# Optimization Introduction

For the Activity Based Costing Analysis (ABCA) model, a black-box optimization was run to have a baseline to compare the brain’s results to. This was accomplished using AnyLogic’s Optimization experiment, which uses OptQuest’s optimization engine. The experiment works by finding an optimal configuration of the model’s parameter values (the ranges of which are specified by the user) resulting in the best possible configuration. The “best” solution is found by either minimizing or maximizing an objective value - a numerical representation of the model’s performance.

There are two additional features that were used in the experiment. The first is requirements, which are a condition evaluated at the end of the model. If not fulfilled, the current solution proposed by the optimization engine (i.e., the set of input parameter values) is considered “infeasible” or an invalid solution. The second feature is replications. With replications, each solution chosen by the optimization engine is tried a specified number of times. The resulting objective value (defining how “good” the solution was) is taken to be the mean of the objective values from all the replications.

Using replications is important with stochastic models, such as the ABCA model, since using the same set of input parameters in multiple model runs can have different outcomes. Another important point is that if *any* of the replications don’t meet the specified requirements, the entire solution is considered infeasible and no further replications are tried.

In short, the optimization experiment works as follows: a configuration of parameters is chosen by the OptQuest optimization engine. This configuration is then run multiple times, determined by the specified number of replications – 100 in this case. If any of the runs result in the requirement check failing, then the iteration is deemed infeasible, the result will be discarded, and the optimization engine moves on to the next configuration to try. On the other hand, if all 100 runs succeed in passing the requirement, the objective value is calculated as the mean of each run’s objective value - total cost per product in this case. Based on this, the optimization engine chooses another configuration to try. It repeats this cycle until a given number of configurations have been tried or it detects the objective value stops significantly improving (depending on how the experiment is setup).

# Optimization Configuration and Results

The list below details the aforementioned experiment properties, in addition to a few others related to the simulation. Note that the ranges for the parameters are listed with a minimum and maximum; for discrete values (integers), a step size is listed.

* Objective: Minimize total cost per product
* Parameters - the ranges are taken from the original model:
  + Resource A capacity: Min=1, Max=20, Step=1
  + Resource B capacity: Min=1, Max=20, Step=1
  + Mean process delay: Min=1, Max=12
  + Conveyor speed: Min=1, Max=15
* Model run time: 6 months
* Requirement condition: The system was never overloaded during the entire simulation run (specifically that the `exceededCapacity` flag was never set to true)
* Replications: 100
* Before each simulation run: Randomly vary the arrival rate between 0.5 and 2.0 (uniformly)
* Automatic stop enabled: This tells the optimization engine an unlimited number of iterations (configurations) are able to be tried, but to automatically stop when the objective value stops significantly improving.

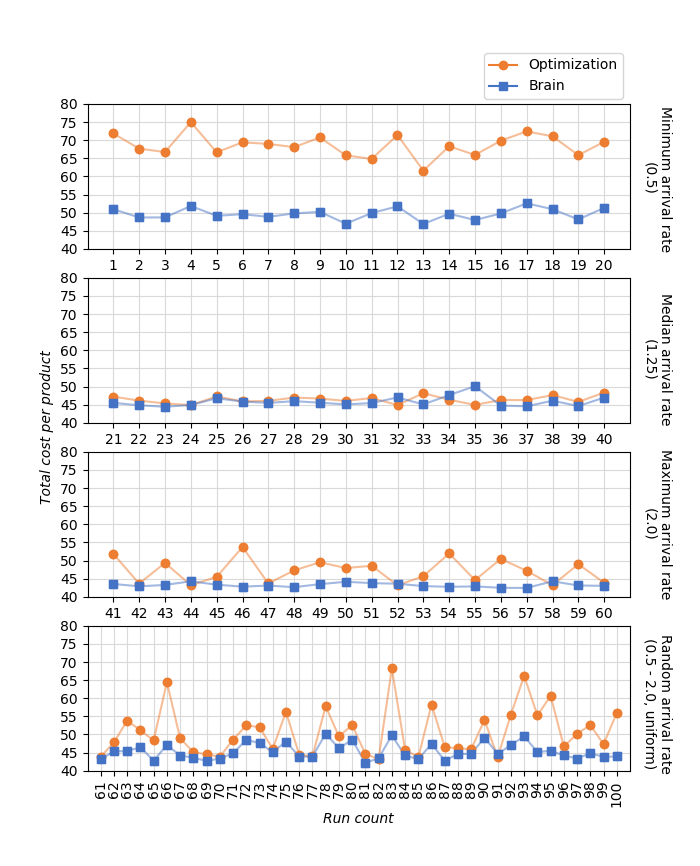
After running the optimization, the best configuration (i.e., the one that most minimized the cost per product, compared to the other configurations tried) consisted of each resource having 5 capacity, a mean delay time of 2.297, and a conveyor speed of 0.1. The objective value for this configuration had a score of 48.34. In other words, the mean cost per product across the 100 runs of the specified configuration (with varying arrival rates) was $48.34.

# Results of the Brain and Optimization Comparison

In both experiments, 100 simulation runs were executed with a fixed, but varied seed to the model. This enabled the results to be directly comparable. The specifics of how this was performed and automated can be read about in the appendix.

For the first 20 runs, the arrival rate was set to the minimum possible value (0.5). The next 20 runs were at the median (1.25), and the 20 after that were at the maximum (2.0). The last 40 runs had the arrival rate at a random value between the minimum and maximum, chosen uniformly.

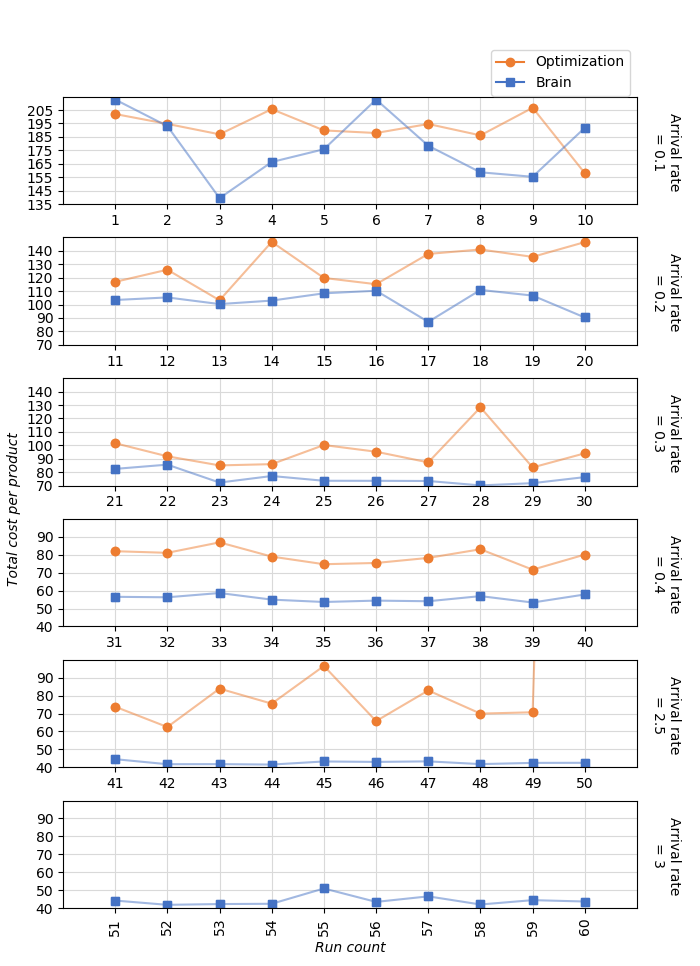
From running both experiments, the average cost per product for the optimization was $52.69 and the average for the brain was $45.89, representing a 14.82% improvement. The individual scores of the four sets are broken down into its their own graphs, shown in the image below.



As it can be seen, the results from the brain are overall more stable compared to the optimization. In addition, the optimization performs sub-optimally the further away from the median the arrival rate is.

A final interesting aspect to consider is how each performs when the target of variation (arrival rate) goes beyond the range each was originally trained on (which was in the interval [0.5, 2.0]). Six sets of 10 simulations were run for the arrival rates 0.1, 0.2, 0.3, 0.4, 2.5, and 3.0 – the first four being below the minimum arrival rate trained on and the last two being above the maximum. The results can be seen in the graphs below, showing the brain continued to perform superiorly across most of all the runs.

Note that due to wide variations in cost per product, different Y-axis scales were required. Additionally, the optimization was unable to handle one of the runs when the arrival rate was set to 2.5, nor was it able to handle any of the runs when the arrival rate was set to 3.0. In these 11 runs, the system had become overloaded.

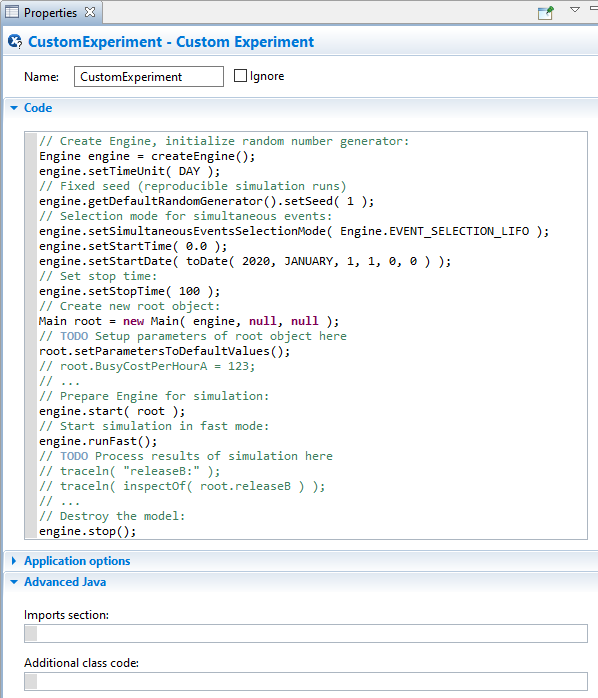


With these results in mind, it should be noted that there is room for improvement in the testing method. By performing multiple optimizations across more narrow subsets of the range, the performance of the optimization compared to the brain may be improved.

# Appendix: Details of Method used to Compare the Brain and Optimization Results

One way to compare the brain and optimization results would be to run a Monte Carlo experiment (one for each method) with the model’s pseudo-random number generator (PRNG) set to “Random seed (unique simulation runs)”. Statistics for the cost per product could be observed across many runs, such as the mean, deviation, etc. These results would be technically valid, but it would be more informational to have a side-by-side comparison. Doing a side-by-side comparison using this method would not be valid since each simulation run would have its own unique PRNG seed. This would not be possible with a unique seed in each simulation run, as two runs – even with the same arrival rate – would not have the same arrival and processing times (both of which are taken from a distribution).

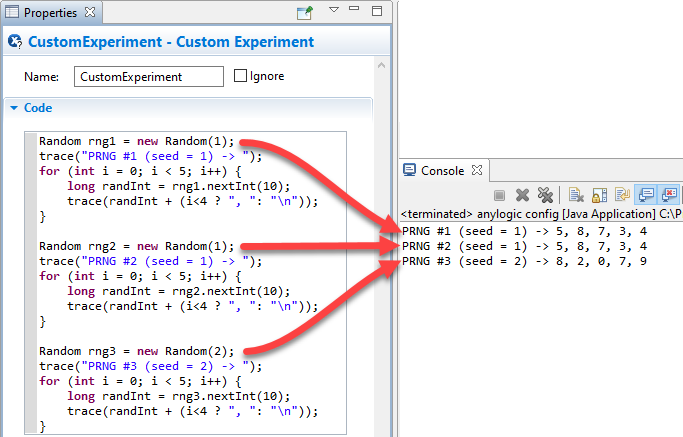
AnyLogic’s experiments allows for the PRNG seed to be a fixed, specified value - this would make for a valid side-by-side comparison. For testing various arrival rates multiple times, having to manually change the seed for each run becomes a tedious process. To resolve this, AnyLogic offers the “Custom Experiment”. Custom experiments are written in pure Java code but give you full control of the setup for the engine and the model’s execution.



The boilerplate code that comes with a new Custom Experiment

What this means is that features of Java can be utilized in automating the experiment. At the top of the code field, an instance of the `Random` Java class can be created with a fixed seed that will be used to generate the pseudo-random seeds for each of the model runs. A for-loop can be used to run the simulation as many times as needed. Before each simulation run, the engine’s PRNG will have its seed be fixed, but pseudo-random number. The benefit of this is that in both custom experiments (one for the optimization and another for the brain), if the `Random` object defined at the start of the code has a fixed seed in both experiments, both experiments will run with the same stream of “random” seeds.

To put it differently, reference the image below. Three `Random` objects are created – the first two with the same seed (1) and the third with its seed incremented by one (2). Each of these objects have five integers between 0 and 9 generated and printed to the console, which can be seen to the right of the image. As you can see, the first two `Random` objects generate the same five random numbers. Although the seed of the third object is only different by one, its output is significantly different. You can imagine `rng1` being placed in the optimization custom experiment and `rng2` being placed in the brain custom experiment.



Using this “random”, but known and reproducible PRNG seed generator allowed both the brain and optimization custom experiments be executed with the same set of seeds. This enables the individual runs to be directly comparable.

Each run was set to run for the same length of time as during training (six months). The inputs to the optimization’s experiment was fixed. To get the inputs for the brain’s experiment, a Java function was written to execute a REST query to the exported brain, passing the arrival rate in the body’s data.