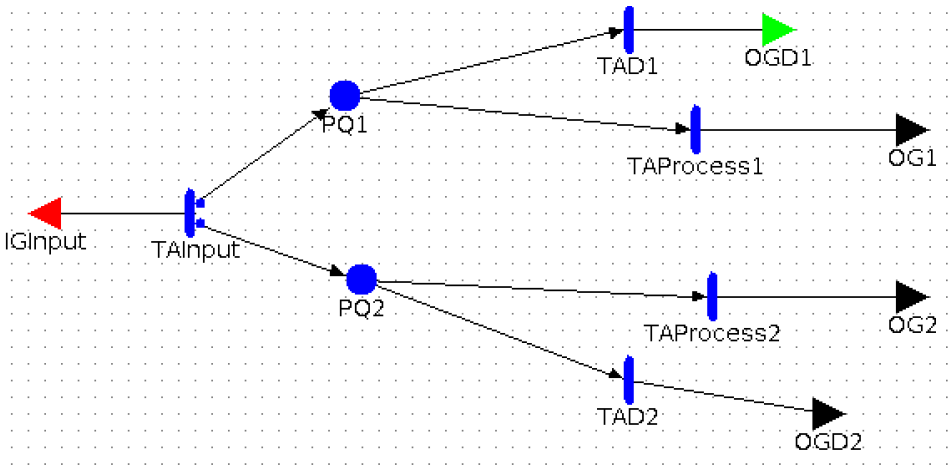
Nima Mohammadi (95013021) - nima.mohammadi@ut.ac.ir
Prof. Movaghar
Assignment #3

Assignment #3 - MM1K-DPS

The analytical solution are provided by employing iterative steady solver over the corresponding SAN model depicted below using Möbius software:



The Möbius models reside in `Mobius_Models/` folder, archived and named `MobExp.tar.gz` and `MobFix.tar.gz`.

`IGInput` checks the sum of markings of `PQ1` and `PQ2` and refrain from enabling `TAInput` in case they are filled to the max. Timed activities `TAProcess1` and `TAProcess2` consume tokens as dictated by DPS. Rate of `TAProcess2` is shown below for the exponential case:

```
double n1 = PQ1->Mark();
double n2 = PQ2->Mark();
return (2.0*n1) / (n1 + n2 *2.0)) * ProcessRate;
```

Similarly it goes for `TAProcess1`.

Code snippet below is calculates the rate for timed activity `TAD2` with fixed waiting time:

```
double n1 = PQ1->Mark();
```

```
    double n2 = PQ1->Mark();
    double mu2 = ((n2 * 2.0) / (n1 + n2 * 2.0)) * ProcessRate;
    return mu2 / (exp((mu2 * MissRate)/n2)-1);
```

And as for the exponential waiting time:

```
    double n2 = PQ2->Mark();
    return n2/MissRate;
```

The same goes for `TAD1`.

The simulation keeps a triplet of values for each customer that arrives (and not blocked). This triplet **(remaining_service_time, remaining_waiting_time, class)** denotes an individual customer of the specified class at a specific time. At each iteration, the simulation progresses for **next_event_t** amount of time unit. It then subtracts from the **remaining_service _time** of each customer, as every job receives some share of processing time, at rates below for each class:

```
total_share = class_weights[0] * class_current_customers[0] + class_weights[1] *
class_current_customers[1];
remaining_service_time -= class_weights[class(customer)] / total_share;
```

It also subtracts **next_event_t** from **remaining_waiting_time.**

upcoming_job_t is time until the arrival of the next job, that is drawn from an Exponential distribution parameterized by Lambda.

next_event_t determines the simulation time at next iteration. Its value is dynamically calculated by taking the minimum of **upcoming_job_t**, minimum of **remaining_service_time** and **remaining_waiting_time*current_waiting_count**, which correspond to the arrival of next customer, completion of one job, or desertion of queue by a customer, respectively. We choose the duration of **next_event_t** to be the maximum time it can go without the necessity to handle an event.

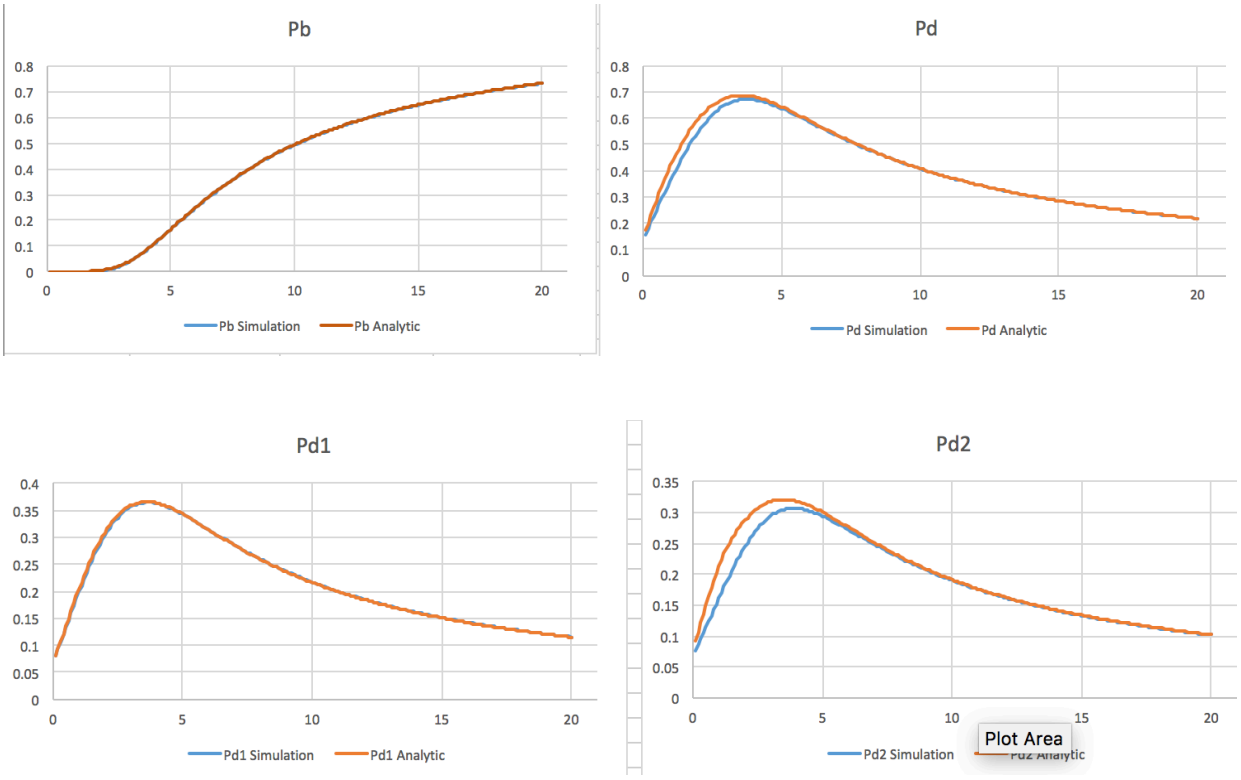
sim_counts is the total number of customers who arrive and determines the length of the simulation. **blocked** and **deserted**, respectively, denote the number of customers who was rejected from the queue, as it was full, and the number of customers who left the queue as the

duration they could stay and wait has passed without fully being serviced. **deserted_1** and **deserted_2** also keep the number of deserted customers of each class. **theta_type=0** corresponds to fixed-time waiting time while **theta_type=1** indicates waiting time to be drawn from an exponential distribution.

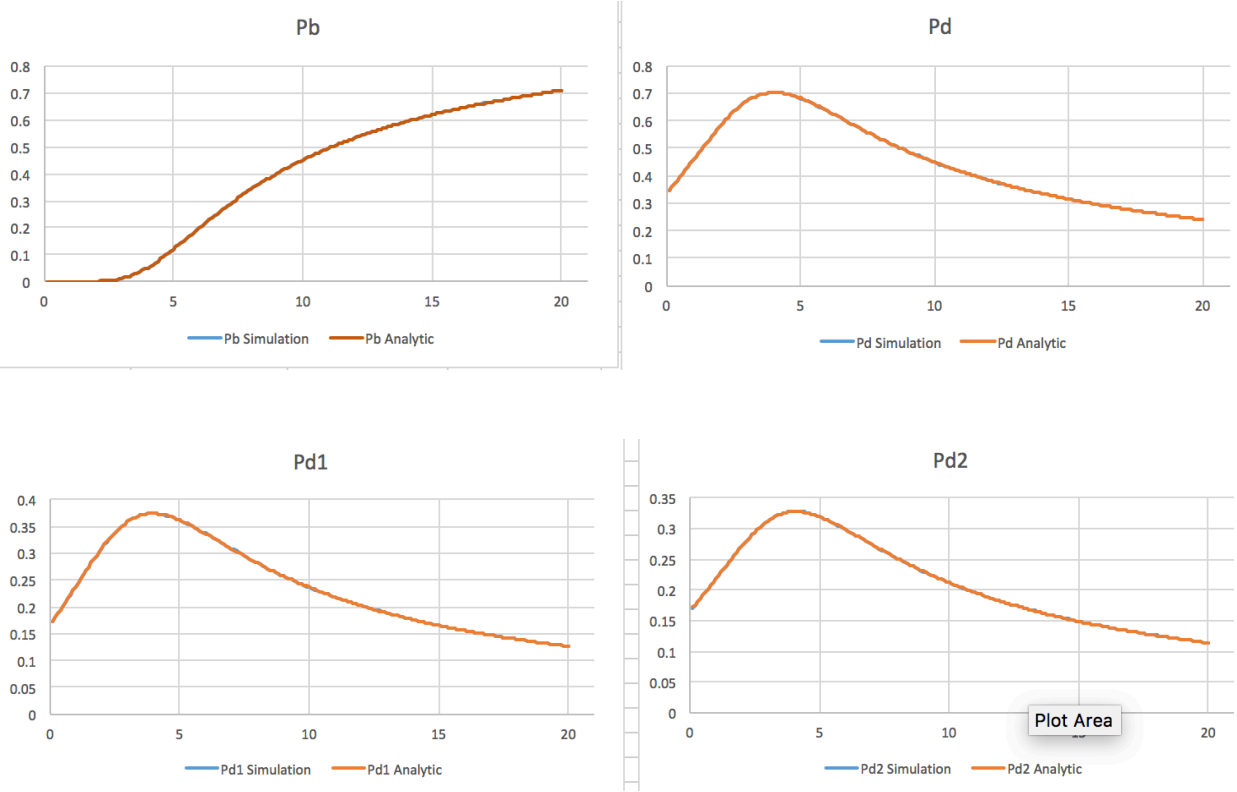
The simulation supports warming up, that is to neglect preliminary results to get more accurate statistics from the simulation.

Given adequate amount of the length of the simulation and warm-up period, and as can be seen from the Excel files and the plots, the simulation reaches a very acceptable error and the curves for analytical and simulated probabilities abundantly match and only differ with a very negligible error.

Fixed Waiting Time



Exponential Waiting Time



In both cases probability of blocked customers increases with increasing the arrival rate, hence the strictly increasing curve for P_b. On the other hand P_d increases at first since there are more non-blocked customers reaching the processor. Though it decrease after lambda=~5 as the queue get congested and more customers get blocked.

The code is implemented in C++ (C++11 to be exact). It has two main functions

```
vector<double> queue_simulator(double lambda, double mu, double theta, int K, int
theta_type, int sim_counts, int warmup=100000)

vector<double> closed_form(double lambda, double mu, double theta, int K, int
theta_type)
```

which correspond to simulation and analytical computation of the queue, respectively.

Running "**make**" compiles the code into two executable files, namely "**mm1k_runparams.o**" and "**mm1k_analysis.o**". Running "**make run**" runs the first executable (i.e. "**mm1k_runparams.o**").

* **mm1k_runparams.o:** The first one reads the parameters, Theta and Mu, from "**parameters.conf**" file and reports P_b, P_d, P_{d1} and P_{d2} as Lambda assumes three different values for Theta being drawn of two different random distributions (Fixed and Exp). Running "make run" executes this executable file.

* **mm1k_rangestudy.o:** The second executable also reports P_b, P_d, P_{d1} and P_{d2} for the exponential and fixed arrival distribution and outputs two .CSV files.

The actual code for the simulator and the analytical computations resides in "MM1K/ mm1k.hpp".

As for the absolute and relative error, these are calculated as below:

```
abserr = abs (analval - simval)
relerr = abserr / analval
```

The codes were tested on macOS Sierra with gcc [Apple LLVM version 8.0.0 (clang-800.0.42.1)]. **mm1k_analysis.o** (consisting 40 simulations) took ~15 minutes to run on my laptop.