Variability and consensus

Replicability

There is not necessarily a consensus across groups' answers. Let's investigate why.

- Consider your code about associative property and floating point computation. List all possible factors that can have an effect on the computed answer.
- Create a git *branch* that produces a variation of your code, considering one previous factor, and that impacts the answer
    - using Github actions or Gitlab runner, show that the answer is indeed different
- Back to your main branch, create another variation that impacts the answer and commit/push
    - using Github actions or Gitlab runner, show that the answer is indeed different
- Through tries/errors and a series of commit/push, document which factors do have an influence on the result

Aligning

In the context of "aligning" results across different repositories, the goal is to standardize or control the variability factors so that different groups or environments compute the same result for the same problem.

Take the git repo of another group that have different answer than you!

- Write a script to detect such repo *r'*
- Verify that previous variability factors you have identified on *your* repo also have an effect on computed answers of the repo *r'*
    - This involves running their code and observing whether the factors you've identified also affect the result in their environment. If they don't, it means there are additional or different factors at play.
- Try to control all variability factors and "align" them with regard to what *you* have used. Controlling variability means identifying and managing the factors that can cause differences in results during a computational process. It involves standardizing or minimizing these factors to ensure consistency and replicability of outcomes across different environments or systems.
    - The overall goal is to have the same result given the same variability factors' values
    - If alignment fails (i.e., you still get different results after controlling for the identified factors), this could indicate:

        - Undetected variability factors in either your repo or the other group's repo.
        - Subtle differences in implementation that need to be reviewed manually.
        - System-level differences that might not be immediately obvious (e.g., operating system-level optimizations or floating-point handling).

Exploring all factors?

A limitation of prior experiments is that:

- we certainly consider a subset of factors (not all)
- we study only one factor at a time (not the combination of factors)

We will write a procedure to automatically explore all variability factors and their combinations. We will consider one programming language and your repo.

And yes, we will write a program that generates other programs.
We will use Python to explain the principles, but feel free to use another language and technology.

Implement using a template-based approach

Here's a small tutorial using Python and Jinja2 to explore variability factors through code generation, with the example of the associative property and other arithmetic properties:

Step 1: Install Jinja2

First, ensure that Jinja2 is installed in your environment. If it's not already installed, you can use pip to install it:

```
pip install Jinja2
```

Step 2: Create a Template with Variability Factors

Let's create a Jinja2 template to generate Python code that checks arithmetic properties, such as the associative property for floating-point operations. The factors we'll explore include:

- The number of repetitions (how often the property is checked).
- The arithmetic property (associative or another).

Here is an example template file called `property_template.py.jinja`:

```
import random

def check_property(repetitions):
    correct_count = 0
    for _ in range(repetitions):
        x = random.random()
        y = random.random()
        z = random.random()

        # Check if the associative property holds
        result1 = {{ operation1 }}
        result2 = {{ operation2 }}

        if result1 == result2:
            correct_count += 1

    print(f"Out of {repetitions} trials, the property held {correct_count} times.")

# Define the number of repetitions
repetitions = {{ repetitions }}

check_property(repetitions)

```

(see property_template.py.ninja https://github.com/acherm/REP-INSA2425/blob/main/property_template.py.ninja)

Step 3: Generate Code with Jinja2

Now, we can write a Python script that uses Jinja2 to fill in the template with specific values for the number of repetitions and the arithmetic property.

Here's how you can do this using Jinja2:

```python
from jinja2 import Template

# Load the template
template_content = open('property_template.py.jinja').read()
template = Template(template_content)

# Define different variability factors
factors = {
    "operation1": "(x + y) + z",
    "operation2": "x + (y + z)",
    "repetitions": 1000,  # Can be changed to any number of repetitions
}

# Render the template with specific values
generated_code = template.render(factors)

# Save the generated code to a Python file
with open('generated_property_check.py', 'w') as f:
    f.write(generated_code)

print("Generated code has been written to 'generated_property_check.py'.")

```

(see jinja_call.py https://github.com/acherm/REP-INSA2425/blob/main/jinja_call.py)

Step 4: Run the Generated Code

After running the Python script, a file `generated_property_check.py` will be created. You can execute this file to run the check on how often the associative property holds true:

```
python generated_property_check.py
```

This script checks how often `(x + y) + z == x + (y + z)` holds for random floating-point values.

Step 5: Experiment with Different Factors

You can change the values of `factors` in the Python script to explore different numbers of repetitions or different arithmetic properties (for example, checking commutativity: `x + y == y + x`).

For example, to check commutativity, you could modify `factors` like this:

```
factors = {
    "operation1": "x + y",
    "operation2": "y + x",
    "repetitions": 1000,
}
```

Step 6: Automate Exploring Factor Combinations

To systematically explore multiple factors and their combinations, you can create a loop that varies the values for each factor and generates/runs the code for each configuration. Here's an example:

```
factor_combinations = [
    {"operation1": "(x + y) + z", "operation2": "x + (y + z)", "repetitions": 1000},
    {"operation1": "x + y", "operation2": "y + x", "repetitions": 500},
    # Add more combinations as needed
]

for factors in factor_combinations:
    generated_code = template.render(factors)
    with open('generated_property_check.py', 'w') as f:
        f.write(generated_code)

    # Run the generated code automatically (in a real scenario, you'd use subprocess)
    print(f"Running check for factors: {factors}")
    exec(generated_code)  # Use exec to run the code directly
```

(see jinja_call.py https://github.com/acherm/REP-INSA2425/blob/main/jinja_call.py)

This approach can be extended to explore multiple combinations of variability factors, ensuring that you document how each factor influences the results.

Implementation using a CLI

A CLI (Command-Line Interface) allows users to interact with a program by passing arguments at runtime, enabling the execution of tasks with customizable input directly from the command line. This approach emphasizes dynamic flexibility, as the program's behavior can be adjusted based on parameters provided at the time of execution.

Compared to a template-based approach, which generates new code by filling predefined templates with user-specified values, the CLI approach avoids the complexity of code generation and is more suited for cases where customization is needed without altering the structure of the program itself. While templates provide more control over code variations, the CLI is simpler and more flexible for runtime experimentation.

```python
import random
import argparse

def check_property(operation1, operation2, repetitions):
    correct_count = 0
    for _ in range(repetitions):
        x = random.random()
        y = random.random()
        z = random.random()

        # Dynamically evaluate operations
        result1 = eval(operation1)
        result2 = eval(operation2)

        if result1 == result2:
            correct_count += 1

    print(f"Out of {{repetitions}} trials, the property held {{correct_count}} times.")

if __name__ == "__main__":
    # Parse arguments at runtime
    parser = argparse.ArgumentParser(description="Check arithmetic property.")
    parser.add_argument('--operation1', type=str, required=True, help="First operation to compare (e.g., '(x + y) + z').")
    parser.add_argument('--operation2', type=str, required=True, help="Second operation to compare (e.g., 'x + (y + z)').")
    parser.add_argument('--repetitions', type=int, required=True, help="Number of repetitions for the check.")

    args = parser.parse_args()

    # Run the property check with runtime arguments
    check_property(args.operation1, args.operation2, args.repetitions)
```

(see cli.py https://github.com/acherm/REP-INSA2425/blob/main/cli.py)

To explore combinations of factors using the CLI-based program, you can write a script that calls the program multiple times, each time passing different combinations of operation1, operation2, and repetitions as arguments. You can achieve this by using Python's *subprocess* module or by writing a shell script that invokes the CLI program with different parameters.

Examples with Bash:

```bash
#!/bin/bash

# List of combinations
combinations=(
    "--operation1='(x + y) + z' --operation2='x + (y + z)' --repetitions=1000"
    "--operation1='x + y' --operation2='y + x' --repetitions=500"
    "--operation1='(x - y) - z' --operation2='x - (y - z)' --repetitions=1000"
    "--operation1='x * (y / z)' --operation2='(x * y) / z' --repetitions=700"
)

# Loop through each combination and run the CLI program
for combination in "${combinations[@]}"
do
    echo "Running: python check_arithmetic_properties.py $combination"
    python check_arithmetic_properties.py $combination
done
```

(see bash_cli.sh https://github.com/acherm/REP-INSA2425/blob/main/bash_cli.sh)

Exploring combination of variability factors with runtime parameters (without CLI)

A key limitation of the CLI approach is the need to write additional scripts to automate multiple runs, as it requires manual input for each execution, making it less efficient for large-scale experimentation.

Instead of using CLI arguments, we can directly use runtime parameters and variables in a single program.

The so-called runtime parameter-based approach allows for automated exploration of multiple input combinations within a single execution, making it highly efficient for large-scale experimentation without manual intervention. Unlike the CLI-based approach, which requires rerunning the program for each new set of parameters, this method systematically tests all combinations in one run, saving time and effort. It eliminates the need for repeated executions and manual input, making it ideal for scenarios involving numerous factors. This approach is especially useful when testing variability in configurations or operations, as it streamlines the process and reduces user interaction.

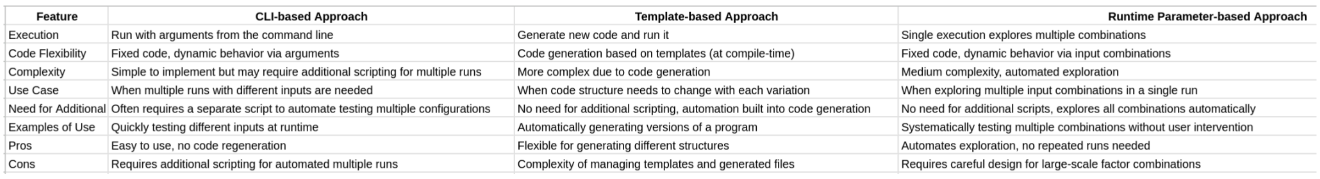
Here is an example:

```python
import random

def check_property(operation1, operation2, repetitions):
    correct_count = 0
    for _ in range(repetitions):
        x = random.random()
        y = random.random()
        z = random.random()

        # Dynamically evaluate the operations
        result1 = eval(operation1)
        result2 = eval(operation2)

        if result1 == result2:
            correct_count += 1

    print(f"Out of {repetitions} trials, {correct_count} times the property held for {operation1} and {operation2}.")

# Define possible combinations of operations and repetition counts
operations = [
    {"operation1": "(x + y) + z", "operation2": "x + (y + z)"},  # Associativity
    {"operation1": "x + y", "operation2": "y + x"},              # Commutativity
    {"operation1": "(x * y) * z", "operation2": "x * (y * z)"},  # Associativity (multiplication)
    {"operation1": "x * (y / z)", "operation2": "(x * y) / z"}   # Combination of operations
]

# Define different repetition counts
repetitions_list = [500, 1000, 2000]

# Loop through all combinations of operations and repetitions
for op in operations:
    for reps in repetitions_list:
        print(f"\nChecking {op['operation1']} vs {op['operation2']} with {reps} repetitions:")
        check_property(op['operation1'], op['operation2'], reps)

```

(see parameterized.py https://github.com/acherm/REP-INSA2425/blob/main/parameterized.py)

Comparison of approaches

Here is an informal comparison of approaches



Feature	CLI-based Approach	Template-based Approach	Runtime Parameter-based Approach
Execution	Run with arguments from the command line	Generate new code and run it	Single execution explores multiple combinations
Code Flexibility	Fixed code, dynamic behavior via arguments	Code generation based on templates (at compile-time)	Fixed code, dynamic behavior via input combinations
Complexity	Simple to implement but may require additional scripting for multiple runs	More complex due to code generation	Medium complexity, automated exploration
Use Case	When multiple runs with different inputs are needed	When code structure needs to change with each variation	When exploring multiple input combinations in a single run
Need for Additional	Often requires a separate script to automate testing multiple configurations	No need for additional scripting, automation built into code generation	No need for additional scripts, explores all combinations automatically
Examples of Use	Quickly testing different inputs at runtime	Automatically generating versions of a program	Systematically testing multiple combinations without user intervention
Pros	Easy to use, no code regeneration	Flexible for generating different structures	Automates exploration, no repeated runs needed
Cons	Requires additional scripting for automated multiple runs	Complexity of managing templates and generated files	Requires careful design for large-scale factor combinations

Both have pros and cons… choose one approach for the rest of the work.

Serializing results

Whatever the approach (template-based, CLI, runtime parameters), there is a need to:
- model the possible values of variability factors
- explore all combinations of factors
    - if it's too much, use a sampling (ie a subset) of all combinations of factors
- store results in a tabular data-like format, eg CSV file (with columns/features = variability factors + answer)

Here is the work to do:
- Implement such serialization.
- Write a Github actions to automatically produce results

Analyzing results

Using the CSV file of the results:
- describe to what extent variability factors impact the result and answer
- recommend values of variability factors to have a stable result

Hint: apply interpretable machine learning (eg decision tree) to systematize the analysis.

Use Jupyter notebook to write the text and conclusions.

Lab report

Your deliverable will be a GitHub repository containing:
- codes to run all the experiments (including Dockerfile and Github actions)

- a Jupyter notebook detailing the experiments (variability factors, analysis of results, recommendations), presenting results, and answering questions
- a README.md explaining the architecture of the git

Deadline: 4 november 11:59 PM (Paris time)

Same for banking (bonus)

Redo the same steps and follow the same methodology (modeling of variability factors, systematic exploration, analysis, recommendations) for the banking problem