

# Homework 9

2025-10-23

## Boilerplate

```
library(printr)      # pretty print for Rmd
library(tidyr)
library(knitr)
library(FrF2)

# set seed for reproducibility
set.seed(42)
```

## Question 12.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a design of experiments approach would be appropriate.

An example situation where DoE might be appropriate is at my job. We are currently moving our analytics from PowerBI to a Databricks hosted Dash app. This gives us a lot of freedom to make changes to patterns that were locked by the previous architecture. As we are making changes it could be useful to see which elements are useful, UI elements, defaults, etc. Doing a partial factorial approach would allow us to determine not only individual factors, but the interaction between them without having to go through so many iterations.

## Question 12.2

To determine the value of 10 different yes/no features to the market value of a house (large yard, solar roof, etc.), a real estate agent plans to survey 50 potential buyers, showing a fictitious house with different combinations of features. To reduce the survey size, the agent wants to show just 16 fictitious houses. Use R's `FrF2` function (in the `FrF2` package) to find a fractional factorial design for this experiment: what set of features should each of the 16 fictitious houses have? Note: the output of `FrF2` is "1" (include) or "-1" (don't include) for each feature.

Given the situation of 10 factors and 16 houses, this sets `nFactors = 10` and `nRuns = 16`. This means we obviously have 16 unique combinations and we'll use the remainder for replication. The `FrF2` package only allows a whole cohort of replications, so the remaining two can be assigned a house at random.

```
doe <- FrF2(
  nFactors = 10,
  nRuns=16,
  replications = 3,
  default.levels = c(1,0)
)

half <- ceiling(nrow(doe) / 2)

# Add explicit row numbers
left <- cbind(Row = 1:half, doe[1:half, ])
right <- cbind(Row = (half + 1):nrow(doe), doe[(half + 1):nrow(doe), ])

# Combine with separator
doe_split <- cbind(left, "|" = rep("|", half), right)

kable(doe_split, booktabs = TRUE, format = "latex", row.names = FALSE)
```

Row	A	B	C	D	E	F	G	H	J	K		Row	A	B	C	D	E	F	G	H	J	K
1	1	1	1	1	0	0	0	0	1	0		25	1	0	0	1	1	1	0	0	1	0
2	1	1	0	1	0	1	1	0	0	1		26	0	0	1	0	0	1	1	0	1	1
3	0	0	0	0	0	0	0	0	0	0		27	0	0	0	1	0	0	0	1	1	1
4	1	1	1	0	0	0	0	1	0	1		28	0	1	1	0	1	1	0	0	0	0
5	0	1	1	0	1	1	0	0	0	0		29	0	0	0	0	0	0	0	0	0	0
6	0	0	1	1	0	1	1	1	0	0		30	0	1	0	0	1	0	1	0	1	1
7	0	1	1	1	1	1	0	1	1	1		31	0	1	0	1	1	0	1	1	0	0
8	0	1	0	0	1	0	1	0	1	1		32	1	1	1	1	0	0	0	0	1	0
9	0	0	0	1	0	0	0	1	1	1		33	0	1	1	0	1	1	0	0	0	0
10	1	0	0	1	1	1	0	0	1	0		34	0	0	0	1	0	0	0	1	1	1
11	1	0	1	0	1	0	1	1	1	0		35	0	1	0	0	1	0	1	0	1	1
12	1	1	0	0	0	1	1	1	1	0		36	1	0	1	0	1	0	1	1	1	0
13	1	0	0	0	1	1	0	1	0	1		37	0	1	0	1	1	0	1	1	0	0
14	0	0	1	0	0	1	1	0	1	1		38	0	0	0	0	0	0	0	0	0	0
15	1	0	1	1	1	0	1	0	0	1		39	1	0	0	0	1	1	0	1	0	1
16	0	1	0	1	1	0	1	1	0	0		40	0	0	1	1	0	1	1	1	0	0
17	0	1	1	1	1	1	0	1	1	1		41	1	1	1	0	0	0	0	1	0	1
18	1	0	0	0	1	1	0	1	0	1		42	0	0	1	0	0	1	1	0	1	1
19	1	0	1	1	1	0	1	0	0	1		43	0	1	1	1	1	1	0	1	1	1
20	1	1	1	0	0	0	0	1	0	1		44	1	1	0	1	0	1	1	0	0	1
21	1	1	0	0	0	1	1	1	1	0		45	1	1	0	0	0	1	1	1	1	0
22	1	0	1	0	1	0	1	1	1	0		46	1	1	1	1	0	0	0	0	1	0
23	0	0	1	1	0	1	1	1	0	0		47	1	0	0	1	1	1	0	0	1	0
24	1	1	0	1	0	1	1	0	0	1		48	1	0	1	1	1	0	1	0	0	1

## **Question 13.1**

For each of the following distributions, give an example of data that you would expect to follow this distribution (besides the examples already discussed in class).

Examples of these distributions:

- a. Binomial - My mother and I play a card game online called cribbage, I am slightly better than her in our win loss rate (52%-48%). The number of wins/losses over a series of  $n$  games should follow a binomial distribution.
- b. Geometric - Building on the binomial example because of the link between the two, the geometric distribution predicts the probability of either of us being on a hot streak and winning a sequence of games.
- c. Poisson - A Poisson distribution can be used to model many events occur over a fixed interval. One example is the number of accidents in a time interval.<sup>1</sup>
- d. Exponential - As was stated in the lecture, the Poisson and exponential distributions are two ways of looking at the same type of data, the exponential distribution models the time between events, so the time between accidents would be an example.
- e. Weibull - Searching for an example that was not survival analysis (that includes warranty claims). An interesting example that I found is particle distribution in grinding operations<sup>2</sup>. I find this charming for some reason.

---

<sup>1</sup><https://www.quora.com/What-is-the-real-life-example-of-Poisson-distribution>

<sup>2</sup>[https://www.researchgate.net/publication/232700851\\_Modelling\\_fragment\\_size\\_distribution\\_using\\_two-parameter\\_Weibull\\_equation](https://www.researchgate.net/publication/232700851_Modelling_fragment_size_distribution_using_two-parameter_Weibull_equation)

## **Question 13.2**

In this problem you can simulate a simplified airport security system at a busy airport. Passengers arrive according to a Poisson distribution with  $\lambda_1 = 5$  per minute (i.e., mean interarrival rate  $\beta_1 = 0.2$  minutes) to the ID/boarding-pass check queue, where there are several servers who each have exponential service time with mean rate  $\beta_2 = 0.75$  minutes. [Hint: model them as one block that has more than one resource.] After that, the passengers are assigned to the shortest of the several personal-check queues, where they go through the personal scanner (time is uniformly distributed between 0.5 minutes and 1 minute). Use the Arena software (PC users) or Python with SimPy (PC or Mac users) to build a simulation of the system, and then vary the number of ID/boarding-pass checkers and personal-check queues to determine how many are needed to keep average wait times below 15 minutes. [If you're using SimPy, or if you have access to a non-student version of Arena, you can use  $\lambda_1 = 50$  to simulate a busier airport.]

Note: I drew heavily on SimPy's carwash example<sup>3</sup> for the code to generate the ID check portion of the process and a Medium post by Aaron Janeiro Stone<sup>4</sup> to model the scanner portion of the simulation. To integrate these two pieces I utilized Github Copilot.

---

<sup>3</sup><https://simpy.readthedocs.io/en/latest/examples/carwash.html>

<sup>4</sup><https://medium.com/swlh/simulating-a-parallel-queueing-system-with-simpy-6b7fcb6b1ca1>

## Simulation Code

### Setup

To store simulation data I wrapped the individual components outlined below in a `run_simulation` function as seen below. This allows me to run many individual runs with different numbers of ID counters, scanning lines, and seeds.

```
import itertools
import logging
import simpy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

logging.basicConfig(level=logging.WARN, format="%(message)s", force=True)
logger = logging.getLogger(__name__)

RANDOM_SEED = 42
SIM_TIME = 2500
SCANNER_LOW = 0.5
SCANNER_HIGH = 1.0
BETA1 = 0.2
BETA2 = 0.75

def run_simulation(
    num_counters,
    num_scanners,
    random_seed=RANDOM_SEED,
    sim_time=SIM_TIME,
    log_level="INFO"
):
    rng = np.random.default_rng(random_seed)
    data = []
    logger.setLevel(log_level)

    ### Simulation code

    env = simpy.Environment()
    env.process(
        setup(
            env,
            num_counters=num_counters,
            beta1=BETA1,
            beta2=BETA2,
            num_scanners=num_scanners,
            scan_low=SCANNER_LOW,
            scan_high=SCANNER_HIGH,
        )
    )

    env.run(until=sim_time)

    return pd.DataFrame(data)
```

## Passenger arrival and ID check

As stated above the `passenger` and setup functions and the `Checkpoint` class drew heavily from SimPy's carwash example with the major difference being changing the process of generating cars changed from a uniform random process to a Poisson/exponential. This initializes the script by creating passengers and feeds them into a single queue that splits into multiple ID check agents.

```
class Checkpoint:
    """A security checkpoint has a limited number of ID counters
    to check passengers in parallel.

    Passengers have to request one of the counters. When they get one,
    they can start the ID checking process and wait for it to finish (which
    takes ``check_time`` minutes).
    """

    def __init__(self, env, num_counters, beta2):
        self.env = env
        self.counter = simpy.Resource(env, num_counters)
        self.beta2 = beta2

    def check_id(self, passenger):
        """The ID checking process. It takes a ``passenger`` process
        and simulates checking their ID."""
        rate = rng.exponential(self.beta2)
        logger.debug(
            f"t={self.env.now:7.4f} | {passenger} | ID check duration: {rate:6.4f}"
        )
        yield self.env.timeout(rate)

    def passenger(env, name, cp, scanners, generation_time):
        """The passenger process (each passenger has a ``name``) arrives
        at the checkpoint (``cp``) and requests an ID counter. After ID check
        they may go to scanners if provided.
        """
        arrive_time = env.now
        logger.debug(
            f"t={env.now:7.4f} | {name} | Arrives at checkpoint"
            f" | Queue length: {NoInSystem(cp.counter)}"
        )

        with cp.counter.request() as request:
            yield request
            wait_time = env.now - arrive_time
            logger.debug(
                f"t={env.now:7.4f} | {name} |"
                " Begins ID check | Wait time: {wait_time:6.4f}"
            )
            yield env.process(cp.check_id(name))
            logger.debug(f"t={env.now:7.4f} | {name} | Completes ID check")

        env.process(scanners.queue(name, generation_time))
```

```

def setup(env, num_counters, beta1, beta2, num_scanners, scan_low, scan_high):
    # Create a passenger arrival process at the checkpoint.
    checkpoint = Checkpoint(env, num_counters, beta2)
    scanners = Scanners(env, num_scanners, scan_low, scan_high)

    passenger_count = itertools.count()

    # Create 4 initial passengers
    for _ in range(4):
        passenger_id = next(passenger_count)
        logger.debug(
            f"t={env.now:7.4f} | Passenger {passenger_id}"
            " | Generated at start"
        )
        env.process(
            passenger(
                env,
                f"Passenger {passenger_id}",
                checkpoint,
                scanners,
                env.now
            )
        )

    # Create more passengers while the simulation is running
    while True:
        arrival_delay = rng.exponential(beta1)
        yield env.timeout(arrival_delay)
        passenger_id = next(passenger_count)
        logger.debug(
            f"t={env.now:7.4f} | Passenger {passenger_id} |"
            f" Generated after {arrival_delay:6.4f} delay"
        )
        env.process(
            passenger(
                env,
                f"Passenger {passenger_id}",
                checkpoint,
                scanners,
                env.now
            )
        )

```

## Scanner

The logging process and `Scanner` class are heavily based off Aaron Janeiro Stone's Medium article on parallel queuing. This takes a stream of passengers and splits them into parallel queues where new passengers choose the shortest line when they start the process.

```
def NoInSystem(R: simpy.Resource):
    """Total number of customers in the resource R"""
    # Support resources that may not have put_queue/users attributes
    return max(0, len(getattr(R, "put_queue", [])) + len(getattr(R, "users", [])))

class Scanners:
    """Passenger scanners helper.

    Maintains a list of scanner resources and exposes a Customer process
    that a arriving passenger can use: choose shortest queue, wait, then
    take a uniform scanning time between wait_low and wait_high.
    """

    def __init__(self, env, num_scanners, wait_low, wait_high):
        self.env = env
        self.num_scanners = num_scanners
        self.wait_low = wait_low
        self.wait_high = wait_high
        self.scanners = [simpy.Resource(env) for _ in range(self.num_scanners)]

    def queue_lengths(self):
        return list(map(NoInSystem, self.scanners))

    def queue(self, name, generation_time):
        """Process for a passenger to join a scanner queue and be scanned.

        Uses the scanner list and the class's environment so multiple Customer
        processes (one per passenger) run in parallel and form queues per scanner.
        """
        arrive = self.env.now
        Qlength = self.queue_lengths()

        logger.debug(
            f"t={self.env.now:.4f} | {name} | Arrives at scanner | "
            f" Queue lengths: {Qlength}"
        )
    
```

```

# choose first scanner with minimal queue length
choice = min(range(len(Qlength)), key=lambda i: Qlength[i])
with self.scanners[choice].request() as req:
    yield req
    wait = self.env.now - arrive
    logger.debug(
        f"t={self.env.now:7.4f} | {name} | Begins scanning at scanner {choice} |"
        f" Wait time: {wait:6.4f}"
    )
    time_in_line = rng.uniform(self.wait_low, self.wait_high)
    yield self.env.timeout(time_in_line)
    total_time = self.env.now - generation_time
    logger.debug(
        f"t={self.env.now:7.4f} | {name} |"
        f" Completes scanning at scanner {choice} |"
        f" Scan duration: {time_in_line:6.4f}"
    )
    logger.info(
        f"t={self.env.now:7.4f} | {name} | Exits system |"
        f" Total time in process: {total_time:6.4f}"
    )
    data.append({
        "name": name,
        "generation_time": generation_time,
        "total_time": total_time
    })

```

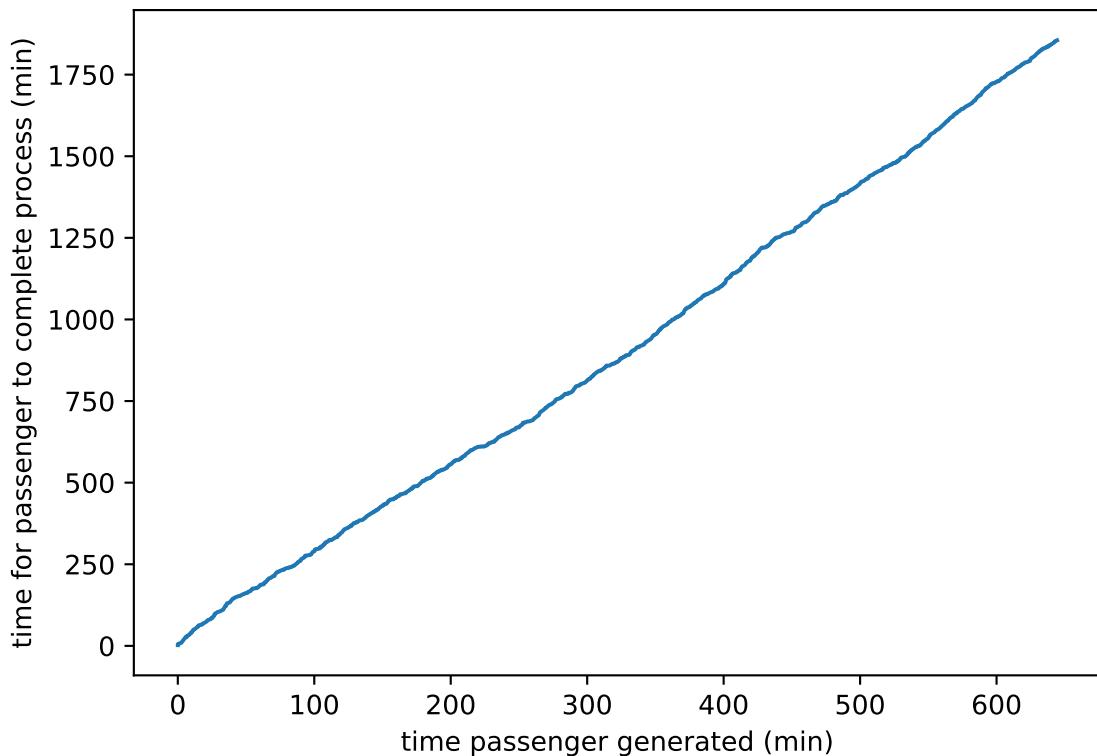
## Steady state search

The first thing we need to do is find the behavior at steady state. Because for example, having one counter and one scanner never converges because passengers arrive faster than they can get through the scanner because  $\beta_1 \ll \beta_2$ .

```
diverging_example = run_simulation(
    num_counters=1,
    num_scanners=1,
)

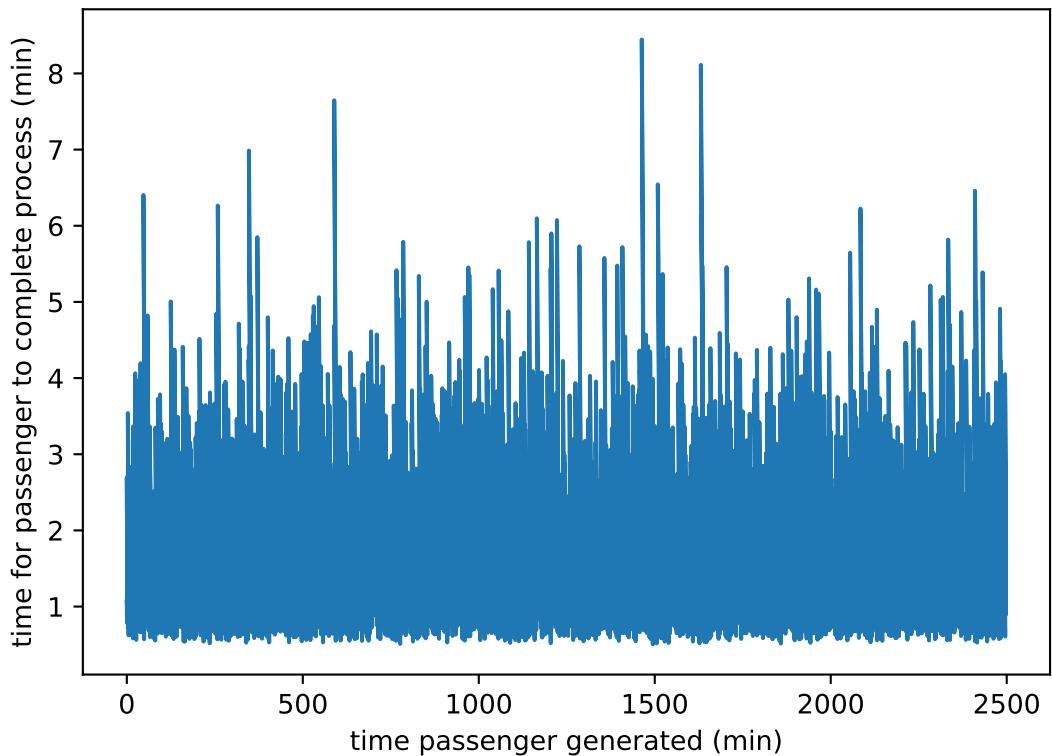
ax = diverging_example.plot(
    x="generation_time",
    y="total_time",
    legend=False
)

ax.set(
    xlabel="time passenger generated (min)",
    ylabel="time for passenger to complete process (min)"
)
```



Whereas with ten counters and ten scanners the process only takes a few minutes.

```
converging_example = run_simulation(  
    num_counters=10,  
    num_scanners=10,  
)  
  
ax = converging_example.plot(  
    x="generation_time",  
    y="total_time",  
    legend=False  
)  
  
ax.set(  
    xlabel="time passenger generated (min)",  
    ylabel="time for passenger to complete process (min)"  
)
```



Note: I am not actually running this or the next cell in the notebook because otherwise it would take 50 minutes to run each time I wanted to make a formatting change to the notebook. Also note that in my actual code this and the next cell are actually one loop so I do not need to hold onto data only to distill it down to a few numbers. I just felt that explaining in two chunks was easier.

So there must be a point where we transition between these two behaviors. To find the boundary of stability I ran a range of ID counters and scanners with many seeds to determine when the number of passengers was stable.

```
counters_test = range(3,7)
scanners_test = range(3,7)
seeds_test = [12345, 8675309] + list(range(100))

merged = []

for num_counters, num_scanners, random_seed in \
    itertools.product(counters_test, scanners_test, seeds_test):
    run = run_simulation(
        num_counters=num_counters,
        num_scanners=num_scanners,
        random_seed=random_seed,
        sim_time=1200,
    ).assign(
        num_counters=num_counters,
        num_scanners=num_scanners,
        random_seed=random_seed,
    ).sort_values("generation_time")

    merged.append(run)

merged = pd.concat(merged)
```

Then I took the slope of the third quartile and fourth quartile of passengers and fit a line to the generation time vs total processing time for each of them. This type of analysis is common in Markov chain Monte Carlo sampling.

```
stationary_test = []

for (num_counters, num_scanners, random_seed), df in merged.groupby([
    "num_counters", "num_scanners", "random_seed"
]):
    num_passengers = len(run)
    half = num_passengers // 2
    quarter = half // 2

    df_half = run.iloc[half:,:]
    df_third_quarter = run.iloc[:quarter,:]
    df_fourth_quarter = run.iloc[quarter:,:]

    second_half_slope, _, _, _, _ = stats.linregress(
        df_half["generation_time"], df_half["total_time"]
    )
    second_half_mean = df_half["total_time"].mean()

    stationary_test.append(
        {
            "num_counters": num_counters,
            "num_scanners": num_scanners,
            "random_seed": random_seed,
            "num_completed_passengers": num_passengers,
            "second_half_slope": second_half_slope,
            "second_half_mean": second_half_mean,
        }
    )

stationary_test = pd.DataFrame(stationary_test)
```

So we can see how often these simulations converge to a stable state by looking for an approximately zero slope indicating the line is stable and determine what fraction of simulations showed stable behavior.

```
stable_region = (
    (
        stationary_test
        .set_index([
            "num_counters",
            "num_scanners",
            "random_seed"
        ])["second_half_slope"].abs() < 0.125
    )
    .groupby([
        "num_counters", "num_scanners"
    ]).mean().reset_index()
)

stable_region.pivot_table(
    index="num_counters",
    columns="num_scanners",
    values="second_half_slope",
    aggfunc="sum"
)

## num_scanners      3      4      5      6
## num_counters
## 3                  0.0  0.0  0.0  0.0
## 4                  0.0  1.0  1.0  1.0
## 5                  0.0  1.0  1.0  1.0
## 6                  0.0  1.0  1.0  1.0
```

To have a stable line we need to have four ID counters and four scanner lines. Let's see what the wait times are for acceptable lines are:

```
stable_region.query(
    "second_half_slope == 1"
).merge(
    stationary_test,
    on=["num_counters", "num_scanners",],
).pivot_table(
    index="num_counters",
    columns="num_scanners",
    values="second_half_mean",
    aggfunc="mean",
    fill_value=0
).round(2)

## num_scanners      4      5      6
## num_counters
## 4                  5.70  4.59  4.19
## 5                  3.29  1.98  1.86
## 6                  3.06  1.79  1.64
```

So in all cases the average wait time is below 15 minutes. I'm not sure if that means there's a bug with my simulation or something else. I've gone over my code several times and everything looks like it's set up correctly.

## Busier airport simulation

Using the same methodology as Steady state search, I scanned for fully stable parameters. I did add additional code for early stopping of duplicate `num_counters`, `num_scanners`. That cut down the parameter scan. Using this methodology, I generated the below. It also has short processing time.

ID counters	Scanner lines	processing time (minutes)
39	39	2.20
39	40	2.06
39	41	2.00
40	39	2.02
40	40	1.87
40	41	1.80
41	39	1.93
41	40	1.79
41	41	1.72