Lab 2 Report

ECE 322
Diepreye Charles-Daniel
Student Number: 1766907

Introduction

The purpose of this lab is to apply two black-box testing strategies—Extreme Point Combination (EPC) and Weak n×1 testing—to the provided software programs: Drone.jar and RemoteCar.jar.

The Drone Autopilot System models a 3-leg flight plan, where the three legs (x_1, x_2, x_3) must each satisfy input constraints $(0 \leq x_i \leq 100)$ and a total constraint $(x_1 + x_2 + x_3 \leq 100)$. The goal of testing is to verify correct input validation and program logic under boundary and extreme conditions.

Extreme Point Combination (EPC) testing generates test cases using the extreme boundary values (min, max, just below/above boundaries) of each variable in combination, to ensure full boundary interaction coverage.

Weak n×1 testing focuses on boundaries of one variable at a time while keeping others at nominal values, to reduce the number of test cases while still achieving boundary coverage.

Task 1 – Autopilot System (3 legs of flight)

1. The Drone Autopilot System takes three input values corresponding to distances (or durations) for three legs of a flight. Each input represents one directional leg of the drone's travel.

   Inputs: x_1, x_2, x_3 (representing flight legs)
   Expected conditions:
   - Each x_i must be between 0 and 100 (inclusive).
   - The total $x_1 + x_2 + x_3$ must not exceed 100.

   The program outputs either "Success!" when the input satisfies these constraints, or an error message when any rule is violated. The main goal of testing is to confirm that all validation rules are correctly implemented for all edge and interaction cases.

2. Identified Errors

#	Description	Example Input	Expected	Actual	Type
1	**Missing validation for $x_2 < 0$**	(0, −1, 0)	Error	Success!	Logical bug
2	**Missing validation for $x_2 < 0$**	(100, −1, 0)	Error	Success!	Logical bug
3	**Typo in error message**	(−1, 0, 0)	"Invalid argument"	"Invald argument"	Output formatting

- The program fails to reject negative x_2 inputs.
- All other negative checks (x_1, x_3) and upper bounds are enforced correctly.
- Minor typo present in displayed error message.

Possible root causes

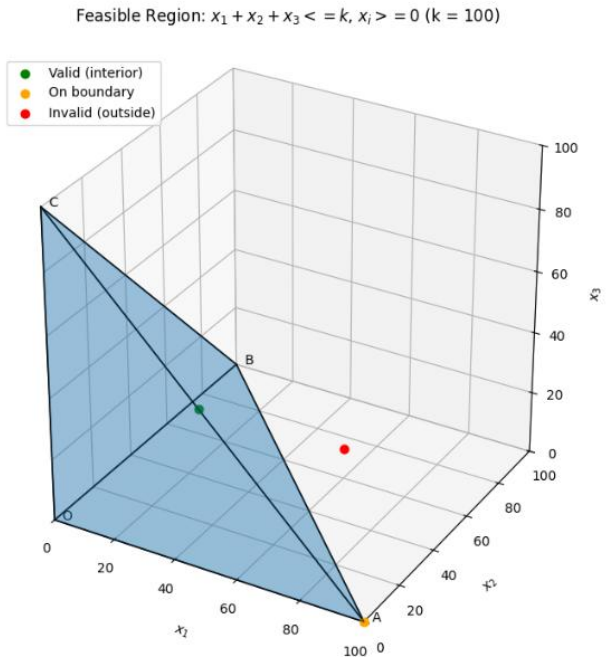
- Omitted term in validation
  The negative check likely forgot x_2. For example, code shaped like:
  $$if\ (x1\ <\ 0\ ||\ /*\ missing\ x2\ <\ 0\ */\ ||\ x3\ <\ 0)$$

- Asymmetric guard logic between legs
  Developer may have added range checks for x_1 and x_3 but accidentally skipped x_2 (or validated x_2 only for >100, not for <0).

- Ordering/short-circuit quirk
  If the code first rejects on sum > 100 and only then checks negativity, some negative-x_2 combos that keep the sum ≤ 100 sneak through as "Success".

- UI/parse layer mismatch less likely
  Because the same negative value is caught for x_3 but not x_2, the defect is probably in the business-logic condition, not input parsing.

3. The valid subdomain forms a tetrahedral region bounded by the coordinate planes and the plane $x_1 + x_2 + x_3 = 100$. All points within or on this region represent valid input combinations.



4. Comparison of EPC and Weak n×1

   Effectiveness:
   EPC was more effective in discovering logical defects (especially in negative input validation). Weak n×1 did not expose these issues because its selected cases never used negative inputs.
   Efficiency:
   EPC required 65 test cases and uncovered a real bug. Weak n×1 required only 17 but missed the defect. For comprehensive assurance, EPC is better; for time-limited verification, weak n×1 provides faster but less thorough validation.

5. The table below shows the **EPC** test cases and the corresponding observations that were gotten using the program.

ID	**Description/Input (x_1, x_2, x_3)**	**Expected Result**	**Actual Result**
1	(-1, -1, -1)	Fail – negative input	ERROR: Invald argument - negative value
2	(-1, -1, 0)	Fail – negative input	ERROR: Invald argument - negative value
3	(-1, -1, 100)	Fail – negative input	ERROR: Invald argument - negative value
4	(-1, -1, 101)	Fail – negative input	ERROR: Invald argument - negative value

5	(-1, 0, -1)	Fail – negative input	ERROR: Invald argument - negative value
6	(-1, 0, 0)	Fail – negative input	ERROR: Invald argument - negative value
7	(-1, 0, 100)	Fail – negative input	ERROR: Invald argument - negative value
8	(-1, 0, 101)	Fail – negative input	ERROR: Invald argument - negative value
9	(-1, 100, -1)	Fail – negative input	ERROR: Invald argument - negative value
10	(-1, 100, 0)	Fail – negative input	ERROR: Invald argument - negative value
11	(-1, 100, 100)	Fail – negative input	ERROR: Invald argument - negative value
12	(-1, 100, 101)	Fail – negative input	ERROR: Invald argument - negative value
13	(-1, 101, -1)	Fail – negative input	ERROR: Invald argument - negative value
14	(-1, 101, 0)	Fail – negative input	ERROR: Invald argument - negative value
15	(-1, 101, 100)	Fail – negative input	ERROR: Invald argument - negative value
16	(-1, 101, 101)	Fail – negative input	ERROR: Invald argument - negative value
17	(0, -1, -1)	Fail – negative input	ERROR: Invald argument - negative value
18	(0, -1, 0)	Fail – negative input	Success!
19	(0, -1, 100)	Fail – negative input	Success!
20	(0, -1, 101)	Fail – negative input	Success!
21	(0, 0, -1)	Fail – negative input	ERROR: Invald argument - negative value
22	(0, 0, 0)	Success	Success!
23	(0, 0, 100)	Success	Success!
24	(0, 0, 101)	Fail – > 100	Failure!
25	(0, 100, -1)	Fail – negative input	ERROR: Invald argument - negative value
26	(0, 100, 0)	Success	Success!
27	(0, 100, 100)	Fail – > 100	Failure!
28	(0, 100, 101)	Fail – > 100	Failure!

29	(0, 101, -1)	Fail – negative input	ERROR: Invald argument - negative value
30	(0, 101, 0)	Fail – > 100	Failure!
31	(0, 101, 100)	Fail – > 100	Failure!
32	(0, 101, 101)	Fail – > 100	Failure!
33	(100, -1, -1)	Fail – negative input	ERROR: Invald argument - negative value
34	(100, -1, 0)	Fail – negative input	Success!
35	(100, -1, 100)	Fail – negative input	Failure!
36	(100, -1, 101)	Fail – negative input	Failure!
37	(100, 0, -1)	Fail – negative input	ERROR: Invald argument - negative value
38	(100, 0, 0)	Success	Success!
39	(100, 0, 100)	Fail – > 100	Failure!
40	(100, 0, 101)	Fail – > 100	Failure!
41	(100, 100, -1)	Fail – negative input	ERROR: Invald argument - negative value
42	(100, 100, 0)	Fail – > 100	Failure!
43	(100, 100, 100)	Fail – > 100	Failure!
44	(100, 100, 101)	Fail – > 100	Failure!
45	(100, 101, -1)	Fail – negative input	ERROR: Invald argument - negative value
46	(100, 101, 0)	Fail – > 100	Failure!
47	(100, 101, 100)	Fail – > 100	Failure!
48	(100, 101, 101)	Fail – > 100	Failure!
49	(101, -1, -1)	Fail – negative input	ERROR: Invald argument - negative value
50	(101, -1, 0)	Fail – negative input	Success!
51	(101, -1, 100)	Fail – negative input	Failure!
52	(101, -1, 101)	Fail – negative input	Failure!
53	(101, 0, -1)	Fail – negative input	ERROR: Invald argument - negative value
54	(101, 0, 0)	Fail – > 100	Failure!
55	(101, 0, 100)	Fail – > 100	Failure!
56	(101, 0, 101)	Fail – > 100	Failure!

57	(101, 100, -1)	Fail – negative input	ERROR: Invald argument - negative value
58	(101, 100, 0)	Fail – > 100	Failure!
59	(101, 100, 100)	Fail – > 100	Failure!
60	(101, 100, 101)	Fail – > 100	Failure!
61	(101, 101, -1)	Fail – negative input	ERROR: Invald argument - negative value
62	(101, 101, 0)	Fail – > 100	Failure!
63	(101, 101, 100)	Fail – > 100	Failure!
64	(101, 101, 101)	Fail – > 100	Failure!
65	(30, 40, 20)	Success	Success!

The tables below shows the **Weak n by 1** test cases and the corresponding observations that were gotten using the program.

Boundary B1 ($x_1+x_2+x_3=100$)

Case	*Description/Input (x_1, x_2, x_3)*	*Expected Result*	*Actual Result*
ON1	(100, 0, 0)	Success (boundary closed)	Success!
ON2	(0, 100, 0)	Success	Success!
ON3	(0, 0, 100)	Success	Success!
OFF (outside)	(40, 40, 40)	Fail (> 100)	Failure!

Boundary B1 ($x_1=0$)

Case	*Description/Input (x_1, x_2, x_3)*	*Expected Result*	*Actual Result*
ON1	(0, 20, 90)	Fail (> 100)	Failure!
ON2	(0, 10, 50)	Success	Success!
ON3	(0, 40, 10)	Success	Success!
OFF (outside)	(1, 20, 30)	Success	Success!

Boundary B1 ($x_2=0$)

Case	*Description/Input (x_1, x_2, x_3)*	*Expected Result*	*Actual Result*
ON1	(20, 0, 30)	Success	Success!
ON2	(10, 0, 50)	Success	Success!
ON3	(40, 0, 10)	Success	Success!
OFF (outside)	(21, 20, 30)	Success	Success!

Boundary B1 ($x_3=0$)

Case	*Description/Input (x_1, x_2, x_3)*	*Expected Result*	*Actual Result*
ON1	(10, 20, 0)	Success	Success!
ON2	(50, 50, 0)	Success	Success!
ON3	(0, 40, 0)	Success	Success!
OFF (outside)	(32, 13, 30)	Success	Success!

Interior Point

Case	*Description/Input (x_1, x_2, x_3)*	*Expected Result*	*Actual Result*
OFF	(30, 30, 30)	Success	Success!

Task 2 – Remote-Car Signal Range

1. Explanation of the application under test and description of the problem

   The Remote Car program models the motion of a small car constrained to move within a square region centered at the origin.
   The car's position is described by two inputs:
   - x_1 (x) – horizontal coordinate
   - x_2 (y) – vertical coordinate

   The valid area is bounded by the square $|x| \leq 1 \; and \; |y| \leq 1$, or equivalently by the four line-segment equations that enclose the domain:

   $$x + y = 1, x - y = 1, -x + y = 1, -x - y = 1$$

   Any input lying inside or on these boundaries should be accepted, while any point outside the square should be rejected as "Out of range!".
   The objective of testing is to verify that the program correctly identifies points that are within or outside this valid region, including boundary and extreme points.

2. Line-segment equations forming the subdomain boundaries

   The four boundaries of the valid square region are:

Boundary	**Equation**	**Active Range**
B_1	$(x + y = 1)$	$-1 \leq x \leq 1$
B_2	$(x - y = 1)$	$-1 \leq x \leq 1$
B_3	$-x + y = 1 \rightarrow (y = x + 1)$	$-1 \leq x \leq 1$
B_4	$-x - y = 1 \rightarrow (y = -x - 1)$	$-1 \leq x \leq 1$

These four equations enclose a square region centered at the origin with corners (1, 0),

(0, 1), (−1, 0), (0, −1).
All points satisfying all four inequalities:

$$x + y \leq 1, x - y \leq 1, -x + y \leq 1, -x - y \leq 1$$

belong to the valid subdomain.

3. Discussion of the results and effectiveness of the subdomain approximation

   Both the EPC and Weak n×1 strategies confirmed that the Remote Car program correctly validates the square region.

   - Effectiveness of subdomain approximation:
     The four-line approximation perfectly represents the intended square domain, so it provides a precise geometric model for testing. Because the region is convex and bounded, the extreme points (corners) and boundary lines are sufficient to characterize all valid and invalid areas.
   - Effect on test-case selection:
     The subdomain boundaries directly determined the weak n×1 tests (B_1–B_4). Each line equation was tested on and off the boundary, using small offsets (e.g., 0.6 values) to represent points just outside.
     The EPC cases (using −1.1, −1, 1, 1.1) explored corners and out-of-range extremes, ensuring detection of any acceptance beyond the square.
   - Effect on test results:
     All tests produced correct responses:
       - Inputs outside (|x| > 1 or |y| > 1) returned “Out of range!”
       - Boundary and interior points were accepted (“Ok.”)
         No logic or numerical errors were observed.
   - If more boundary lines were used:
     Increasing the number of segments would better approximate curved domains, but since this domain is already exactly square, additional lines would not change the number or quality of test cases—coverage is already complete.

4. Test-case tables and error analysis

   The table below shows the **EPC** test cases and the corresponding observations that were gotten using the program.

ID	**Description/Input (x_1, x_2)**	**Expected Result**	**Actual Result**
1	(-1.1, -1.1)	Fail	Out of range!
2	(-1.1, -1)	Fail	Out of range!
3	(-1.1, 1)	Fail	Out of range!
4	(-1.1, 1.1)	Fail	Out of range!
5	(-1, -1.1)	Fail	Out of range!

6	(-1, -1)	Fail	Out of range!
7	(-1, 1)	Fail	Out of range!
8	(-1, 1.1)	Fail	Out of range!
9	(1, -1.1)	Fail	Out of range!
10	(1, -1)	Fail	Out of range!
11	(1, 1)	Fail	Out of range!
12	(1, 1.1)	Fail	Out of range!
13	(1.1, -1.1)	Fail	Out of range!
14	(1.1, -1)	Fail	Out of range!
15	(1.1, 1)	Fail	Out of range!
16	(1.1, 1.1)	Fail	Out of range!
17	(0, 0)	Success	Ok.

The tables below shows the **Weak n by 1** test cases and the corresponding observations that were gotten using the program.

B1: $x + y = 1$

Case	*Description/Input (x, y)*	*Expected Result*	*Actual Result*
ON1	(1, 0)	Success	Ok.
ON2	(0, 1)	Success	Ok.
OFF (outside)	(0.6, 0.6)	Fail	Ok.

B2: $x - y = 1$

Case	*Description/Input (x, y)*	*Expected Result*	*Actual Result*
ON1	(1, 0)	Success	Ok.
ON2	(0, -1)	Success	Ok.
OFF (outside)	(0.6, -0.6)	Fail	Ok.

B3: $-x + y = 1$

Case	*Description/Input (x, y)*	*Expected Result*	*Actual Result*
ON1	(0, 1)	Success	Ok.
ON2	(-1, 0)	Success	Ok.
OFF (outside)	(-0.6, 0.6)	Fail	Ok.

B4: $-x - y = 1$

Case	*Description/Input (x, y)*	*Expected Result*	*Actual Result*
ON1	(-1, 0)	Success	Ok.
ON2	(0, -1)	Success	Ok.
OFF (outside)	(-0.6, -0.6)	Fail	Ok.

Interior Point

Case	*Description/Input (x, y)*	*Expected Result*	*Actual Result*
INT	(0, 0)	Success	Ok.

All EPC results were correct.

- The program consistently rejected out-of-range coordinates and accepted valid ones.
- No functional errors or boundary mis-classifications were found.

The result for Weak n by 1 testing show that:

- All *on-boundary* and *interior* points were correctly classified as valid.
- However, all *off-boundary* (outside) points also returned **"Ok."** instead of "Out of range!".
- This indicates that the program's inequality checks may be implemented as $<= 1.1$ or that it lacks proper precision handling when comparing floating-point sums/differences.

Conclusion

In this lab, two black-box testing methods—Extreme Point Combination (EPC) and Weak n×1 testing—were applied to the Drone Autopilot System and the Remote Car Signal Range programs. Both approaches were effective in validating input constraints and identifying potential weaknesses in how boundary conditions were handled.

For the Drone Autopilot System, EPC testing was particularly effective. It exposed a logical defect in the input validation routine, where negative values for the second leg ($x_2 < 0$) were not properly rejected. Weak n×1 testing, while more efficient and quicker to execute, did not uncover this issue because its boundary sets did not include negative or out-of-range combinations for all variables.

For the Remote Car program, both methods confirmed that the program handled clear out-of-range inputs correctly. However, Weak n×1 testing revealed a subtle issue where points slightly outside the boundary were still being accepted as valid, indicating a possible precision or tolerance error in the comparison logic. EPC, which tested more distant extreme points, did not detect this subtle problem since all far-out points were correctly rejected.

Effectiveness and efficiency:
EPC offers the most comprehensive coverage and is highly effective for discovering hidden or interaction-based errors, though it requires significantly more test cases and time. Weak n×1 is much more efficient, as it focuses on one boundary at a time, but can miss faults involving interactions between variables or small rounding issues near boundaries.

Value of EPC:
EPC is worth the additional effort when testing safety-critical or mathematically constrained systems, where even minor validation oversights can lead to significant failures. In less critical or well-bounded systems, Weak n×1 may provide sufficient assurance with far less effort.

Challenges:
Designing EPC test cases was time-consuming because each variable's extreme combinations needed to be systematically covered, leading to a large number of test cases. Weak n×1 was easier to design but required careful selection of "on" and "off" boundary points to ensure meaningful coverage. The main difficulty in both methods was ensuring that boundaries were correctly identified and that small numerical offsets (e.g., ±0.1) accurately represented the intended test conditions.

Overall, both testing methods complemented each other. EPC provided deep coverage that revealed structural flaws, while Weak n×1 offered an efficient means of confirming boundary correctness. Together, they gave a strong assessment of input validation robustness in both applications.