Analyzing Simulation Output

Quiz Study Guide — Chapter 5

Manuel D. Rossetti, PhD P.E.

How to Use This Deck

This deck is a **checklist**, not a summary. Every slide is phrased as something you should be able to *state*, *derive*, or *compute* without looking it up.

- **Question formats:** multiple choice, true/false, matching, fill-in-the-blank, multi-slot fill-in-the-blank, and numeric answers.
- The **Common Traps** slides are built from the actual wrong answers — read them last.

The spine of this chapter

Within-replication versus across-replication statistics; finite-horizon versus infinite-horizon analysis; and how to compare two or more system configurations.

What This Chapter Covers

Capturing output

- Within- versus across-replication statistics
- `Response`, `TWResponse`, `Counter`, `IndicatorResponse`
- Reporters, collectors, traces, the KSL database
- The `ModelElementObserver` lifecycle

Analyzing output

- Finite horizon: independent replications
- Infinite horizon: warm-up, replication-deletion, batch means
- Comparing systems: CRN, paired-t, MCB
- Controls, RV parameters, scenarios

Objectives — Capturing Output

You should be able to:

- Define within-replication and across-replication statistics and say how each is formed
- Classify data as tally or time-persistent and pick the right KSL class
- State what each output-capture class does and when it fires
- Order the `ModelElementObserver` lifecycle methods correctly
- Name the KSL database tables and what each holds

Objectives — Analyzing Output

You should be able to:

- Distinguish finite-horizon from infinite-horizon simulation and name the method for each
- Explain initialization bias and how a warm-up period mitigates it
- Describe the KSL batch-means algorithm and Schmeiser's recommendation
- Write $\mathrm{Var}(\bar{X}_1 - \bar{X}_2)$ under independent sampling and say when CRN helps
- Apply the Bonferroni inequality and explain what MCB buys you

Within versus Across Replication

- **Within-replication statistics** are collected **during** a replication.
- **Across-replication statistics** are formed from the **final values** of the within-replication statistics — **one observation per replication**.

Tally data is a sequence of **equally weighted** observations associated with the duration or interval an object is in a state, observed at the end of that state. It is collected with `Response`.

Time-persistent data is collected with `TWResponse`.

The Response Classes

Class	Role
`Response`	Tally statistics on values observed at discrete event times
`TWResponse`	Time-weighted statistics on time-persistent (state) variables
`Counter`	Incremented/decremented during a replication; **reset to 0** at replication start
`IndicatorResponse`	A `Response` subclass recording 1.0/0.0 from a boolean predicate on another response
`TWResponseFunction`	A `TWResponse` derived by applying a function (e.g. x/numWorkers) to another `TWResponse`

In the pallet example, $P\{\text{total time} > 480\}$ is captured with an `IndicatorResponse` whose predicate is `{ x -> x >= 480.0 }`.

Two Lifecycle Methods That Matter

- `initialize()` — called once at the **beginning** of each replication, conceptually at $t = 0^-$; the recommended place to schedule initial events.
- `replicationEnded()` — called automatically **infinitesimally before the end** of each replication; the recommended place to capture final within-replication observations **before the accumulators are cleared**.

Observations made inside `replicationEnded()` **are still within-replication observations** and can be tallied by a `Response`.

The Observer Lifecycle — In Order

Table 1: The ordering is a guaranteed question — learn it as a sequence

Method	When it fires
`beforeExperiment()`	Once, before the first replication and any events
`beforeReplication()`	Before each replication; the calendar is cleared **after** this action
`initialize()`	Start of each replication, after the calendar is cleared
`warmUp()`	At the warm-up event, clearing the statistical accumulators
`replicationEnded()`	Just before the end of each replication
`afterReplication()`	After each replication
`afterExperiment()`	Once, after all replications

Capturing Output

Class	Purpose
`SimulationReporter`	Text, MarkDown, and LaTeX summary reports, at a user-specified confidence level
`ReplicationDataCollector`	Observer retaining **end-of-replication values** in arrays via `allReplicationDataAsMap`
`ResponseTrace`	Writes a per-change trace of an observed response to a file
`KSLDatabaseObserver`	Persists every within- and across-replication statistic to a relational database
`WelchFileObserver`	Captures observations across replications for warm-up analysis

The KSL Database

`autoCSVReports = true` produces **two CSV files**. For anything richer, attach a `KSLDatabaseObserver`.

Table or view	Contents
`SIMULATION_RUN`	Run metadata: experiment name, replications, length, warm-up
`MODEL_ELEMENT`	The model element hierarchy per run: name, ID, parent ID, class type
`WITHIN_REP_STAT`	Per-replication ending statistics for `Response` and `TWResponse`
`WITHIN_REP_COUNTER_STAT`	Per-replication counter statistics
`ACROSS_REP_STAT`	Across-replication summary statistics
`PW_DIFF_WITHIN_REP_VIEW`	Pairwise within-replication differences $(A - B)$, where A has the higher simulation ID

Keeping More Than One Experiment

> ⚠ Trap
>
> Re-running a simulation with the **same name** in the same output directory **deletes and recreates** the database, losing the previous results.

To store several experiments in the same database during one program execution, change the **`experimentName`** between calls to `simulate()` — **not** the simulation name.

```
model.experimentName = "Three Workers"
model.simulate()
model.experimentName = "Four Workers"
model.simulate()
```

Finite Horizon

A **finite-horizon (terminating)** simulation has a well-defined ending time or ending condition that clearly demarks the end of the simulation.

The classical method of analysis is the **method of independent replications**.

The `AcrossReplicationHalfWidthChecker` is a `ModelElementObserver` attached to a `Response` that calls `endSimulation()` once the across-replication half-width on that response meets a desired value.

Infinite Horizon and Initialization Bias

Within-replication queueing data violates **both** standard IID assumptions:

- It is strongly **positively autocorrelated** — not independent.
- Early observations are influenced by the initial conditions — not identically distributed.

Initialization bias is the bias in steady-state estimators caused by the (often non-representative) initial conditions.

Mitigation: a **warm-up period**, set by `lengthOfReplicationWarmUp`. Banks et al.'s rule of thumb: $T_e \geq 10\,T_w$.

> ⚠ Trap
>
> Even after a warm-up, within-replication observations are **still not independent**. And "the system has reached steady state" is **not** endorsed as a correct reading.

Welch Plots

The classes live in **`ksl.observers.welch`**:

Class	Role
`WelchFileObserver` / `WelchDataFileCollector`	Attach to a response; gather per-replication observations to disk
`WelchDataFileAnalyzer`	Read the files; compute per-row averages and cumulative averages
`WelchPlot`	Render the plot in a browser or to a file

For a **time-persistent** variable, first **discretize** the data into equally spaced intervals (e.g. 10 time units) so each interval average is one observation.

Batch Means

The KSL algorithm (after Kelton et al.):

- Start with $k = 20$ batches.
- When **40** batches have formed, collapse back to **20** by pairwise averaging — which **doubles** the batch size.
- The number of batches is thus maintained between 20 and 39.

Add `StatisticalBatchingElement` to the model via `model.statisticalBatching()`.

Schmeiser: use roughly **10 to 30 batches**; there is little benefit beyond 30.

If a warm-up analysis used n_0 replications of length T_e with warm-up T_w, a reasonable single-replication starting point is run length $n_0 T_e$ with warm-up $n_0 T_w$.

Comparing Two Systems

Under **independent sampling**, with $\hat{D} = \bar{X}_1 - \bar{X}_2$:

$$\mathrm{Var}(\hat{D}) = \frac{\sigma_1^2}{n} + \frac{\sigma_2^2}{n}$$

Common Random Numbers reduces $\mathrm{Var}(\bar{X}_1 - \bar{X}_2)$ whenever there is **positive correlation** between the paired outputs.

- Set `resetStartStreamOption = true` so all streams reset before each experiment.
- The KSL **advances each stream to the next substream at the end of every replication** by default, which assists synchronization.
- A `MultipleComparisonAnalyzer` built from **independent** data may show a noticeably **larger** standard deviation of the differences than one built from **CRN-paired** data.

Multiple Comparisons

- **Bonferroni:** for c intervals with individual levels α_i, the overall error probability satisfies $\alpha_E \le \sum_i \alpha_i$. Set $\alpha_i = \alpha_E / c$.
- **MCB** builds only k intervals of the form $\theta_i - \max_{j \neq i} \theta_j$ instead of all $\binom{k}{2}$ pairwise ones. For $k = 6$: **15 pairwise versus 6 MCB**.
- **Nelson-Matejcik two-stage Bonferroni** second-stage sample size:

$$n = \max\left\{ n_0, \left\lceil \frac{t^2 \hat{S}^2}{\delta^2} \right\rceil \right\}, \qquad t = t_{1-\alpha/(k-1),\, n_0 - 1}$$

- Looking for a **maximum**, system i is **not statistically different from the best** when $\hat{\theta}_i - \hat{\theta}_{\hat{i}} + \delta > 0$.
- `db.multipleComparisonAnalyzerFor(listOf(exp1, exp2), responseName)` is the easy construction path.

Controlling Model Inputs

Construct	Role
`KSLControl`	Annotation marking a property as a **controllable input**, exposed via `Model.controls()`
`RVParameterSetter`	Generic access to every random variable parameter, with flat keys like `ProcessingTimeRV.mode`
`Scenario`	A model + an input map (control or RV-parameter names to doubles) + a unique scenario name
`ScenarioRunner`	Executes a list of `Scenario` instances and stores results to a KSL database
`MultipleComparisonAnalyzer`	Consumes the scenario results to compute MCB and related comparisons

Three Details About Controls

- The control **key** is `elementName.propertyName`. For `numWorkers` on a `PalletWorkCenter` named "PWC", the key is **`PWC.numWorkers`**.
- `RVParameterSetter` keys are `RVName.parameterName`, such as **`ProcessingTimeRV.mode`**.
- An `RVParameterSetter` change takes effect on the model **only after `applyParameterChanges(model)`** is invoked.

The payoff of `ScenarioRunner` plus a `KSLDatabase` is that many configurations are captured automatically under unique experiment names, and can then be queried — for MCB analysis, say — **without manually re-running the models**.

Drills

This chapter's question bank has no numeric items — these are practice problems built from the formulas and rules it tests in other formats. Expect the quiz to test the same relationships, not these exact numbers.

1. With $c = 10$ confidence intervals sharing an overall error probability of $\alpha_E = 0.05$, what common α should each interval use?
2. For $k = 6$ systems, how many intervals does full pairwise comparison require? How many does MCB require?
3. The KSL batch-means algorithm starts with how many batches, collapses at how many, and what does the collapse do to the batch size?
4. A warm-up analysis used $n_0 = 10$ replications of length $T_e = 5000$ with $T_w = 1000$. What run length and warm-up would you use for a single-replication batch-means analysis?
5. Banks et al.'s rule of thumb gives what minimum run length for a warm-up of $T_w = 200$ time units?

Common Traps

- Across-replication statistics come from the **final** within-replication values — one per replication, not from running totals.
- `IndicatorResponse` is a **`Response`** subclass, not a `TWResponse` subclass.
- Observations in `replicationEnded()` **are** still within-replication observations.
- Changing the **simulation name** does not preserve prior database results — change the **`experimentName`**.
- A warm-up removes *initialization bias*; it does **not** make the observations independent.
- CRN needs **positive** correlation between paired outputs to help.
- MCB gives k intervals, not $\binom{k}{2}$.
- The batch-means algorithm holds the count between **20 and 39**, not at a fixed 10 or 100.

Appendix — Drill Answers

1. $\alpha = \alpha_E / c = 0.05/10 = \mathbf{0.005}$
2. Pairwise: $\binom{6}{2} = \mathbf{15}$. MCB: $\mathbf{6}$.
3. Start at **20**; collapse at **40** back to **20** by pairwise averaging, which **doubles** the batch size.
4. Run length $n_0 T_e = 10 \times 5000 = \mathbf{50{,}000}$; warm-up $n_0 T_w = 10 \times 1000 = \mathbf{10{,}000}$.
5. $T_e \geq 10\, T_w = 10 \times 200 = \mathbf{2000}$ time units.