IT - 314 Software Engineering

Assignment 7: Program Inspection, Debugging and Static Analysis

Prof. : Saurabh Tiwari

Name: Tirth Modi (202201513)

October 7, 2024

CODE 1:

1. How many errors are there in the program? Mention the errors you have identified.

- **Category A: Data Reference Errors**
  - Unset or uninitialized variables: The method `__init__` is incorrectly defined as `_init_`, meaning the constructor is never called. Thus, `self.matrix`, `self.vector`, and `self.res` remain uninitialized.
- **Category C: Computation Errors**
  - Division by zero: In the `gauss` method, if `A[i][i]` is zero, there is a risk of division by zero when calculating `x[i] /= A[i][i]`.
- **Category D: Comparison Errors**
  - Incorrect comparison logic: In the `diagonal_dominance` method, the check for diagonal dominance can produce misleading results if there are duplicate maximum values in rows, causing unexpected behavior.
- **Category E: Control-Flow Errors**
  - Off-by-one errors: In the `get_upper_permute` method, the loop iterating over k has the condition `for k in range(i+1,n+1)`, which should be `for k in range(i+1,n)` to prevent accessing out-of-bounds.
- **Category F: Interface Errors**
  - Handling of parameter attributes: The method signatures assume a specific format of input data (list of lists for matrix and list for vector) without validation or error handling for unexpected input formats.
- **Category G: Input/Output Errors**
  - The code lacks error handling for cases where input matrices are not square or incompatible dimensions for the operations performed.
- **Category H: Other Checks**
  - **Missing Libraries**: Not importing numpy and `matplotlib.pyplot`, which are necessary for the program to run correctly.

2. Which category of program inspection would you find more effective?

Category D: Comparison Errors and **Category A: Data Reference Errors** seem most effective in this context because many issues arise from incorrect comparisons (such as ensuring diagonal dominance) and the failure to properly initialize the constructor, leading to uninitialized references that can cause runtime errors.

3. Which type of error are you not able to identify using the program inspection?

Logic Errors: While program inspection can identify syntax and runtime errors, it might miss logical errors—where the code runs without crashing but produces incorrect results due to flawed algorithms or miscalculations. For instance, in numerical methods like Jacobi or Gauss-Seidel, the algorithms might converge slowly or not at all under certain conditions, which would not be identified through inspection alone.

4. Is the program inspection technique worth applicable?

Yes, the program inspection technique is quite applicable. It can effectively identify common and critical types of errors, particularly those related to data reference, computation, comparison, and control flow. It serves as a valuable practice for improving code quality and robustness, but it should be complemented by other testing methods (like unit testing, integration testing, etc.) to catch logical errors and ensure that the program behaves as expected under various conditions.

CODE 2:

1. How many errors are there in the program? Mention the errors you have identified.

- **Category A: Data Reference Errors**
  - **Constructor Naming Error**: The constructor is defined as `_init_` instead of `__init__`. This prevents the constructor from being called when creating an instance of the `Interpolation` class.
  - **Potential Index Errors**: The code accesses elements of `matrix` directly in the `cubicSpline` and `piecewise_linear_interpolation` methods without verifying if the indices are within bounds, leading to potential `IndexError`.
  - **No Type Checking**: There are no checks to ensure that the elements in the `matrix` are numeric (integers or floats). Passing non-numeric types could lead to runtime errors.

- **Category C: Computation Errors**
  - **Division by Zero Risk**: The calculation of `slope` in the `piecewise_linear_interpolation` method can lead to division by zero if two consecutive x-values are the same.
  - **Uninitialized Variables**: The `err` attribute is defined but not properly initialized in some methods, leading to potential inconsistencies in its use.

- **Category D: Comparison Errors**
  - **Floating Point Comparisons**: While not explicitly shown, comparisons involving floating-point arithmetic can lead to inaccuracies. When comparing values that may be results of floating-point operations, the code should ensure that the precision is adequately handled.
- **Category E: Control-Flow Errors**
  - **Loop Indexing Issues**: The loop in `mynewtonint` method should ensure it does not exceed bounds, especially with arrays, to prevent `IndexError` if `n` is small.
  - **Missing Edge Case Handling**: In `cubicSpline`, there's a lack of handling for cases where the input matrix has fewer than 4 points, which could lead to errors during matrix operations.
- **Category F: Interface Errors**
  - **Insufficient Documentation**: Functions lack docstrings that explain their purpose, parameters, and return values, making it harder to understand how to use them effectively.
  - **No Parameter Validation**: There's no validation of input parameters across methods, which could lead to incorrect results or runtime errors.
- **Category G: Input/Output Errors**
  - **No Error Handling for Plotting**: The plotting functions do not have checks for valid input data, which could result in runtime errors when plotting invalid data.
- **Category H: Other Checks**
  - **Missing Compiler Warnings**: The code may generate warnings during compilation or execution regarding unused variables or potential errors, but there are no mechanisms in place to catch or handle them.

2. Which category of program inspection would you find more effective?

The **Data Reference Errors (Category A)** would be the most effective category for inspection in this code. This category focuses on crucial issues related to initialization and index handling, such as the incorrect constructor naming (`_init_` instead of `__init__`) and potential index errors when accessing the `matrix`. Addressing these issues is vital to prevent runtime errors and ensure the stability of the program.

3. Which type of error are you not able to identify using the program inspection?

Runtime Errors: Certain runtime errors, particularly those arising from floating-point inaccuracies (such as division by zero in `piecewise_linear_interpolation`) or unexpected input data (such as passing a non-numeric type to the methods), may not be detectable through static inspection alone. These types of errors can lead to incorrect behavior during execution, which would require dynamic testing or error handling to identify.

4. Is the program inspection technique worth applicable?

Yes, program inspection is a valuable technique. It helps identify many potential issues early in the development process, enhances code quality, promotes adherence to best practices, and reduces long-term costs associated with debugging and maintenance. However, it should be complemented with other testing and debugging methods, such as unit testing and dynamic analysis, to ensure comprehensive coverage of all potential errors.

CODE 3:

1. How many errors are there in the program? Mention the errors you have identified.

- **Category A: Data Reference Errors**
  - **Redundant Function Definitions:** The functions `fun` and `dfun` are redefined multiple times for different equations without clarifying which is which.
  - **Undefined Behavior on Reuse of Variables:** The `data` variable is reused to store different iterations of results but is not clearly reset in each function call, potentially leading to unexpected behavior when running multiple roots consecutively.

- **Category B: Data-Declaration Errors**
  - **Uninitialized Variables:** In the initial loop, `next` is calculated before it has been properly initialized with a value in the case of the first iteration, potentially leading to `NaN` values.
  - **Improper DataFrame Initialization:** The DataFrame `df` is created only after the loop, which means if the loop does not run (due to an immediate convergence), `data` may not be well-formed, causing errors.
- **Category C: Computation Errors**
  - **Inaccurate Function Evaluation:** `fpresent = fun(present)` should also check for convergence on `|fun(present)|` rather than only relying on the value of `next` for convergence.
  - **C2: Error Calculation Logic:** The computation of `error.append(next - present)` at the end may not reflect true convergence behavior since it compares the last and the second last iterations rather than the previous two values used in the iterative process.

- **Category D: Comparison Errors**

    - **Incorrect Error Condition:** The error condition checks the difference between `next` and `present`, but may not account for the possibility that `present` might be very close to `alpha` without converging effectively.
    - **Insufficient Convergence Criteria:** The convergence criteria only rely on `abs(next - present) > err`, ignoring the potential impact of the function value itself (i.e., `|fun(next)| < err`).

- **Category E: Control-Flow Errors**
    - **Infinite Loop Risk:** If the initial guess is far from the actual root or if `dfun(present)` is zero (i.e., vertical tangents), the loop may enter an infinite loop without reaching convergence.
    - **Lack of Break Conditions:** There are no safeguards to break the loop after a set number of iterations or to prevent division by zero in `next = present - (fpresent / dfpresent)`.

- **Category F. Input/Output Errors**
    - **Lack of Iteration Logging:** The code does not provide any logging or console output during iterations, making it difficult to trace how the algorithm progresses.
    - **Misleading Plot Titles:** The plot titles do not clearly indicate what function or root they are representing, leading to confusion when analyzing multiple roots from different functions.

- **Category G. Other Checks**
    - **Unchecked Edge Cases:** The code does not handle edge cases where the function may not have a root in the specified domain or the derivative might lead to undefined behavior.
    - **Multiple Plots Without Clearing Previous Data:** Each new plot is created without clearing the previous plot's data, which can lead to cluttered visualizations if multiple functions are tested consecutively.

2. Which category of program inspection would you find more effective?

Data Reference Errors (Category A): This category helps ensure that inputs are properly handled and defined, preventing many runtime errors and improving program robustness.

3. Which type of error are you not able to identify using the program inspection?

Non-obvious Logical Errors: These could include issues like convergence to a wrong root or numerical instability that does not manifest until runtime with specific input values.

4. Is the program inspection technique worth applicable?

Yes, program inspection is a valuable technique.This program inspection techniques can uncover a wide range of errors and improve code quality significantly. They are particularly beneficial in collaborative environments, enhancing code maintainability and readability.

CODE 4:

1. How many errors are there in the program? Mention the errors you have identified.

- **Category A: Data Reference Errors**
    - **Inconsistent Input Structure:** The input `matrix` is expected to be a 2D array, but the code does not validate or handle incorrect input shapes, which could lead to runtime errors.
    - **Variable Reuse Without Clear Definition:** The variable names `coef` and `poly_i` are reused in different scopes (inside and outside the function), which can lead to confusion regarding their intended meanings.

- **Category B: Data-Declaration Errors**
    - **Uninitialized Variables in Plotting:** The plotting function `plot_fun` does not account for scenarios where `y` might be empty or not initialized properly, which could lead to errors when attempting to plot.
    - **No Error Handling for Matrix Inversion:** There is no check to ensure that `ATA` is invertible before calling `np.linalg.inv(ATA)`, which could cause a crash if the matrix is singular.

- **Category C: Computation Errors**
    - **Potential Loss of Precision:** The line `coef = coef[::-1]` reverses the coefficients, but the original polynomial construction using `np.poly1d` expects the coefficients to be in descending order. This could lead to unexpected polynomial behavior if not properly aligned.
    - **Overwriting Coefficients:** The coefficients for each order are computed and stored in `coef` in a loop, but they are not isolated for each polynomial in the final output, which may lead to confusion about which coefficients correspond to which polynomial.

- **Category D: Comparison Errors**
  - **Incorrect Error Tolerance:** The `err = 1e-3` is hardcoded in the `plot_fun` function, which may not be appropriate for all datasets and does not allow for dynamic adjustment based on input ranges.
  - **Inadequate Comparison Logic in Plotting:** When plotting multiple polynomial fits, the code does not ensure that each polynomial is distinctly labeled or that the legend accurately reflects the plotted lines.

- **Category E: Control-Flow Errors**
  - **Infinite Loop Risk in Plotting:** The plotting function could potentially enter an infinite loop if it processes incorrectly formatted data, especially if there are no points to plot.
  - **Lack of Early Exit Conditions:** The `leastSquareErrorPolynomial` function does not implement any conditions to exit early if it detects that the matrix is poorly conditioned or the degree `m` is too high for the number of points.

- **Category F: Input/Output Errors**
  - **No User Feedback on Processing:** There is no print statement or logging mechanism to indicate progress or completion of polynomial fitting, making it difficult for the user to track what's happening during execution.
  - **Misleading Variable Naming:** The variable `poly_i` could be misleading as it suggests a single polynomial when it actually stores the polynomial object. A more descriptive name would enhance clarity.

- **Category G: Other Checks**
  - **No Handling of Edge Cases:** The function does not handle edge cases where all y values are the same, which would lead to a constant polynomial and could confuse the user.
  - **Lack of Unit Tests or Assertions:** There are no unit tests or assertions present to validate the input parameters and ensure that the function behaves as expected across a variety of cases.

- **Category H: General Code Quality**

- **Redundant Code Sections:** The code for plotting multiple polynomials is somewhat redundant and could be encapsulated in a function to improve code reusability.
- **Missing Function Documentation:** There is no documentation for the functions, which makes it harder for other users (or even the author later) to understand the purpose and expected behavior of the code.

2. Which category of program inspection would you find more effective?

Computation Errors (Category C): Ensuring that computations are performed correctly is crucial for the accuracy of results, especially in numerical methods like polynomial fitting.

3. Which type of error are you not able to identify using the program inspection?

Data-Specific Errors: For instance, certain edge cases with input data (e.g., all y values being the same) may not be identified until the function is executed with specific datasets.

4. Is the program inspection technique worth applicable?

Yes, program inspection is a valuable technique.It allows for systematic error identification and improvement in code structure and maintainability, making it especially valuable in complex numerical methods and data analysis tasks.

REFERENCE CODE : GITHUB LINK