2.1. I verified my simulation code by creating validation checks that grab a few values out of the simulation at different steps (such as spawn station numbers and return bike numbers) and then calculate the expected values of these values using the initial distributions from the files. These tests can be shown below:

```
# simulation validation checks
self.stats["validation"] = {
    "spawn_stations": {
        "expected": {
            **{
                i: self.p[i] * self.stats["total_riders"] for i in range(self.m)
            },
            "Limited": max(0, self.lambda_ * self.end_time - self.n),
        },
        "simulated": self.stats["invocations"]["event__spawn_rider"][
            "ret_vals"
        ],
    },
    "return_bike": {
        "expected": {
            j: np.sum(
                [
                    self.q[i, j]
                    * (
                        self.stats["invocations"]["event__take_bike"]["args"][
                            "i"
                        ].get(i, 0)
                        - self.stats["invocations"]["event__return_bike"][
                            "overtime"
                        ]["args"]
                        .get("i", {})
                        .get(i, 0)
                    )
                    for i in range(self.m)
                ]
            )
            for j in range(self.m)
        },
        "simulated": self.stats["invocations"]["event__return_bike"][
            "ret_vals"
        ],
    },
}

for validation_check in self.stats["validation"]:
    self.stats["validation"][validation_check]["error"] = {
        k: self.stats["validation"][validation_check]["simulated"].get(k, 0)
        - self.stats["validation"][validation_check]["expected"][k]
        for k in self.stats["validation"][validation_check]["expected"]
    }
```

These results here calculate the expected values and the pulls the simulated values from the simulation. The expected values are calculated in different ways. For the spawn stations, it is simply the probability of the station times the total riders. For the return bike value, it is the sum over transition probabilities that get to the station time the number of bikes taken from the source station (for each transition probability). There is an additional term to account for bikes returned after the 24 hour period.

Finally, the errors between the simulated and the expected values are calculated. Looking at output errors, they are not that high compared to the total number of riders, so it is about expected. These runs were done with a simplified configuration seen in the demo.json and simple.json config files. These configs are located in the config folder.

Further, I did some runs with the simpler configurations while changing certain parameters like mean rate, number of riders, bikes per stations, etc. and got results that made intuitive sense.

2.2. The mean probability of a successful rental is approximately 0.9798 over 100 runs of the simulation with the default parameters noted in the pdf. The 90% confidence interval for this value is (0.97861, 0.98097). There are more values calculated from the raw runs of the simulations shown in the default.json results in ci_results directory. They are also shown on the right.

Probability of Successful Rental:

```
"mean": 0.9797887951588585,
"std": 0.007083054020294788,
"se": 0.0007118737137920052,
"n": 100,
"confidence": 0.9,
"confidence_interval": [
  0.978606806340292,
  0.9809707839774251
]
```

The average waiting time has mean over 100 runs of the simulations is 8.55367 again with default parameters noted in the pdf. The 90% confidence interval for this value is (7.99121, 9.07612). Again, there are more values calculated form the raw data of the runs of the simulations.

Average Waiting Time:

```
"mean": 8.533671151747333,
"std": 3.250670195981135,
"se": 0.32670464718971143,
"n": 100,
"confidence": 0.9,
"confidence_interval": [
  7.991213644930615,
  9.07612865856405
]
```