 2.0],
...                                        'NoArrivals',
...                                        'NoArrivals']},
...     Service_distributions={'Class 0': [['Exponential', 4.0],
...                                        ['Exponential', 1.0],
...                                        ['Deterministic', 0.0]],
...                            'Class 1': [['Exponential', 6.0],
...                                        ['Deterministic', 0.0],
...                                        ['Exponential', 1.0]]},
...     Transition_matrices={'Class 0': [[0.0, 1.0, 0.0],
...                                      [0.0, 0.0, 0.0],
...                                      [0.0, 0.0, 0.0]],
...                          'Class 1': [[0.0, 0.0, 1.0],
...                                      [0.0, 0.0, 0.0],
...                                      [0.0, 0.0, 0.0]]},
...     Number_of_servers=[1, 2, 3],
... )
```

Notice that where we know certain customer classes will not require a service (for example babies will never require service at the children's specialist: Class 0 customers will never require service at Node 3) we are still required to input a service distribution. We choose the dummy distribution `['Deterministic', 0.0]`.

Let's simulate this clinic for 9 hours:

```
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(9)
>>> recs = Q.get_all_records()
```

Now we should see that no customer of Class 0 ever reached Node 3; and no customer of Class 1 ever reached Node 2:

```
>>> visited_by_babies = {1, 2}
>>> set([r.node for r in recs if r.customer_class==0]) == visited_by_babies
True

>>> visited_by_children = {1, 3}
>>> set([r.node for r in recs if r.customer_class==1]) == visited_by_children
True
```

Now say we'd like to find the average waiting time at the reception, baby specialist's clinic, and children's specialist's clinic. We'll simulate for 24 hours, using 3 hour warm-up and 3 hour cool-down, for 16 trials. Let's collect the average waiting times at each node every time:

```
>>> average_waits_1 = []
>>> average_waits_2 = []
>>> average_waits_3 = []
>>> for trial in range(16):
...     ciw.seed(trial)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(30)
...     recs = Q.get_all_records()
...     waits1 = [r.waiting_time for r in recs if r.node==1 and r.arrival_date > 3␣
↪and r.arrival_date < 27]
...     waits2 = [r.waiting_time for r in recs if r.node==2 and r.arrival_date > 3␣
↪and r.arrival_date < 27]
...     waits3 = [r.waiting_time for r in recs if r.node==3 and r.arrival_date > 3␣
↪and r.arrival_date < 27]
...     average_waits_1.append(sum(waits1) / len(waits1))
...     average_waits_2.append(sum(waits2) / len(waits2))
...     average_waits_3.append(sum(waits3) / len(waits3))
```

Now we can find the average wait over the trials:

```
>>> sum(average_waits_1) / len(average_waits_1)
0.244591...

>>> sum(average_waits_2) / len(average_waits_2)
0.604267...

>>> sum(average_waits_3) / len(average_waits_3)
0.252556...
```

These results imply that on average babies wait 0.6 of an hour, around 36 minutes for an appointment. This could then be used as a baseline measure against which to compare potential reconfigurations of the clinic.

CHAPTER 4

Guides

This selection of How-to guides will give a tour of some of the features that Ciw has to offer.

Contents:

4.1 How to Set a Seed

To ensure reproducibility of results users can set a seed for all the random number streams that Ciw uses. This can be done using the Ciw functon `ciw.seed`:

```
>>> import ciw
>>> ciw.seed(5)
```

Note that due to sampling on initialisation, the seed will need to be set **before** the `ciw.Simulation` object is created.

As an example, take the following network:

```
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 5]],
...     Service_distributions=[['Exponential', 10]],
...     Number_of_servers=[1]
... )
```

Now let's run the system for 20 time units, using a seed of 1, and get the average waiting time:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(20)
>>> waits = [r.waiting_time for r in Q.get_all_records()]
>>> sum(waits)/len(waits)
0.0544115013161...
```

Using the same seed again, the exact same average waiting time result will occur:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(20)
>>> waits = [r.waiting_time for r in Q.get_all_records()]
>>> sum(waits)/len(waits)
0.0544115013161...
```

Now using a different seed, a different result will occur:

```
>>> ciw.seed(2)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(20)
>>> waits = [r.waiting_time for r in Q.get_all_records()]
>>> sum(waits)/len(waits)
0.0832990490158...
```

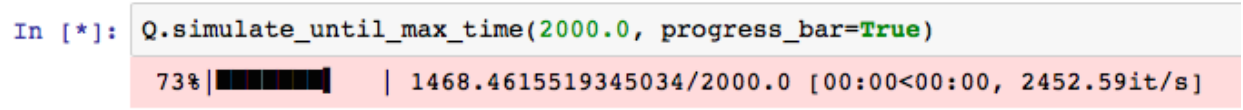
4.2 How to Implement a Progress Bar

For an individual run of a simulation, Ciw can enable a progress bar to appear. This can help visualise how far through a simulation run currently is. A progress bar may be implemented when using the methods `simulate_until_max_time` and `simulate_until_max_customers`. In order to implement this, add the option `progress_bar=True`.

An example when using the `simulate_until_max_time` method:

```
>>> Q.simulate_until_max_time(2000.0, progress_bar=True)
```

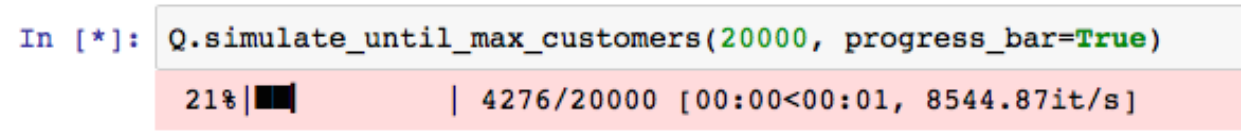
The image below shows an example of the output:



An example when using the `simulate_until_max_customers` method:

```
>>> Q.simulate_until_max_customers(20000, progress_bar=True)
```

And the image below shows the output:



4.3 How to Implement Exact Arithmetic

Due to the issues and limitations that arise when dealing with floating point numbers, Ciw offers an exact arithmetic option. Beware however, that using this option may affect performance, and so should only be used if issues with floating point numbers are affecting your results. This may happen for example while using deterministic distributions with server schedules.

In order to implement exact arithmetic, add this argument when creating the simulation object:

```
>>> Q = ciw.Simulation(N, exact=26)
```

The argument `exact` is used to indicate the precision level.

Let's look at an example:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 5]],
...     Service_distributions=[['Exponential', 10]],
...     Number_of_servers=[1]
... )
```

Without envoking exact arithmetic, we see that floats are used throughout:

```
>>> ciw.seed(2)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(100.0)
>>> waits = [r.waiting_time for r in Q.get_all_records()]
>>> waits[-1]
0.202518877171...
>>> type(waits[-1])
<class 'float'>
```

When envoking exact arithmetic, `decimal.Decimal` types are used throughout:

```
>>> ciw.seed(2)
>>> Q = ciw.Simulation(N, exact=26)
>>> Q.simulate_until_max_time(100.0)
>>> waits = [r.waiting_time for r in Q.get_all_records()]
>>> waits[-1]
Decimal('0.20251887717143828 60')
>>> type(waits[-1])
<class 'decimal.Decimal'>
```

4.4 How to Simulate For a Certain Number of Customers

A simulation run may be terminated once a certain number of customers have passed through. This can be done using the `simulate_until_max_customers` method. The method takes in a variable `max_customers`. There are three methods of counting customers:

- `'Finish'`: Simulates until `max_customers` has reached the Exit Node.
- `'Arrive'`: Simulates until `max_customers` have spawned at the Arrival Node.
- `'Accept'`: Simulates until `max_customers` have been spawned and accepted (not rejected) at the Arrival Node.

The method of counting customers is specified with the optional keyword argument `method`. The default value is is `'Finish'`.

Consider an *M/M/1/3* queue:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 10]],
```

(continues on next page)

(continued from previous page)

```
...     Service_distributions=[['Exponential', 5]],
...     Number_of_servers=[1],
...     Queue_capacities=[3]
... )
```

To simulate until 30 customers have finished service:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_customers(30, method='Finish')
>>> len(Q.nodes[-1].all_individuals)
30
```

To simulate until 30 customers have arrived:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_customers(30, method='Arrive')
>>> len(Q.nodes[-1].all_individuals), len(Q.nodes[1].all_individuals), len(Q.
↪rejection_dict[1][0])
(13, 3, 14)
```

To simulate until 30 customers have been accepted:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_customers(30, method='Accept')
>>> len(Q.nodes[-1].all_individuals), len(Q.nodes[1].all_individuals)
(26, 4)
```

4.5 How to Set Arrival & Service Distributions

Ciw offeres a variety of inter-arrival and service time distributions. A full list can be found *here*. They are defined when creating a Network with the `'Arrival_distributions'` and `'Service_distributions'` keywords.

- `'Arrival_distributions'`: This is the distribution that inter-arrival times are drawn from. That is the time between two consecutive arrivals. It is particular to specific nodes and customer classes.
- `'Service_distributions'`: This is the distribution that service times are drawn from. That is the amount of time a customer spends with a server (independent of how many servers there are). It is particular for to specific node and customer classes.

The following example, with two nodes and two customer classes, uses eight different arrival and service rate distributions:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions={'Class 0': [['Deterministic', 0.4],
...                                        ['Empirical', [0.1, 0.1, 0.1, 0.2]]],
...                            'Class 1': [['Deterministic', 0.2],
...                                        ['Custom', [0.2, 0.4], [0.5, 0.5]]]},
...     Service_distributions={'Class 0': [['Exponential', 6.0],
...                                        ['Lognormal', -1, 0.5]],
...                            'Class 1': [['Uniform', 0.1, 0.7],
```

(continues on next page)

(continued from previous page)

```
...                                          ['Triangular', 0.2, 0.7, 0.3]]},
...     Transition_matrices={'Class 0': [[0.0, 0.0], [0.0, 0.0]],
...                          'Class 1': [[0.0, 0.0], [0.0, 0.0]]},
...     Number_of_servers=[1, 1]
... )
```

We'll run this (in *exact* mode) for 25 time units:

```
>>> ciw.seed(10)
>>> Q = ciw.Simulation(N, exact=10)
>>> Q.simulate_until_max_time(50)
>>> recs = Q.get_all_records()
```

The system uses the following eight distributions:

- **['Deterministic', 0.4]:**
  - Always sample 0.4.
- **['Deterministic', 0.2]:**
  - Always sample 0.2.
- **['Empirical', [0.1, 0.1, 0.1, 0.2]:**
  - Randomly sample from the numbers 0.1, 0.1, 0.1 and 0.2.
- **['Custom', [0.2, 0.4], [0.5, 0.5]]:**
  - Sample 0.2 half the time, and 0.4 half the time.
- **['Exponential', 6.0]:**
  - Sample from the exponential distribution with parameter $\lambda = 6.0$. Expected mean of 0.1666...
- **['Uniform', 0.1, 0.7]:**
  - Sample any number between 0.1 and 0.7 with equal probablity. Expected mean of 0.4.
- **['Lognormal', -1, 0.5]:**
  - Sample from the lognormal distribution with parameters $\mu = -1$ and $\sigma = 0.5$. Expected mean of 0.4724...
- **['Triangular', 0.2, 0.7, 0.3]:**
  - Sample from the triangular distribution, with mode 0.3, lower limit 0.2 and upper limit 0.7. Expected mean of 0.4.

From the records, collect the service times and arrival dates for each node and each customer class:

```
>>> servicetimes_n1c0 = [r.service_time for r in recs if r.node==1 and r.customer_
↪class==0]
>>> servicetimes_n2c0 = [r.service_time for r in recs if r.node==2 and r.customer_
↪class==0]
>>> servicetimes_n1c1 = [r.service_time for r in recs if r.node==1 and r.customer_
↪class==1]
>>> servicetimes_n2c1 = [r.service_time for r in recs if r.node==2 and r.customer_
↪class==1]
>>> arrivals_n1c0 = sorted([r.arrival_date for r in recs if r.node==1 and r.customer_
↪class==0])
>>> arrivals_n2c0 = sorted([r.arrival_date for r in recs if r.node==2 and r.customer_
↪class==0])
```

(continues on next page)

(continued from previous page)

```
>>> arrivals_n1c1 = sorted([r.arrival_date for r in recs if r.node==1 and r.customer_
↪class==1])
>>> arrivals_n2c1 = sorted([r.arrival_date for r in recs if r.node==2 and r.customer_
↪class==1])
```

Now let's see if the mean service time and inter-arrival times of the simulation matches the distributions:

```
>>> from decimal import Decimal

>>> sum(servicetimes_n1c0) / len(servicetimes_n1c0) # Expected 0.1666...
Decimal('0.1650563448')

>>> sum(servicetimes_n2c0) / len(servicetimes_n2c0) # Expected 0.4724...
Decimal('0.4228601677')

>>> sum(servicetimes_n1c1) / len(servicetimes_n1c1) # Expected 0.4
Decimal('0.4352210564')

>>> sum(servicetimes_n2c1) / len(servicetimes_n2c1) # Expected 0.4
Decimal('0.4100529676')

>>> set([r2-r1 for r1, r2 in zip(arrivals_n1c0, arrivals_n1c0[1:])]) # Should only␣
↪sample 0.4
{Decimal('0.4')}

>>> set([r2-r1 for r1, r2 in zip(arrivals_n1c1, arrivals_n1c1[1:])]) # Should only␣
↪sample 0.2
{Decimal('0.2')}

>>> expected_samples = {Decimal('0.2'), Decimal('0.1')} # Should only sample 0.1 and␣
↪0.2
>>> set([r2-r1 for r1, r2 in zip(arrivals_n2c0, arrivals_n2c0[1:])]) == expected_
↪samples
True

>>> expected_samples = {Decimal('0.2'), Decimal('0.4')}#  Should only sample 0.2 and␣
↪0.4
>>> set([r2-r1 for r1, r2 in zip(arrivals_n2c1, arrivals_n2c1[1:])]) == expected_
↪samples
True
```

4.6 How to Define Time Dependent Distributions

In Ciw we can get a time dependent distribution, that is a service time, inter-arrival time, or batching distribution that changes as the simulation time progresses. In order to do this a time dependent function, that returns a sampled time, must be defined. This must take in a time variable *t*.

For example, say we wish to have arrivals once every 30 minutes in the morning, every 15 minutes over lunch, every 45 minutes in the afternoon, and every 90 minutes throughout the night:

```
>>> def time_dependent_function(t):
...     if t % 24 < 12.0:
...         return 0.5
...     if t % 24 < 14.0:
```

(continues on next page)

(continued from previous page)

```
...         return 0.25
...     if t % 24 < 20.0:
...         return 0.75
...     return 1.5
```

This function returns inter-arrival times of 0.5 hrs between midnight (0) and 12, 0.25 hrs between 12 and 14, 0.75 hrs between 14 and 20, and 1.5 hrs between 20 and midnight (24). Then repeats. Testing this function we see:

```
>>> time_dependent_function(9.5)
0.5
>>> time_dependent_function(11.0)
0.5
>>> time_dependent_function(13.25)
0.25
>>> time_dependent_function(17.0)
0.75
>>> time_dependent_function(22.0)
1.5
>>> time_dependent_function(33.2) # half 9 the next day
0.5
```

Let's implement this into a one node infinite server queue:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['TimeDependent', time_dependent_function]],
...     Service_distributions=[['Deterministic', 0.0]],
...     Number_of_servers=['Inf']
... )
```

We'll then simulate this for 1 day. We would expect 24 arrivals in the morning (12 hours, one every half an hour); 8 arrivals over lunch (2 hours, one every 15 minutes); 8 arrivals in the afternoon (6 hours, one every 45 mins); and 2 arrivals in the night (4 hours, one every hour and a half). Therefore a total of 42 customers passed through the system:

```
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(24.0)

>>> len(Q.nodes[-1].all_individuals)
42
```

4.7 How to Set Priority Classes

Ciw has the capability to assign priorities to the customer classes. This is done by mapping customer classes to priority classes, included as a keyword when creating the Network object. An example is shown:

```
Priority_classes={'Class 0': 0,
                  'Class 1': 1,
                  'Class 2': 1}
```

This shows a mapping from three customer classes to two priority classes. Customers in class 0 have the highest priority and are placed in priority class 0. Customers in class 1 and class 2 are both placed in priority class 1; they have the same priority as each other but less that those customers in class 0.

Note:

- The lower the priority class number, the higher the priority. Customers in priority class 0 have higher priority than those with in priority class 1, who have higher priority than those in priority class 2, etc.
- Priority classes are essentially Python indices, therefore if there are a total of 5 priority classes, priorities **must** be labelled 0, 1, 2, 3, 4. Skipping a priority class, or naming priority classes anything other than increasing integers from 0 will cause an error.
- The priority discipline used is non-preemptive. Customers always finish their service and are not interrupted by higher priority customers.

To implement this, create the Network object with the `Priority_classes` option included with the mapping:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions={'Class 0': [['Exponential', 5]],
...                            'Class 1': [['Exponential', 5]]},
...     Service_distributions={'Class 0': [['Exponential', 10]],
...                            'Class 1': [['Exponential', 10]]},
...     Priority_classes={'Class 0': 0, 'Class 1': 1},
...     Number_of_servers=[1]
... )
```

Now let's run the simulation, comparing the waiting times for Class 0 and Class 1 customers, those with higher priority should have lower average wait than those with lower priority:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(100.0)
>>> recs = Q.get_all_records()

>>> waits_0 = [r.waiting_time for r in recs if r.customer_class==0]
>>> sum(waits_0)/len(waits_0)
0.1529189...

>>> waits_1 = [r.waiting_time for r in recs if r.customer_class==1]
>>> sum(waits_1)/len(waits_1)
3.5065047...
```

4.8 How to Set Batch Arrivals

Ciw allows batch arrivals, that is more than one customer arriving at the same time. This is implemented using the `Batching_distributions`. Similar to `Arrival_distributions` and `Service_distributions`, this takes in distributions for each node and customer class that will sample the size of the batch. Only discrete distributions are allowed, that is distributions that sample integers.

Let's show an example:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Deterministic', 18.5]],
...     Service_distributions=[['Deterministic', 3.0]],
...     Batching_distributions=[['Deterministic', 3]],
...     Number_of_servers=[1]
... )
```

If this system is simulated for 30 time units, only one arrival event will occur. However, 3 customers will arrive at that node simultaneously. As there is only one server, two of those customers will have to wait:

```
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(30.0)
>>> recs = Q.get_all_records()

>>> [r.arrival_date for r in recs]
[18.5, 18.5, 18.5]
>>> [r.waiting_time for r in recs]
[0.0, 3.0, 6.0]
```

Just like arrival and service distributions, batching distributions can be defined for multiple nodes and multiple customer classes, using lists and dictionaries:

```
Batching_distributions={
    'Class 0': [['Deterministic', 3],
                ['Deterministic', 1]],
    'Class 1': [['Deterministic', 2],
                ['Deterministic', 2]]},
```

Note:

- ***Only discrete distributions may be used,*** **currently implemented are:**
  - `Deterministic`
  - `Empirical`
  - `Custom`
  - `Sequential`
  - `TimeDependent`
- If the keyword `Batching_distributions` is omitted, then no batching is assumed. That is only one customer arrives at a time. Equivalent to `['Deterministic', 1]`.
- If some nodes/customer classes require no batching, but others do, please use `['Deterministic', 1]`.
- Batch arrivals may lead to *simultaneous events*, please take care.

4.8.1 How to Set Time Dependent Batches

Ciw allows batching distributions to be time dependent. That is the batch size, the number of customers arriving simultaneously, is sampled from a distribution that varies with time.

Let's show an example, we wish to have batch sizes of 2 for the first 10 time units, but batch sizes of 1 thereafter. Define a time dependent batching distribution:

```
>>> def time_dependent_batches(t):
...     if t < 10.0:
...         return 2
...     return 1
```

Now use this when defining a network:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Deterministic', 3.0]],
...     Service_distributions=[['Deterministic', 0.5]],
```

(continues on next page)

(continued from previous page)

```
...     Batching_distributions=[['TimeDependent', time_dependent_batches]],
...     Number_of_servers=[1]
... )
```

We'll simulate this for 16 time units. Now at times 3, 6, and 9 we would expect 2 customers arriving (a total of 6). And at times 12 and 15 we would expect 1 customer arriving (a total of 2). So 8 customers in total should finish service:

```
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(16.0)
>>> len(Q.nodes[-1].all_individuals)
8
```

4.9 How to Simulate Baulking Customers

Ciw allows customer's to baulk (decide not join the queue) upon arrival, according to baulking functions. These functions take in a parameter `n`, the number of individuals at the node, and returns a probability of baulking.

For example, say we have an *M/M/1* system where customers:

- Never baulk if there are less than 3 customers in the system
- Have probability 0.5 of baulking if there are between 3 and 6 customers in the system
- Always baulk if there are more than 6 customers in the system

We can define the following baulking function:

```
>>> def probability_of_baulking(n):
...     if n < 3:
...         return 0.0
...     if n < 7:
...         return 0.5
...     return 1.0
```

When creating the Network object we tell Ciw which node and customer class this function applies to with the `Baulking_functions` keyword:

```
>>> import ciw
>>> N = ciw.create_network(
...      Arrival_distributions={'Class 0': [['Exponential', 5]]},
...      Service_distributions={'Class 0': [['Exponential', 10]]},
...      Baulking_functions={'Class 0': [probability_of_baulking]},
...      Number_of_servers=[1]
... )
```

When the system is simulated, the baulked customers are recorded in the Simulation object's `baulked_dict`. This is a dictionary, that maps node numbers to dictionaries. These dictionaries map customer class numbers to a list of dates at which customers baulked:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(45.0)
>>> Q.baulked_dict
{1: {0: [21.1040..., 42.2023..., 43.7558..., 43.7837..., 44.2266...]}}
```

Note that baulking works and behaves differently to simply setting a queue capacity. Filling a queue's capacity results in arriving customers being *rejected* (and recorded in the `rejection_dict`), and transitioning customers to be blocked. Baulking on the other hand does not effect transitioning customers, and customer who have baulked are recorded in the `baulked_dict`. This means that if you set a deterministic baulking threshold of 5, but do not set a queue capacity, then the number of individuals at that node may exceed 5, due to customers transitioning from other nodes ignoring the baulking threshold. This also means you can use baulking and limited capacities in conjunction with one another.

4.10 How to Set Server Schedules

Ciw allows users to assign cyclic work schedules to servers at each service centre. An example cyclic work schedule is shown in the table below:

Shift Times	0-10	10-30	30-100
Number of Servers	2	0	1

This schedule is cyclic, therefore after the last shift (30-100), schedule begins again with the shift (0-10). The cycle length for this schedule is 100. This is defines by a list of lists indicating the number of servers that should be on duty during that shift, and the end date of that shift:

```
[[2, 10], [0, 30], [1, 100]]
```

Here we are saying that there will be 2 servers scheduled between times 0 and 10, 0 between 10 and 30, etc. This fully defines the cyclic work schedule.

To tell Ciw to use this schedule for a given node, in the `Number_of_servers` keyword we replace an integer with the schedule:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 5]],
...     Service_distributions=[['Exponential', 10]],
...     Number_of_servers=[[[2, 10], [0, 30], [1, 100]]]
... )
```

Simulating this system, we'll see that no services begin between dates 10 and 30, nor between dates 110 and 130:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(205.0)
>>> recs = Q.get_all_records()

>>> [r for r in recs if 10 < r.service_start_date < 30]
[]
>>> [r for r in recs if 110 < r.service_start_date < 130]
[]
```

Note that currently server schedules are incompatible with infinite servers, and so a schedule cannot include infinite servers.

4.10.1 Pre-emption

When a server is due to go off duty during a customer's service, there are two options of what may happen.

- During a pre-emptive schedule, that server will immediately stop service and leave. Whenever more servers come on duty, they will prioritise the interrupted customers and continue their service. However those customers' service times are re-sampled.
- During a non-pre-emptive schedule, customers cannot be interrupted. Therefore servers finish the current customer's service before disappearing. This of course may mean that when new servers arrive the old servers are still there.

In order to implement pre-emptive or non-pre-emptive schedules, put the schedule in a tuple with a `True` or a `False` as the second term, indicating pre-emptive or non-pre-emptive interruptions. For example:

```
Number_of_servers=[([[2, 10], [0, 30], [1, 100]], True)] # preemptive
```

And:

```
Number_of_servers=[([[2, 10], [0, 30], [1, 100]], False)] # non-preemptive
```

Ciw defaults to non-pre-emptive schedules, and so the following code implies a non-pre-emptive schedule:

```
Number_of_servers=[[[2, 10], [0, 30], [1, 100]]] # non-preemptive
```

4.10.2 Overtime

Non-preemptive schedules allow for the possibility of overtime, that is servers working after their shift has ended in order to complete a customer's service. The amount of overtime each server works is recorded in the Node object's `overtime` attribute. Consider the following example:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Deterministic', 3.0]],
...     Service_distributions=[['Deterministic', 5.0]],
...     Number_of_servers=[[[1, 4.0], [2, 10.0], [0, 100.0]]]
... )
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(20.0)

>>> Q.transitive_nodes[0].overtime
[4.0, 1.0, 4.0]
```

Here we see that the first server that went off duty worked 4.0 time units of overtime, the second worked 1.0 time unit of overtime, and the third worked 4.0 time units of overtime.

4.11 How to Set Dynamic Customer Classes

Ciw 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 $P(J = j \mid I = i, K = k)$. These probabilities are input into the system through the `Class_change_matrices` keyword.

Consider a one node system with three classes of customer. After service (at Node 1) customers always change customer class, equally likely between the two other customer classes. The `Class_change_matrices` for this system are shown below:

$$\begin{pmatrix} 0.0 & 0.5 & 0.5 \\ 0.5 & 0.0 & 0.5 \\ 0.5 & 0.5 & 0.0 \end{pmatrix}$$

This is input into the simulation model by including `Class_change_matrices` keyword when creating a Network object:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions={'Class 0': [['Exponential', 5]],
...                            'Class 1': ['NoArrivals'],
...                            'Class 2': ['NoArrivals']},
...     Service_distributions={'Class 0': [['Exponential', 10]],
...                            'Class 1': [['Exponential', 10]],
...                            'Class 2': [['Exponential', 10]]},
...     Transition_matrices={'Class 0': [[1.0]],
...                          'Class 1': [[1.0]],
...                          'Class 2': [[1.0]]},
...     Class_change_matrices={'Node 1': [[0.0, 0.5, 0.5],
...                                       [0.5, 0.0, 0.5],
...                                       [0.5, 0.5, 0.0]]},
...     Number_of_servers=[1]
... )
```

Notice in this network only arrivals from Class 0 customer occur. Running this system, we'll see that the count of the number of records with customer classes 1 and 2 more than zero, as some Class 0 customers have changed class after service:

```
>>> from collections import Counter
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(50.0)
>>> recs = Q.get_all_records()
>>> Counter([r.customer_class for r in recs])
Counter({0: 255, 2: 125, 1: 105})
```

Note that when more than one node is used, each node requires a class change matrix. This means than difference class change matrices can be used for each node.

4.12 How to Detect Deadlock

Deadlock is the phenomenon whereby all movement and customer flow in a restricted queueing network ceases, due to circular blocking. The diagram below shows an example, where the customer at the top node is blocked to the bottom node, and the customer at the bottom node is blocked to the top node. This circular blockage results is no more natural movement happening.

Ciw's has built in deadlock detection capability. With Ciw, a queueing network can be simulated until it reaches deadlock. Ciw then records the time until deadlock from each state. (Please see the documentation on *state trackers*.)

In order to take advantage of this feature, set the `deadlock_detection` argument to one of the deadlock detection methods when creating the Simulation object. Currently only `'StateDigraph'` is implemented. Then use the `simulate_until_deadlock` method. The attribute `times_to_deadlock` contains the times to deadlock from each state.

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:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 6.0]],
...     Service_distributions=[['Exponential', 5.0]],
...     Transition_matrices=[[0.5]],
...     Number_of_servers=[1],
...     Queue_capacities=[3]
... )
```

Running until deadlock:

```
>>> import ciw
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N, deadlock_detector='StateDigraph')
>>> Q.simulate_until_deadlock()
>>> Q.times_to_deadlock
{((0, 0),): 0.94539784..., ((1, 0),): 0.92134933..., ((2, 0),): 0.68085451..., ((3,␣
↪0),): 0.56684471..., ((3, 1),): 0.0, ((4, 0),): 0.25332344...}
```

Here the keys correspond to states recorded by the state tracker.

4.12.1 How to Set a State Tracker

Ciw has the option to activate a state `tracker` in order to track the state of the system as the simulation progresses towards deadlock. The default is the basic `StateTracker` which does nothing (unless the simulation is detecting deadlock, in which case `NaiveTracker` is the default). The state trackers have their uses when simulating until deadlock, as a time to deadlock is recorded for every state the simulation reaches.

For a list and explanation of the state trackers that Ciw currently supports see *List of Implemented State Trackers for Deadlock Detection*.

Consider the M/M/2/1 queue with a feedback loop. The following states are expected if a Naive Tracker is used: ((0, 0)), ((1, 0)), ((2, 0)), ((3, 0)), ((2, 1)), ((1, 2)). Simulating until deadlock, the `times_to_deadlock` dictionary will contain a subset of these states as keys:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 6.0]],
...     Service_distributions=[['Exponential', 5.0]],
...     Transition_matrices=[[0.5]],
...     Number_of_servers=[2],
...     Queue_capacities=[1]
... )

>>> ciw.seed(1)
>>> Q = ciw.Simulation(N, deadlock_detector='StateDigraph', tracker='Naive')
>>> Q.simulate_until_deadlock()
>>> Q.times_to_deadlock
{((0, 0),): 1.3354..., ((1, 0),): 1.3113..., ((1, 2),): 0.0, ((2, 0),): 1.0708...,␣
↪((2, 1),): 0.9353..., ((3, 0),): 0.9568...}
```

The following states are expected if a Matrix Tracker is used: ((()), (0)), ((()), (1)), ((()), (2)), ((()), (3)), (((1)), (3)), (((1, 2)), (3)).

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N, deadlock_detector='StateDigraph', tracker='Matrix')
```

(continues on next page)

(continued from previous page)

```
>>> Q.simulate_until_deadlock()
>>> Q.times_to_deadlock
{((((),),), (0,)): 1.3354..., ((((),),), (1,)): 1.3113..., ((((),),), (2,)): 1.0708...
↪, ((((),),), (3,)): 0.9568..., ((((1,),),),), (3,)): 0.9353..., ((((1, 2),),),), (3,)):␣
↪0.0}
```

Notice that in this simple case, the Naive and Matric trackers correspond to the same states. In other examples, where customers may get blocked in different orders and to different places, then the two trackers may track different system states.

4.13 How to Get More Custom Behaviour

Custom behaviour can be obtained by writing new `Node` and `ArrivalNode` classes, that inherit from the original `ciw.Node` and `ciw.ArrivalNode` classes, that introduce new beahviour into the system. The classes that can be overwitten are:

- `Node`: the main node class used to represent a service centre.
- `ArrivalNode`: the node class used to generate individuals and route them to a specific `Node`.

These new Node and Arrival Node classes can be used with the Simulation class by using the keyword arugments `node_class` and `arrival_node_class`.

Consider the following two node network, where arrivals only occur at the first node, and there is a queueing capacity of 10. The second node is redundent in this scenario:

```
>>> import ciw
>>> from collections import Counter

>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 6.0], 'NoArrivals'],
...     Service_distributions=[['Exponential', 5.0], ['Exponential', 5.0]],
...     Transition_matrices=[[0.0, 0.0], [0.0, 0.0]],
...     Number_of_servers=[1, 1],
...     Queue_capacities=[10, 'Inf']
... )
```

Now we run the system for 100 time units, and see that we get 484 services at the first node, and none at the second node:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N)
>>> Q.simulate_until_max_time(100)

>>> service_nodes = [r.node for r in Q.get_all_records()]
>>> Counter(service_nodes)
Counter({1: 484})
```

We will now create a new `CustomArrivalNode` such that any customers who arrive when the first node has 10 or more customers present will be sent to the second node. First create the `CustomArrivalNode` that inherits from `ciw.ArrivalNode`, and overwrites the `send_individual` method:

```
>>> class CustomArrivalNode(ciw.ArrivalNode):
...     def send_individual(self, next_node, next_individual):
...         """
```

(continues on next page)

(continued from previous page)

```
...         Sends the next_individual to the next_node
...         """
...         self.number_accepted_individuals += 1
...         if len(next_node.all_individuals) <= 10:
...             next_node.accept(next_individual, self.next_event_date)
...         else:
...             self.simulation.nodes[2].accept(next_individual, self.next_event_date)
```

To run the same system, we need to remove the keyword `'Queue_capacities'` when creating a network, so that customers are not rejected before reaching the `send_individual` method:

```
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 6.0], 'NoArrivals'],
...     Service_distributions=[['Exponential', 5.0], ['Exponential', 5.0]],
...     Transition_matrices=[[0.0, 0.0], [0.0, 0.0]],
...     Number_of_servers=[1, 1]
... )
```

Now rerun the same system, telling Ciw to use the new `arrival_node_class` to use. We'll see that the same amount of services take place at Node 1, however rejected customers now have services taking place at Node 2:

```
>>> ciw.seed(1)
>>> Q = ciw.Simulation(N, arrival_node_class=CustomArrivalNode)
>>> Q.simulate_until_max_time(100)

>>> service_nodes = [r.node for r in Q.get_all_records()]
>>> Counter(service_nodes)
Counter({1: 484, 2: 85})
```

4.14 How to Read & Write to/from File

When running experiments, it may be useful to read in parameters from a file, and to export data records to file. This can be done easily in Ciw. Parameter dictionaries can be represented as `.yml` files, and results can be output as `.csv` files.

4.14.1 Parameter Files

Consider the following Network:

```
>>> import ciw
    >>> N = ciw.create_network(
    ...     Arrival_distributions={'Class 0': [['Exponential', 6.0], ['Exponential',␣
↪2.5]]},
    ...     Service_distributions={'Class 0': [['Exponential', 8.5], ['Exponential',␣
↪5.5]]},
    ...     Transition_matrices={'Class 0': [[0.0, 0.2], [0.1, 0.0]]},
    ...     Number_of_servers=[1, 1],
    ...     Queue_capacities=['Inf', 4]
    ... )
```

This can be represented by the `.yml` file below:

```
parameters.yml

Arrival_distributions:
  Class 0:
  - - Exponential
    - 6.0
  - - Exponential
    - 2.5
Service_distributions:
  Class 0:
  - - Exponential
    - 8.5
  - - Exponential
    - 5.5
Transition_matrices:
  Class 0:
  - - 0.0
    - 0.2
  - - 0.1
    - 0.0
Number_of_servers:
- 1
- 1
Queue_capacities:
- "Inf"
- 4
```

This can then be created into a Network object using the `ciw.create_network_from_yml` function:

```
>>> import ciw
>>> N = ciw.create_network_from_yml('<path_to_file>')
>>> Q = ciw.Simulation(N)
```

4.14.2 Exporting Results

Once a simulation has been run, all data records can be exported to file using the `write_records_to_file` method of the Simulation object. This method writes all results that are obtained by the `get_all_records` method (see *here* for more information) to a `.csv` file, where each row is an observation ad each column a variable:

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

This method also takes the optional keyword argument `header`. If this is set to `True` then the first row of the `.csv` file will be the variable names. The default value is `True`, set to `False` is a row of variable names are not needed:

```
>>> Q.write_records_to_file('<path_to_file>', headers=True)
>>> Q.write_records_to_file('<path_to_file>', headers=False)
```

CHAPTER 5

Reference

Contents:

5.1 List of Parameters

Below is a full list of the parameters that the `create_network` function can take, along with a description of the values required. If using a parameters file then here are the arguments and values of the required `.yml` file.

5.1.1 Arrival_distributions

Required

Describes the inter-arrival distributions for each node and customer class. This is a dictionary, with keys as customer classes, and values are lists describing the inter-arrival distributions for each node. If only one class of customer is required it is sufficient to simply enter a list of inter-arrival distributions. For more details on inputting distributions, see *How to Set Arrival & Service Distributions*.

An example is shown:

```
Arrival_distributions={'Class 0': [['Exponential', 2.4],
                                   ['Uniform', 0.3, 0.5]],
                       'Class 1': [['Exponential', 3.0],
                                   ['Deterministic', 0.8]]}
```

An example where only one class of customer is required:

```
Arrival_distributions=[['Exponential', 2.4],
                       ['Exponential', 2.0]]
```

5.1.2 Batching_distributions

Optional

Describes the discrete distributions of size of the batch arrivals for each node and customer class. This is a dictionary, with keys as customer classes, and values are lists describing the batch distributions for each node. If only one class of customer is required it is sufficient to simply enter a list of batch distributions. For more details on batching, see *How to Set Batch Arrivals*.

An example is shown:

```
Batching_distributions={'Class 0': [['Deterministic', 1],
                                    ['Sequential', [1, 1, 2]]],
                        'Class 1': [['Deterministic', 3],
                                    ['Deterministic', 2]]}
```

An example where only one class of customer is required:

```
Batching_distributions=[['Deterministic', 2],
                        ['Deterministic', 1]]
```

5.1.3 Baulking_functions

Optional

A dictionary of baulking functions for each customer class and each node. It describes the baulking mechanism of the customers. For more details see *How to Simulate Baulking Customers*. If left out, then no baulking occurs.

Example:

```
Baulking_functions={'Class 0': [probability_of_baulking]}
```

5.1.4 Class_change_matrices

Optional

A dictionary of class change matrices for each node. For more details see *How to Set Dynamic Customer Classes*.

An example for a two node network with two classes of customer:

```
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, 0.1],
                                  [0.2, 0.2, 0.6]]}
```

5.1.5 Number_of_servers

Required

A list of the number of parallel servers at each node. If a server schedule is used, the schedule is given instead of a number. For more details on server schedules, see *How to Set Server Schedules*. A value of 'Inf' may be given is infinite servers are required.

Example:

```
Number_of_servers=[1, 2, 'Inf', 1, 'schedule']
```

5.1.6 Priority_classes

Optional

A dictionary mapping customer classes to priorities. For more information see *How to Set Priority Classes*. If left out, no priorities are used, that is all customers have equal priorities.

Example:

```
Priority_classes={'Class 0': 0,
                  'CLass 1': 1,
                  'Class 2': 1}
```

5.1.7 Queue_capacities

Optional

A list of maximum queue capacities at each node. If ommitted, default values of 'Inf' for every node are given.

Example:

```
Queue_capacities=[5, 'Inf', 'Inf', 10]
```

5.1.8 Service_distributions

Required

Describes the service distributions for each node and customer class. This is a dictionary, with keys as customer classes, and values are lists describing the service distributions for each node. If only one class of customer is required it is sufficient to simply enter a list of service distributions. For more details on inputting distributions, see *How to Set Arrival & Service Distributions*.

An example is shown:

```
Service_distributions={'Class 0': [['Exponential', 4.4],
                                   ['Uniform', 0.1, 0.9]],
                       'Class 1': [['Exponential', 6.0],
                                   ['Lognormal', 0.5, 0.6]]}
```

An example where only one class of customer is required:

```
Service_distributions=[['Exponential', 4.8],
                       ['Exponential', 5.2]]
```

5.1.9 Transition_matrices

Required for more than 1 node

Optional for 1 node

Describes the transition matrix for each customer class. This is a dictionary, with keys as customer classes, and values are lists of lists (matrices) containing the transition probabilities. If only one class of customer is required it is sufficient to simply enter single transition matrix (a list of lists).

An example is shown:

```
Transition_matrices={'Class 0': [[0.1, 0.3],
                                 [0.0, 0.8]],
                     'Class 1': [[0.0, 1.0],
                                 [0.0, 0.0]]}
```

An example where only one class of customer is required:

```
Transition_matrices=[[0.5, 0.3],
                     [0.2, 0.6]]
```

If using only one node, the default value is:

```
Transition_matrices={'Class 0': [[0.0]]}
```

5.2 List of Available Results

Each time an individual completes service at a service station, a data record of that service is kept. The records should look something like the table below:

I.D	Class	Node	Arrival Date	Wait Time	Service Start Date	Service Time	Service End Date	Time Blocked	Exit Date	Dest.	Queue Size at Arrival	Queue Size at Depart.
22759	1	1	245.601	0.0	245.601	0.563	246.164	0.0	246.164	1	0	2
41129	0	1	245.633	0.531	246.164	0.608	246.772	0.0	246.772	1	1	5
00195	0	2	247.821	0.0	247.841	1.310	249.151	0.882	250.033	3	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...

You may access these records as a list of named tuples, using the Simulation's `get_all_records` method:

```
>>> recs = Q.get_all_records()
```

The data records contained in this list are named tuples with the following variable names:

- **id_number**
  - The unique identification number for that customer.
- **customer_class**
  - The number of that customer's customer class. If dynamic customer classes are used, this is the customer's previous class, before a new customer class is sampled after service.
- **node**
  - The number of the node at which the service took place.
- **arrival_date**
  - The date of arrival to that node, the date which the customer joined the queue.
- **waiting_time**

  - The amount of time the customer spent waiting for service at that node.
- **service_start_date**
  - The date at which service began at that node.
- **service_time**
  - The amount of time spent in service at that node.
- **service_end_date**
  - The date which the customer finished their service at that node.
- **time_blocked**
  - The amount of time spent blocked at that node. That is the time between finishing service at exiting the node.
- **exit_date**
  - The date which the customer exited the node. This may be immediatly after service if no blocking occured, or after some period of being blocked.
- **destination**
  - The number of the customer's destination, that is the next node the customer will join after leaving the current node. If the customer leaves the system, this will be -1.
- **queue_size_at_arrival**
  - The size of the queue at the customer's arrival date. Does not include the individual themselves.
- **queue_size_at_departure**
  - The size of the queue at the customer's exit date. Does not include the individual themselves.

5.3 List of Supported Distributions

Ciw allows a number continuous service and inter-arrival time distributions, as well as empirical, user defined, time dependent, and custom discrete distributions. Note that when choosing parameters for these distributions, ensure that no negative numbers may be sampled. The following are currently supported:

- *Time Dependent Distributions*
- *No Arrivals*

5.3.1 The Uniform Distribution

The uniform distribution samples a random number between two numbers a and b. Write a uniform distribution between *4* and *9* as follows:

```
['Uniform', 4.0, 9.0]
```

5.3.2 The Deterministic Distribution

The deterministic distribution is non-stochastic, and produces the same service time repeatedly. Write a deterministic distribution that repeatedly gives a value of *18.2* as follows:

```
['Deterministic', 18.2]
```

5.3.3 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*. Write a triangular distribution between *2.1* and *7.6* with mode of *3.4* as follows:

```
['Triangular', 2.1, 7.6, 3.4]
```

5.3.4 The Exponential Distribution

The exponential distribution samples a random number from the negative exponential distribution with mean $1/\lambda$. Write an exponential distribution with mean *0.2* as follows:

```
['Exponential', 5]
```

5.3.5 The Gamma Distribution

The gamma distribution samples a random number from the gamma distribution with shape parameter α and scale parameter β. Write a gamma distribution with parameters $\alpha = 0.6$ and $\beta = 1.2$ as follows:

```
['Gamma', 0.6, 1.2]
```

5.3.6 The Truncated Normal Distribution

The truncated normal distribution samples a random number from the normal distribution with mean μ and standard deviation σ. The distribution is truncated at 0, thus if negative numbers are sampled then that observation is resampled until a positive value is sampled. Write a normal distribution with parameters $\mu = 0.7$ and $\sigma = 0.4$ as follows:

```
['Normal', 0.7, 0.4]
```

5.3.7 The Lognormal Distribution

The lognormal distribution samples a random number from the log of the normal distribution with mean μ and standard deviation σ. Write a lognomal distribution, that is a log of the normal distribution with $\mu = 4.5$ and $\sigma = 2.0$, as follows:

```
['Lognormal', 4.5, 2.0]
```

5.3.8 The Weibull Distribution

The Weibull distribution samples a random number from the Weibull distribution with scale parameter α and shape parameter β. Write a Weibull distribution with $\alpha = 0.9$ and $\beta = 0.8$ as follows:

```
['Weibull', 0.9, 0.8]
```

5.3.9 Empirical Distributions

There are two methods of defining empirical distributions in Ciw, either by inputting a list of observations, or through giving a path to a `.csv` file containing observations:

Input list of observations:

```
['Empirical', [0.3, 0.3, 0.3, 0.4, 0.5, 0.6, 0.8, 0.9, 1.1, 1.1, 1.1, 1.1]]
```

Input path to `.csv` file:

```
['Empirical', '<path_to_file>']
```

5.3.10 Sequential Distributions

The sequential distribution takes a list, and iteratively returns the next observation in that list over time. The distribution is cyclic, and so once all elements of the list have been sampled, the sequence of sampled values begins again from the beginning of the list:

```
['Sequential', [0.1, 0.1, 0.2, 0.1, 0.3, 0.2]]
```

5.3.11 Custom PDFs

Ciw allows users to define their own custom PDFs to sample from. This distribution samples from a set of values given a probability for each value, that is sampling the value x with probability $P(x)$. For example, if $P(1.4) = 0.2$, $P(1.7) = 0.5$, and $P(1.9) = 0.3$, this is defined in the following way:

```
['Custom', [1.4, 1.7, 1.9], [0.2, 0.5, 0.3]]
```

5.3.12 User Defined Distributions

Ciw allows users to input their own function to generate service and inter-arrival times. This is done by feeding in a function in the following way:

```
['UserDefined', random.random]
```

5.3.13 Time Dependent Distributions

Similar to adding `UserDefined` functions, Ciw allows for time dependent functions. These are lambda functions that take in a time parameter. Ciw uses the simulation's current time to sample a new service or inter-arrival time:

```
['TimeDependent', time_dependent_function]
```

5.3.14 No Arrivals

If a node does not have any arrivals of a certain class, then the following may be input instead of a distribution:

```
'NoArrivals'
```

Note the lack of square brackets here. Also note that this is only valid for arrivals, and shouldn't be input into the `Service_distributions` option.

5.4 List of Implemented State Trackers for Deadlock Detection

Currently Ciw has the following state trackers:

- *The Naive Tracker*
- *The Matrix Tracker*

5.4.1 The Naive Tracker

The Naive Tracker records the number of customers at each node, and how many of those customers are currently blocked. An example for a four node queueing network is shown below:

```
((3, 0), (1, 4), (10, 0), (8, 1))
```

This denotes 3 customers at the first node, 0 of which are blocked; 5 customers at the second node, 4 of which are blocked; 10 customers at the third node, 0 of which are blocked; and 9 customers at the fourth node, 1 of which are blocked.

The Simulation object takes in the optional argument `tracker` used as follows:

```
>>> Q = ciw.Simulation(N, tracker='Naive')
```

5.4.2 The Matrix Tracker

The Matrix Tracker records the order and destination of blockages in the form of a matrix. Alongside this the number of customers at each node is tracked. The first component, a matrix, lists the blockages from row node to column node. The entries are lists of all blockages of this type, and the numbers within denote the order at which these become blocked. An example for a four node queueing network is shown below:

```
( ( ( (),  (),     (), ()  ),
    ( (),  (1, 4), (), (2) ),
    ( (),  (),     (), ()  ),
    ( (3), (),     (), ()  ) ),
  (3, 5, 10, 9) )
```

This denotes:

- 3 customers at the first node
- 5 customers at the second node
- 10 customers at the third node
- 9 customers at the fourth node

It also tells us the order and destination of the blockages:

- Of the customers blocked, the first to be blocked was at node 2 to node 2
- The second was at node 2 to node 4
- The third was at node 4 to node 1
- The fourth was at node 2 to node 2.

The Simulation object takes in the optional argument `tracker` used as follows:

```
>>> Q = ciw.Simulation(N, tracker='Matrix')
```

5.5 Citing the Library

Please use the following to cite the latest version of the Ciw library:

```
@misc{ciwpython,
  author       = {{{The Ciw library developers}}},
  title        = {Ciw: <RELEASE TITLE>},
  year         = <YEAR>,
  doi          = {<DOI INFORMATION>},
  url          = {http://dx.doi.org/10.5281/zenodo.<DOI NUMBER>}
}
```

To check the details (RELEASE TITLE, YEAR, DOI INFORMATION and DOI NUMBER) please view the Zenodo page for the project. Click on the badge/link below:

5.6 How to Contribute

5.6.1 Contributing

Contributions from anyone are awesome! This may include opening issues, communicating ideas for new features, letting us know about use cases of the library, and code contributions. We would love to recieve your pull requests. Here's a handy guide:

Fork, then clone the repo:

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

Make sure the tests pass (Ciw uses unit & doc testing):

```
python -m unittest discover ciw
python doctests.py
```

We encourage the use of coverage, enusring all aspects of the code are tested:

```
coverage run --source=ciw -m unittest discover ciw.tests
coverage report -m
```

Add tests for your change. Make your change and make the tests pass. Please update the documentation too, and ensure doctests pass.

Push to your fork and submit a pull request!

Some ideas of where to begin:

- Take a look through our issues.
- Open new issues!
- Bug reporting & fixes.
- Code tidying & performance improvements.
- Improvements to the documentation.
- New features.

We look forward to your contributions!

5.7 Glossary

arrival The event in which a customer enters a node.

arrival distribution The distribution that a node's inter-arrival times are drawn from.

blocking Blocking occurs when a customer finishes service and attempts to transition to their destination node, however that node's queueing capacity is full. In this case a blockage occurs where the customer remains at the original node, still holding a server, until space becomes free at the destination node. During this blockage time, the held server is not free to serve any other customers.

closed A queueing network is described as closed is customers cannot arrive from the outside nor completely leave the system.

customer The entities that are being simulated as they wait, spend time in service, and flow through the system. (Interchangeable with *individual*.)

cycle length The length of a cycle of a *work schedule*.

deadlock A state of mutual blocking whereby a subset of customers may never move due to circular blockages.

external arrival An external arrival is an arrival from outside the network. That is all arrivals to nodes where customers have not transitioned from another node.

individual Interchangeable with *customer*.

node A node contains a queue of waiting customers, and a service centre of servers.

open A queueing network is described as open is customers can arrive from the outside and completely leave the system.

queue The part of the node in which customers wait. Waiting follows a first-in-first-out (FIFO) discipline.

queueing network A set of nodes, connected in a network by a transition matrix.

queueing capacity A node's queueing capacity is the maximum number of customers allowed to wait at the node at any time.

restricted A queueing network is described as restricted if there are nodes with limited queueing capacity, and blockages may occur.

server A resource that a customer holds for their period of time in service. The server is also held during blockage times.

server schedule Interchangeable with *work schedule*.

service The period of time which the customer spends with a server. This is the activity which customers wait to begin.

service centre The part of a node in which customers are served. This can contain multiple parallel servers. (Interchangeable with *service station*).

service distribution The distribution that a customer's service times are drawn from.

service station Interchangeable with *service centre*.

traffic intensity The traffic intensity is a measure of how busy the system becomes. For a given node it is defined as the ratio of the mean service time of the mean inter arrival time.

transition matrix A matrix of transition probabilities. The entry in row i of column j, r_{ij}, is the probability of transitioning to node j after service at node i.

warm-up time A period of time at the beginning of a simulation where the data records do not count towards any analysis. This is due to the bias of beginning the simulation from an empty system. Results are only analysed after the system has reached some form of steady-state.

work schedule A schedule of how many servers are present at certain time periods for a given node. Work schedules are cyclic, and so once the *cycle length* has been reached the schedule begins again.

5.8 Change Log

5.8.1 History

v1.1.6 (2018-10-22)

- Fixed bug in which preemptively iterrupted individuals remained blocked once service resampled.
- Fixed bug in which interrupted individuals not removed from interrupted list when restarting service.
- Some performance improvements.
- Improve deadlock detection to check for knots less often.

v1.1.5 (2018-01-11)

- Fixed bug calculating the utilisation of servers.

v1.1.4 (2017-12-12)

- Time dependent batching distributions
- Hard pin requirements versions

v1.1.3 (2017-08-18)

- Replace DataRecord object with namedtuple.
- Number of minor tweaks for speed improvements.

v1.1.2 (2017-07-05)

- Batch arrivals.

v1.1.1 (2017-06-23)

- Server utilisation & overtime.
- Small fixes to docs.
- Testing on Python 3.6.

v1.1.0 (2017-04-26)

- Replace kwargs with actual keyword arguments in ciw.create_network.
- Refactor server schedule inputs (schedules placed inside Number_of_servers instead of as their own keyword).

v1.0.0 (2017-04-04)

- ciw.create_network takes in kwargs, not dictionary.
- Add Sequential distribution.
- Add truncated Normal distribution.
- Refactor inputs for custom PDF.
- Refactor inputs for server schedules.
- Transition matrix now optional for 1 node networks.
- Overhaul of documentation.
- Add CONTRIBUTING.rst.
- Slight improvement of ciw.random_choice.

v0.2.11 (2017-03-13)

- Add ability to simulate until max number of customers have passed arrived/been accepted/passed through the system.

v0.2.10 (2017-03-10)

- Performance improvements.
- Drop dependency on numpy.

v0.2.9 (2017-02-24)

- Allow zero servers.

v0.2.8 (2016-11-10)

- Add option for time dependent distributions.

v0.2.7 (2016-10-26)

- Run tests on Appveyor.
- Check docs build and pip installable on Travis.
- Remove hypothesis cache.

v0.2.6 (2016-10-17)

- Add AUTHORS.rst.
- Add progress bar option.

v0.2.5 (2016-10-06)

- Fix bug that didn't include .rst files in MANIFEST.in.

v0.2.4 (2016-09-27)

- Fixed bug in which priority classes and dynamic classes didn't work together.
- New feature: preemptive interruptions for server schedules.

v0.2.3 (2016-07-27)

- Ability to set seed. More docs. Fixes to tests.

v0.2.2 (2016-07-06)

- Baulking implemented, and minor fixes to order of unblocking.

v0.2.1 (2016-06-29)

- Priority classes implemented.

v0.2.0 (2016-06-20)

- Python 3.4 and 3.5 compatible along with 2.7.
- Data records now kept in list.

v0.1.1 (2016-06-06)

- Ability to incorporate behaviour nodes.
- Data records are now named tuples.

v0.1.0 (2016-04-25)

- Re-factor inputs.
- Simulation takes in a Network object.
- Helper functions to import yml and dictionary to a Network object.
- Simulation object takes optional arguments: deadlock_detector, exact, tracker.
- simulate_until_max_time() takes argument max_simulation_time.

v0.0.6 (2016-04-04)

- Exactness implemented.
- Restructure some features e.g. times_to_deadlock.
- Custom simulation names.

v0.0.5 (2016-03-18)

- State space tracker plug-and-playable.
- Add rejection dictionary.

v0.0.4 (2016-02-20)

- Empirical and UserDefined distributions added.
- Tidy ups.

v0.0.3 (2016-02-09)

- Arrival distributions.
- MMC options removed.
- Fix server schedule bugs.

v0.0.2 (2016-01-06)

- Some kwargs optional.
- Hypothesis tests.
- Minor enhancements.

v0.0.1 (2015-12-14)

- Initial release.

v0.0.1dev (2015-12-14)

- Initial release (dev).

CHAPTER 6

Background

Contents:

6.1 Simulation Practice

Ensuring good practice when simulation modelling is important to get meaningful analyses from the models. This is shown in *Tutorial IV*. A recommended resource on the subject is *[SW14]*. This page will briefly summarise some important aspects of carrying out simulation model analysis.

6.1.1 Performing Multiple Repetitions

Users should not rely on the results of a single run of the simulation due to the intrinsic stochastic nature of simulation. When only running one repetition, users cannot know whether the behaviour of that run is typical, extreme or unusual. To counter this multiple replications must be performed, each using different random number streams. Then analyses on the distribution of results can be performed (for example taking mean values of key performance indicators).

In Ciw, the simplest way of implementing this is to create and run the simulation in a loop, using a different random seed every time.

6.1.2 Warm-up Time

Simulation models often begin in unrealistic circumstances, that is they have unrealistic initial conditions. In Ciw, the default initial condition is an empty system. Of course there may be situations where collecting all results from an empty system is required, but in other situations, for example when analysing systems in equilibrium, these initial conditions cause unwanted bias. One standard method of overcoming this is to use a warm-up time. The simulation is run for a certain amount of time (the warm-up time) to get the system in an appropriate state before results are collected.

In Ciw, the simplest way of implementing this is to filter out records that were created during the warm-up time.

6.1.3 Cool-down Time

If collecting records using the `get_all_records` method, then this will only collect completed records. There may be a need to collect arrival or waiting information of those individuals still in service. In Ciw, we can do this by simulating past the end of the observation period, and then only collect those relevant records that are in the observation period.

In Ciw, the simplest way of implementing this is to filter out records that were created after the cool-down time began.

6.1.4 Example

The example below shows the simplest way to perform multiple replications, and use a warm-up and cool-down time, in Ciw. It shows how to find the average waiting time in an *M/M/1* queue:

```
>>> import ciw
>>> N = ciw.create_network(
...     Arrival_distributions=[['Exponential', 5.0]],
...     Service_distributions=[['Exponential', 8.0]],
...     Transition_matrices=[[0.0]],
...     Number_of_servers=[1]
... )
>>>
>>> average_waits = []
>>> warmup = 10
>>> cooldown = 10
>>> maxsimtime = 40
>>>
>>> for s in range(25):
...     ciw.seed(s)
...     Q = ciw.Simulation(N)
...     Q.simulate_until_max_time(warmup + maxsimtime + cooldown)
...     recs = Q.get_all_records()
...     waits = [r.waiting_time for r in recs if r.arrival_date > warmup and r.
↪arrival_date < warmup + maxsimtime]
...     average_waits.append(sum(waits) / len(waits))
>>>
>>> average_wait = sum(average_waits) / len(average_waits)
>>> average_wait
0.204764...
```

6.2 Notes on Ciw's Mechanisms

6.2.1 General

Ciw uses the *event scheduling* approach *[SW14]* , similar to the three phase approach. In the event scheduling approach, three types of event take place: **A Events** move the clock forward, **B Events** are pre scheduled events, and **C Events** are events that arise because a **B Event** has happened.

Here **A-events** correspond to moving the clock forward to the next **B-event**. **B-events** correspond to either an external arrival, a customer finishing service, or a server shift change. **C-events** correspond to a customer starting service, customer being released from a node, and being blocked or unblocked.

In event scheduling the following process occurs:

1. Initialise the simulation

2. **A Phase**: move the clock to the next scheduled event
3. Take a **B Event** scheduled for now, carry out the event
4. Carry out all **C Events** that arose due to the event carried out in (3.)
5. Repeat (3.) - (4.) until all **B Event** scheduled for that date have been carried out
6. Repeat (2.) - (5.) until a terminating criteria has been satisfied

6.2.2 Blocking Mechanism

In Ciw, Type I blocking (blocking after service) is implemented for restricted networks.

After service, a customer's next destination is sampled from the transition matrix. If there is space at the destination node, that customer will join the queue there. Else if the destination node's queueing capacity is full, then that customer will be blocked. That customer remains at that node, with its server, until space becomes available at the destination. This means the server that was serving that customer remains attached to that customer, being unable to serve anyone else until that customer is unblocked.

At the time of blockage, information about this customer is added to the destination node's `blocked_queue`, a virtual queue containing information about all the customers blocked to that node, and *the order in which they became blocked*. Thus, the sequence of unblockages happen in the order which customers were blocked.

Circular blockages can lead to *deadlock*.

6.2.3 Simultaneous Events

In discrete event simulation, simultaneous event are inevitable. That is two or more events that are scheduled to happen at the same time. However due to the nature of discrete event simulation, these event cannot be carried out computationally at the same time, and the order at which these events are computed can greatly effect their eventual outcome. For example, if two customers are scheduled to arrive at an empty *M/M/1* queue at the same date: which one should begin service and which one should wait?

In Ciw, to prevent any bias, whenever more than one event is scheduled to happen simultaneously, the next event to be computed is uniformly randomly selected from the list of events to be undertaken.

6.3 Kendall's Notation

Kendall's notation is used as shorthand to denote single node queueing systems *[WS09]*.

A queue is characterised by:

$$A/B/C/X/Y/Z$$

where:

- A denotes the distribution of inter-arrival times
- B denotes the distribution of service times
- C denotes the number of servers
- X denotes the queueing capacity
- Y denotes the size of the population of customers
- Z denotes the queueing discipline

For the parameters A and B, a number of shorthand notation is available. For example:

- M: Markovian or Exponential distribution
- E: Erlang distribution (a special case of the Gamma distribution)
- C_k: Coxian distribution of order k
- D: Deterministic distribution
- G / GI: General / General independent distribution

The parameters X, Y and Z are optional, and are assumed to be ∞, ∞, and First In First Out (FIFO) respectively. Other options for the queueing schedule Z may be SIRO (Service In Random Order), LIFO (Last In First Out), and PS (Process Sharing).

Some examples:

- $M/M/1$**:**
  - Exponential inter-arrival times
  - Exponential service times
  - 1 server
  - Infinite queueing capacity
  - Infinite population
  - First in first out
- $M/D/\infty/\infty/1000$**:**
  - Exponential inter-arrival times
  - Deterministic service times
  - Infinite servers
  - Infinite queueing capacity
  - Population of 1000 customers
  - First in first out
- $G/G/1/\infty/\infty/$**SIRO:**
  - General distribution for inter-arrival times
  - General distribution for service times
  - 1 server
  - Infinite queueing capacity
  - Infinite population
  - Service in random order
- $M/M/4/5$**:**
  - Exponential inter-arrival times
  - Exponential service times
  - 4 servers
  - Queueing capacity of 5
  - Infinite population

  - First in first out

6.4 Code Structure

Ciw is structured in an object orientated way:

Ciw consists of 3 types of objects, Core, Input, and Optional:

Core:

- Simulation
- Arrival Node
- Exit Node
- Node
- Server
- Individual

Input:

- Network
- Service Centre
- Customer Classe

Optional:

- State Tracker
- Deadlock Detector

6.5 Other Computer Simulation Options

Ciw is just one way in which simulations of queueing networks can be carried out. Here is a list of other options that may be of interest, both commercial and open source:

- SimPy
- SIMUL8
- AnyLogic
- Khronos
- NetSim
- Arena

6.6 Bibliography

- Link to Example Jupyter Notebooks
- genindex

- search

Bibliography

[SW14] 19. Robinson. *Simulation: the practice of model development and use.* Palgrave Macmillan, 2014.

[WS09] 23. Stewart. *Probability, markov chains, queues, and simulation.* Princeton university press, 2009.

Index