Gabe^3
GabeL@virginia.edu
https://gabriel-vzh2vs.github.io/SYS3062-Website/
UVA's Systems and Information Engineering

Feburary 24th, 2026

SYS 3062: Lab 5

Uncertainity and Output Analysis: The Game

Lab 5: The Context

The Source Material

Based on Eliyahu Goldratt's novel *The Goal* (Theory of Constraints).

- **Scenario:** A group of Boy Scouts hiking (a manufacturing line).
- **The Problem:** The line moves only as fast as the slowest hiker (bottleneck) + random fluctuations.

The Simulation: A manufacturing line with N workers.

- **Process Time:** Random (originally a die roll, here Gamma distributed).
- **Buffers:** Finite storage space between workers (b).
- **Objective:** Maximize Throughput (y) by tuning the system parameters.

GabeL@virginia.edu

The Dice Game: Physical vs. Digital

In *The Goal*, Alex Rogo uses a line of Boy Scouts to demonstrate manufacturing physics.

The Physical Game

- **Workers:** The Scouts.
- **Product:** Matches moved from bowl to bowl.
- **Process Time:** Rolling a Die (1 to 6).
- **Dependency:** You cannot move matches you don't have.

The Digital Model

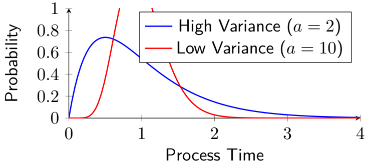
- **Workers:** `simpy.Process`.
- **Product:** Integer tokens.
- **Process Time:** Gamma Distribution (Continuous version of a die).
- **Dependency:** `yield buffer.get()`.

The Uniform Distribution (The Die):

- Mean $= 3.5$. Variance is fixed.
- *Problem:* Real workers don't have hard "min/max" caps like a die.

The Gamma Distribution (Γ):

- We normalize the Mean to $\mathbf{1.0}$.
- We control Variance using shape parameter $\mathbf{a}$.
- **Natural Boundaries:** Time cannot be negative, but can occasionally be very long (tail events).



SimPy Strategy: Blocking & Starving

SimPy handles the logic of Blocking (Output Full) and Starving (Input Empty) automatically via 'yield input.get()' and 'yield output.put()'.

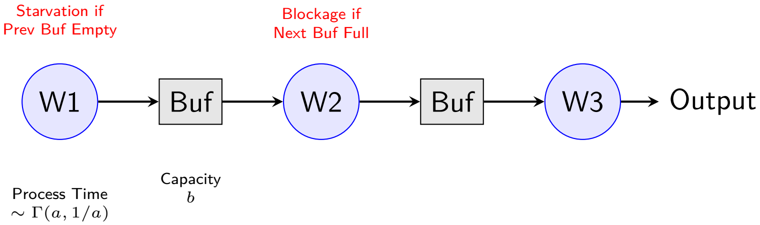
```python
def worker_process(env, name, input_buffer,
output_buffer, shape_a):
    while True:
        # Wait until item is available in input buffer
        item = yield input_buffer.get()
        time_to_process = np.random.gamma(shape_a, 1.0 /
        shape_a)
        yield env.timeout(time_to_process)

        # Wait until space is available in output buffer
        yield output_buffer.put(item)
```

GabeL@virginia.edu

The Assembly Line Model

A linear sequence of Workers separated by Finite Buffers.



GabeL@virginia.edu

Implementation Strategy: In SimPy, we use ‘simpy.Store(capacity=b)‘.

```
def setup_line(env, n, b):
    buffers = []
    # 1. Source (Infinite)
    buffers.append(simpy.Store(env, capacity=inf))
    # 2. Intermediate Buffers (The Kanban Limit)
    for i in range(n - 1):
        buffers.append(simpy.Store(env, capacity=b))
    # 3. Sink (Infinite)
    buffers.append(simpy.Store(env, capacity=inf))
    return buffers
```

Note: If Worker 1 tries to put an item into Buffer 2, and Buffer 2 has ‘b‘ items, Worker 1 freezes (Blocking). This is how Kanban limits overproduction.

Measuring Success: Throughput (y)

In the Dice Game, raw speed doesn't matter. Only finished goods matter.

Throughput Calculation

$$y = \frac{\text{Total Items in Sink}}{\text{Total Simulation Time}}$$

Setup Requirements:

1. **Warm-up:** Pre-fill buffers with 2 items to avoid empty pipe startup bias.
2. **Duration:** Run for 1,000 time units to allow statistical fluctuations to average out.
3. **Seeding:** Use "np.random.seed(42)" to ensure your dice rolls are reproducible.

Now, it's your turn to implement this Queuing System in Python or using Excel! Good Luck!

GabeL@virginia.edu

How do we analyze a system: Experimental Design (DOE)

The Problem

A simulation is a "Black Box." We put inputs in, and get an output out. But we don't inherently know *which* input caused the output change.

The Solution: experimental design, specifically the sub-method of factorial design We systematically vary the input parameters (Factors) in a grid.

Lab 6 Factors:

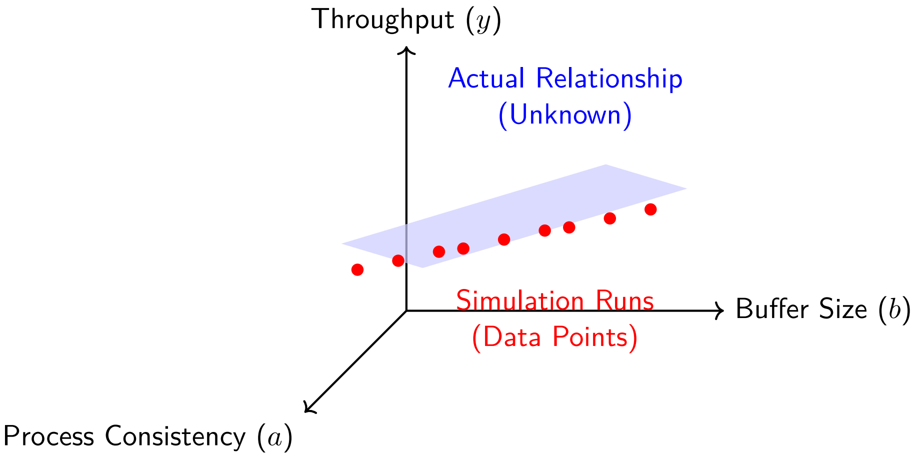
- n **(Line Length):** 11 levels $(5, 6, \ldots, 20)$.
- b **(Buffer Size):** 10 levels $(1, \ldots, 10)$.
- a **(Variance):** 6 levels $(2, 3, \ldots, 20)$.
- σ **(Worker Skill):** 5 levels $(0 \ldots 0.1)$.

$$\text{Total Runs} = 11 \times 10 \times 6 \times 5 = \mathbf{3,300} \text{ Simulations}$$

GabeL@virginia.edu

Visualizing the Design Space

Instead of guessing random points, we create a structured grid (Hypercube) of scenarios.



Experimental Design: Choosing the Points

How do we choose which input combinations to run?

Full Factorial Design (L^k)

Test **every possible combination** of all factors.

- **Pros:** Captures detailed behavior and complex interactions.
- **Cons:** Curse of Dimensionality. (10 levels^4 factors $= 10,000$ runs).

2^k Factorial Design

Test only the **Low (-1)** and **High (+1)** levels of each factor.

- **Pros:** Extremely efficient ($2^4 = 16$ runs). Great for "Screening" (finding what matters).
- **Cons:** Assumes linearity. Cannot see curvature (peaks/valleys) inside the range.

GabeL@virginia.edu

What if 2^k is still too expensive? (e.g., 20 factors $\rightarrow$ 1,000,000 runs).

3. Fractional Factorial (2^{k-p})

We run only a structured subset (e.g., a Half-Fraction).

- **Logic:** "Sparsity of Effects." High-order interactions (e.g., Factor A $\times$ B $\times$ C $\times$ D) are usually negligible.
- **Trade-off:** Aliasing. We might confuse Main Effects with Interaction Effects.

Example: In a 2^{3-1} design, we run 4 experiments instead of 8. We lose some detail but save 50% of the computing cost.

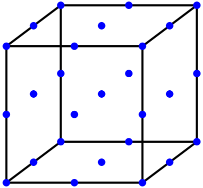
GabeL@virginia.edu

Visualizing Factorial Spaces

Comparing a 3-Factor Experiment ($k = 3$).

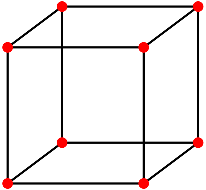
Full Factorial (3^3)

(27 Runs)



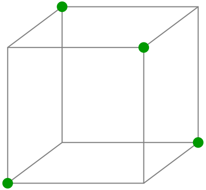
2^k **Factorial (2^3)**

(8 Runs)



Fractional (2^{3-1})

(4 Runs)



Full-Factorial Experiment

We run a **Full Factorial Experiment** (3,300 runs) to understand how four parameters affect throughput (y).

Input Factors (X):

- n **(Line Length):** 5 to 20 workers.
- b **(Buffer Size):** 1 to 10 items.
- a **(Process Consistency):** Shape parameter (Higher a = Less Variance).
- σ **(Worker Variance):** Skill variation between workers.

Statistical Model (OLS):

$$y \sim 1 + n + \log(1+b) \times \log(a) + \sigma$$

We hypothesize an interaction between Buffer Size (b) and Process Consistency (a).

GabeL@virginia.edu

Summarizing the Results

These conclusions are backed by a *meta-model* on top of the full-factorial simulation.

Key Insights:

- **Buffers Help:** Increasing buffer size (b) significantly improves throughput, but with diminishing returns (Logarithmic).
- **Variance Kills:** High process variance (Low a) destroys throughput, even with buffers.
- **Line Length Hurts:** Longer lines (n) generally have lower throughput due to "Dependency Accumulation".

GabeL@virginia.edu

How can we interpret complex systems: Metamodels?

Definition

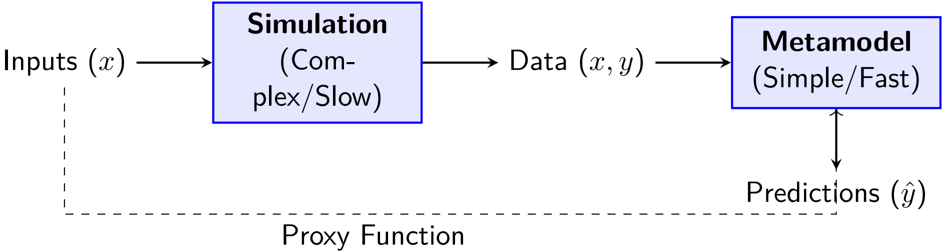
A metamodel is a "Model of a Model."

- **The Simulation:** A complex, slow, stochastic representation of reality (e.g., Our SimPy Factory).
- **The Metamodel:** A simple, fast, mathematical approximation of the simulation (e.g., Linear Regression).

Why use them?

- **Speed:** Running the simulation 1,000 times takes an hour. Evaluating the regression equation takes milliseconds.
- **Optimization:** It is easier to find the maximum of a smooth equation ($y = \beta x$) than a noisy simulation.
- **Insight:** Coefficients reveal *sensitivity* (which factors matter most).

We do not replace the simulation; we use it to *train* the metamodel.



OLS Regression as a Metamodel

Once we have the data, we use Ordinary Least Squares (OLS) or a Generalized Linear Model (GLM) to fit a mathematical equation to our simulation results.

Why OLS?

1. **Quantification:** It tells us exactly *how much* y changes when we increase b by 1 unit.
2. **Significance:** It tells us if a factor matters at all (P-Values).
3. **Metamodeling:** The regression equation becomes a Fast Proxy. We can predict throughput without running the slow simulation again.

The Lab 6 Model:

$$\hat{y} = \beta_0 + \beta_1 n + \beta_2 \log(1 + b) + \beta_3 \log(a) + \dots$$

GabeL@virginia.edu

Regression Output (Statsmodels)

We use a GLM (in OLS mode here) Regression to quantify the effects.

Factor	Coef	P-Value	Interpretation
Intercept	0.669	0.000	Baseline Throughput
n (Length)	-0.001	0.181	Minimal negative impact
$\log(1+b)$ (Buffer)	**0.105**	**0.000**	**Strong Positive Effect**
$\log(a)$ (Consistency)	**0.106**	**0.000**	**Strong Positive Effect**
Interaction $b:a$	-0.035	0.008	Buffers matter less if variance is low
σ (Worker Var)	-0.749	0.000	Strong Negative Effect

Conclusion

To fix the factory: **Reduce Variance** first, then **Add Buffers**.

Interpreting the Regression Table

How to read the `statsmodels` output from Lab 6:

Component	Example Value	Meaning
Coefficient	$+0.105$ ($\log b$) -0.749 (σ)	**Magnitude.** Increasing buffer size raises throughput. Increasing worker variance *hurts* throughput.
P-Value	0.000 0.181 (n)	**Certainty.** This effect is real, not random noise. This factor (Line Length) might not matter statistically.
R-Squared	0.735	**Fit.** Our model explains 73.5% of the system's behavior.

The Interaction Term

We found a significant interaction: $\log(1+b) \times \log(a)$. *Translation: Buffers are extremely useful when variance is high (low a), but less useful when the process is already stable.*

GabeL@virginia.edu

Feedback Form

Throughout the course, I may ask for your feedback on:

- **Course Materials**
- **Lecture & Lab**
- **Assignments**

GabeL@virginia.edu

Action Items

GabeL@virginia.edu

First, I will thank you all again for coming to lab at 5 pm on a Tuesday!

And now, you need to do the following tasks:

- **Submit** your Python file to Gradescope by Spring Break!
- **Have** fun on spring break!

Your exams will be graded sooner than you think!™