University of Alberta

ECE 322 – Software Requirements and Testing

Lab 3 Report

Unit and Pairwise Testing (White-box Testing #1)

Student: Diepreye Charles-Daniel

CCID: diepreye

Lab Section: D22

Instructor: Prof. Witold Pedrycz

Date: November 19, 2025

Contents

1 Introduction

This lab focuses on applying core white-box testing techniques to different Python programs and understanding how each method contributes to finding errors early. The goal is not only to run tests, but to look inside the code and reason about its behaviour, its control flow, and the situations where things may fail.

1.1 Statement of Problem

The lab is divided into three main parts. In Part 1, we work with a bisection-based root-finding algorithm and design unit tests that achieve meaningful statement and branch coverage, including error handling and ensuring loop bodies execute more than once. Part 2 uses the `MathPackage` module, where mocking becomes important to isolate specific behaviours and reach code paths that are otherwise difficult to trigger. Part 3 explores a simple three-variable system to apply pairwise testing and compare different ways of generating reduced test sets.

1.2 Identification of What Is Being Tested

Across the lab, the following components are tested:

- the **bisection method implementation** (Part 1),
- the **MathPackage utilities** and any functions that require mocks to test properly (Part 2),
- a **conceptual A–B–C system** used for generating pairwise combinations (Part 3).

These programs are intentionally small so the focus stays on the testing methods rather than the applications themselves.

1.3 Testing Methodologies and Criteria Used

The testing techniques used in this lab include Python unit testing, statement and branch coverage analysis, loop coverage requirements, mock objects for isolating dependent behaviour, and pairwise test generation using both orthogonal arrays and the `allpairspy` tool. Together, these methods allow us to compare their strengths, what types of faults they help uncover, and how effective each approach is in practice.

2 Part One – General

2.1 Q1: Unit Testing as a White-box Testing Methodology

Unit testing focuses on testing small, isolated pieces of code to ensure they behave as intended. In this lab, unit testing is used as a white-box technique because we have full access to the source code and can design tests based on its internal structure. This allows us to specifically target branches, error conditions, and loop behaviours that may not be visible from a pure black-box perspective.

2.2 Q2: Code Coverage Criteria

Code coverage measures how much of the program's internal structure is exercised by the test suite. The two main criteria used in this lab are:

- **Statement coverage**: verifying that every executable statement runs at least once.
- **Branch coverage**: verifying that both outcomes of each decision point (true/false) are executed.

Coverage helps confirm that tests are not passing accidentally and that the underlying logic has actually been exercised.

2.3 Q3: Purpose of Pairwise Testing

Pairwise testing is a strategy that ensures all possible two-way interactions between input parameters are covered. The motivation is that most faults are caused by the interaction of at most two variables. This makes pairwise testing an efficient alternative to exhaustive testing, especially when the number of combinations becomes large.

2.4 Q4: Purpose of Mock Objects in Testing

Mock objects allow us to replace real dependencies with controlled stand-ins so we can test behaviours in isolation. They are particularly useful when the real dependency is slow, difficult to access, or hard to force into specific states. Using mocks makes it possible to trigger error paths, verify expected calls, and reach code paths that would otherwise be difficult to cover.

3 Part One – Bisection Method Unit Tests

3.1 Q1: Problem Description

The target algorithm for this part of the lab is a Python implementation of the bisection method, which approximates a root of a continuous function on an interval $[a, b]$. The method repeatedly computes the midpoint, checks the sign of the function, and selects the sub-interval that still contains the root. It terminates when the result meets a given tolerance or when the maximum number of iterations is reached. If the initial interval is invalid or the function values do not bracket a root, the implementation is expected to raise an appropriate error. The goal of the tests in this section is to verify both normal behaviour and error handling.

3.2 Q2: Control Flow Graph for the Algorithm

Figure 1 in the Appendix shows the control flow graph (CFG) of the bisection method algorithm used as the basis for test design.

3.3 Q3: Test Case Tables for the Bisection Method

Tables 1 and 2 summarize the unit tests designed for `MyBisect`. The first table focuses on the constructors and property accessors, and the second table covers the main algorithm in `run(x1, x2)`.

Table 1: Constructor and accessor test cases for `MyBisect`.

ID	Target	Description	Input / Setup	Expected Outcome (Actual)
T1	`__init__` (3 args)	Explicit tolerance and maxIterations provided.	`b = MyBisect(0.01, 10, f)`	`b.tolerance = 0.01`, `b.max_iterations = 10`, `b._func is f`; object created without error. (Pass)
T2	`__init__` (2 args, float)	First argument is a float, treated as tolerance, with default maxIterations.	`b = MyBisect(0.001, f)`	`b.tolerance = 0.001`, `b.max_iterations = 50`, `b._func is f`. (Pass)
T3	`__init__` (2 args, int)	First argument is an int, treated as maxIterations, with default tolerance.	`b = MyBisect(5, f)`	`b.tolerance = 0.000001`, `b.max_iterations = 5`, `b._func is f`. (Pass)
T4	`__init__` (1 arg)	Only function provided; both tolerance and maxIterations default.	`b = MyBisect(f)`	`b.tolerance = 0.000001`, `b.max_iterations = 50`, `b._func is f`. (Pass)
T5	`__init__` (0 args)	No arguments; defaults set and no function associated.	`b = MyBisect()`	`b.tolerance = 0.000001`, `b.max_iterations = 50`, `b._func is None`. (Pass)
T6	`tolerance` setter (true branch)	Updating tolerance with a positive value.	`b = MyBisect(f); b.tolerance = 0.005`	`b.tolerance` changes from default to `0.005`. (Pass)
T7	`tolerance` setter (false branch)	Attempting to set tolerance with non-positive values.	`b = MyBisect(f); b.tolerance = 0; b.tolerance = -1.0`	`b.tolerance` remains at the original positive value; non-positive inputs are ignored. (Pass)
T8	`max_iterations` setter (true branch)	Updating max_iterations	`b = MyBisect(f);`	`b.max_iterations` changes from

Table 2: `run(x1, x2)` test cases for the bisection method.

ID	Description	Input / Setup	Function	Expected Outcome (Actual)
B1	Valid interval with default tolerance and maxIterations; root should be found before the iteration limit.	`f = Polynomial(1, -1); b = MyBisect(f); b.run(0, 4)`	$f(x) = x - 1$	Method returns an approximate root close to 1.0; no exception is raised. (Observed root ≈ 1.0; Pass)
B2	Initial interval does not bracket a root; same sign at both endpoints.	`f = Polynomial(1, -1); b = MyBisect(f); b.run(2, 4)`	$f(x) = x - 1$	First iteration detects $f(x_1) \cdot f(x_2) > 0$ and raises `ValueError('Root Not Found')`. (Exception raised as expected; Pass)
B3	Valid initial interval but very small maxIterations so the iteration limit is exceeded before convergence.	`f = Polynomial(1, -1); b = MyBisect(1, f); b.run(0, 4)`	$f(x) = x - 1$	Loop terminates due to exceeding `maxIterations`, then raises `ValueError('Root Not Found')` after the loop. (Exception raised as expected; Pass)

3.4 Q4: Discussion of Coverage Criteria and Effectiveness

Effectiveness of statement coverage and difficulty. Statement coverage was relatively easy to achieve because the class is small and most statements are reachable through simple constructor calls or setter operations. Tests T1–T9 exercised all constructor branches and both outcomes of the property setters. The statements inside `run` were also reachable as long as both successful and failing executions were included.

Effectiveness of branch coverage and difficulty. Branch coverage required more deliberate planning, especially within `run(x1, x2)`. Test B1 exercised both outcomes of the midpoint update decision, while Tests B2 and B3 triggered the two different exception branches. Although not difficult, achieving full branch coverage required careful selection of intervals that produce the correct sign behaviours.

Achievement of coverage criteria. The test suite satisfies both statement and branch coverage requirements. Tests T1–T9 cover all constructor and accessor branches, while Tests B1–B3 cover all major branches in the main algorithm, including the sign check, the midpoint update, the loop continuation and break condition, and the iteration-limit exception.

Errors discovered. No defects were discovered during testing. All outputs and exceptions matched

the intended behaviour specified by the algorithm.

Difficulty of path testing. Exhaustive path testing is not practical because the bisection loop can iterate many times, and each iteration may follow a different left/right sub-interval update. This results in an unbounded number of possible paths. Instead, achieving strong statement and branch coverage provides meaningful confidence in the correctness of the implementation.

3.5 Q5: Coverage Reports

Test Code

The Python implementation and associated tests used for statement and branch coverage analysis are provided in Listing 1 in the Appendix (Section .2).

Coverage Results

Coverage results for the `MyBisect` module are shown in Figures 2 and 3 in the Appendix (Section .2). The coverage tool reports both statement and branch coverage for all constructor paths, property setters, and the main `run(x1, x2)` algorithm.

Summary of Coverage

The results show that the test suite achieved:

- **Full statement coverage** across all constructor branches, property getters and setters, and the majority of statements in `run(x1, x2)`.
- **Full branch coverage** for the constructor decision logic, the setter conditions, the initial sign-check branch, the midpoint update decision, and both exception paths in the main algorithm.

All expected statements and branches were executed at least once. No uncovered lines remained in the constructor or accessor logic, and all major decision points in the bisection algorithm were exercised by tests B1–B3. These results confirm that the test suite satisfies the coverage requirements for this part of the lab.

4 Part Two – Mocking and MathPackage

4.1 System Under Test: MathPackage

`MathPackage` provides simple mathematical utilities, including the generation of random values in a range and functions to compute the maximum and minimum of a list of numbers. In this part of the lab, the focus is on testing these operations directly and then using mocks to simulate their behaviour and examine method interactions.

4.2 Q1: Test Case Tables for `MathPackage`

Table 3: Unit test cases for `MathPackage.random`, `max` and `min`.

ID	Target	Description	Input / Setup	Expected Outcome (Actual)
M1	`MathPackage.random`	Generate n random values within the interval $[a, b]$.	`n = 5, a = 1.0, b = 3.0` `values = MathPackage.random(n, a, b)`	$\lvert$`values`$\rvert = 5$ and each element satisfies $1.0 \leq v \leq 3.0$. (Observed: 5 values all within range; Pass)
M2	`MathPackage.random`	Handle the boundary case where $n = 0$.	`n = 0, a = 0.0, b = 1.0` `values = MathPackage.random(n, a, b)`	Returns an empty list `[]` when $n = 0$. (Observed: `[]`; Pass)
M3	`MathPackage.max`	Return the largest value from a list with mixed positive and negative numbers.	`values = [1.0, -5.0, 10.0, 3.5]` `result = MathPackage.max(values)`	`result = 10.0`, the maximum element in the list. (Observed: `10.0`; Pass)
M4	`MathPackage.max`	Return the largest value when all elements are negative.	`values = [-10.0, -3.0, -7.5]` `result = MathPackage.max(values)`	`result = -3.0`, the maximum element in the list. (Observed: `-3.0`; Fail)
M5	`MathPackage.min`	Return the smallest value from a list with mixed positive and negative numbers.	`values = [1.0, -5.0, 10.0, 3.5]` `result = MathPackage.min(values)`	`result = -5.0`, the minimum element in the list. (Observed: `-5.0`; Pass)
M6	`MathPackage.min`	Return the smallest value when all elements are positive.	`values = [2.0, 10.0, 3.5, 100.0]` `result = MathPackage.min(values)`	`result = 2.0`, the minimum element in the list. (Observed: `2.0`; Pass)

4.3 Q2: Why We Need Mocking and Its Effectiveness

Mock objects are useful when testing code that depends on methods we cannot, or do not want to, execute directly. In this task, mocking allowed us to replace the real `MathPackage` methods with controlled stand-ins so that the behaviour of the tests could be isolated from the actual computations

performed by the package. This made it possible to verify how each method was invoked rather than relying on random values or specific data conditions.

Mocking also made it easy to force particular return values and to test scenarios that would be difficult to reproduce with real data. The use of mock assertions such as `assert_called`, `assert_called_once`, `assert_called_with`, and inspection attributes like `call_args` and `method_calls` provided precise visibility into how methods were used.

Overall, mocking was effective because it allowed us to break dependencies, control method outputs, and verify detailed interaction behaviour that would not be observable using the real `MathPackage` implementations. This gave clear evidence that the tests themselves were structured correctly and that the expected interactions occurred.

4.4 Q3: Coverage Reports for `MathPackage`

Test Code (Real Implementation)

The unit tests for the real `MathPackage` implementation are provided in Listing 2 in the Appendix (Section .3).

Test Code (Mock-Based Tests)

The mock-based tests that mirror the structure of the real tests and use detailed mock assertions are shown in Listing 3 in the Appendix (Section .3).

Coverage Results

Coverage results for the `MathPackage` module are shown in Figures 4 and 5 in the Appendix (Section .3). These results are obtained from running the real implementation tests.

Summary of Coverage

The coverage reports show that the test suite achieves high statement coverage for the `MathPackage` module and exercises the main behaviours of `random`, `max` and `min`. The unit tests cover normal and boundary cases (for example, $n = 0$ in `random`, mixed-sign and all-positive lists for `max` and `min`).

The mock-based tests do not contribute additional coverage of the implementation, but they verify the interaction patterns and demonstrate the use of `assert_called`, `assert_called_once`, `assert_called_with`, `assert_called_once_with`, `call_count`, `call_args`, `call_args_list` and `method_calls`. Together, these results confirm that both the implementation and the way it is invoked are well understood and tested.

5 Part Three – Pairwise Testing with `allpairspy`

5.1 Problem Description

In this part of the lab, the objective is to apply pairwise testing to a small conceptual system with three independent input variables. Each variable, A, B, and C, can take one of three possible values: 0, 1, or 2. Although no executable program is provided, the purpose of this task is to illustrate how pairwise test design can reduce the total number of required tests while still covering all two-way interactions among the input variables.

The assignment requires constructing a suitable orthogonal array manually and then generating a second set of pairwise test cases using the `allpairspy` tool. These two approaches are then compared to demonstrate the efficiency and effectiveness of pairwise testing.

5.2 Q1: Orthogonal Array Design

Table 4 shows the orthogonal array constructed for the three input variables A, B, and C, each taking values $\{0, 1, 2\}$. The array contains nine tests and provides full pairwise coverage of all two-way interactions among the variables.

Table 4: Orthogonal array for three variables with three values each (OA(9, 3^3, 3)).

Test #	**A**	**B**	**C**
1	0	0	0
2	0	1	1
3	0	2	2
4	1	0	1
5	1	1	2
6	1	2	0
7	2	0	2
8	2	1	0
9	2	2	1

5.3 Q2: Programmatically Generated Pairwise Tests

Using the `allpairspy` tool, a set of pairwise test cases was generated for the three variables A, B, and C, each taking values from $\{0, 1, 2\}$. The resulting combinations are shown in Table 5.

Table 5: Pairwise test cases generated using `allpairspy`.

Test #	A	B	C
0	0	0	0
1	1	1	0
2	2	2	0
3	2	1	1
4	1	0	1
5	0	2	1
6	0	1	2
7	1	2	2
8	2	0	2

5.4 Q3: Discussion of Pairwise Testing and Tool Effectiveness

Both the manually constructed orthogonal array and the pairwise test set generated using `allpairspy` produced nine test cases for the three variables A, B, and C. In each case, the resulting tests achieve full pairwise coverage, ensuring that every two-way combination of input values appears at least once.

The orthogonal array provides a mathematically balanced design in which levels appear uniformly across the columns and every pair of values is systematically represented. However, constructing such an array manually requires familiarity with orthogonal array structures and is more prone to design errors without a reference.

The `allpairspy` tool generates a valid pairwise set automatically with minimal effort. Although the ordering and specific combinations differ from the orthogonal array, the coverage achieved is equivalent. The tool is particularly advantageous when the number of variables or values increases, as it scales efficiently and avoids the complexity of manual design.

Overall, both methods significantly reduce the number of tests from the full $3^3 = 27$ combinations to 9, while still exercising all important pairwise interactions. The orthogonal array offers a structured and balanced layout, whereas `allpairspy` provides a more practical and scalable approach for real-world testing scenarios.

5.5 Q4: Pairwise Test Generation Code

The Python script used to generate the pairwise test cases with `allpairspy` is provided in Listing 4 in the Appendix (Section .4).

6 Conclusion and Discussion

This lab brought together several white-box and combinatorial testing techniques across three different tasks. In the first part, unit tests were developed for the bisection algorithm, and full statement and branch coverage were achieved through a systematic set of constructor tests, accessor tests, and behaviour tests for the main `run(x1, x2)` method. These tests covered normal operation, invalid

initial intervals, and iteration-limit failures, demonstrating the effectiveness of targeted unit tests for exercising internal control flow.

In the second part, the `MathPackage` functions were tested both through direct unit tests and through mock-based tests. The direct tests validated the behaviour of the implementation, while the mock tests illustrated how mocking can isolate method interactions, force specific outcomes, and verify call behaviour. This highlighted the usefulness of mocks when testing code with dependencies or when precise control over a method is required.

The final part of the lab showed the value of pairwise testing. Both the manually constructed orthogonal array and the pairwise set generated using `allpairspy` produced nine tests that covered all two-way interactions among the input variables. This demonstrated how pairwise testing can significantly reduce the number of test cases compared to exhaustive testing, while still providing strong interaction coverage.

Overall, the lab reinforced the strengths and limitations of several testing strategies. Unit testing and coverage analysis provide detailed insight into program behaviour; mocking offers control and isolation when testing dependent components; and pairwise testing improves efficiency in systems with multiple parameters. Together, these techniques form a solid foundation for building reliable and well-tested software.

Appendix

.1 Bisection Method Control Flow Graph

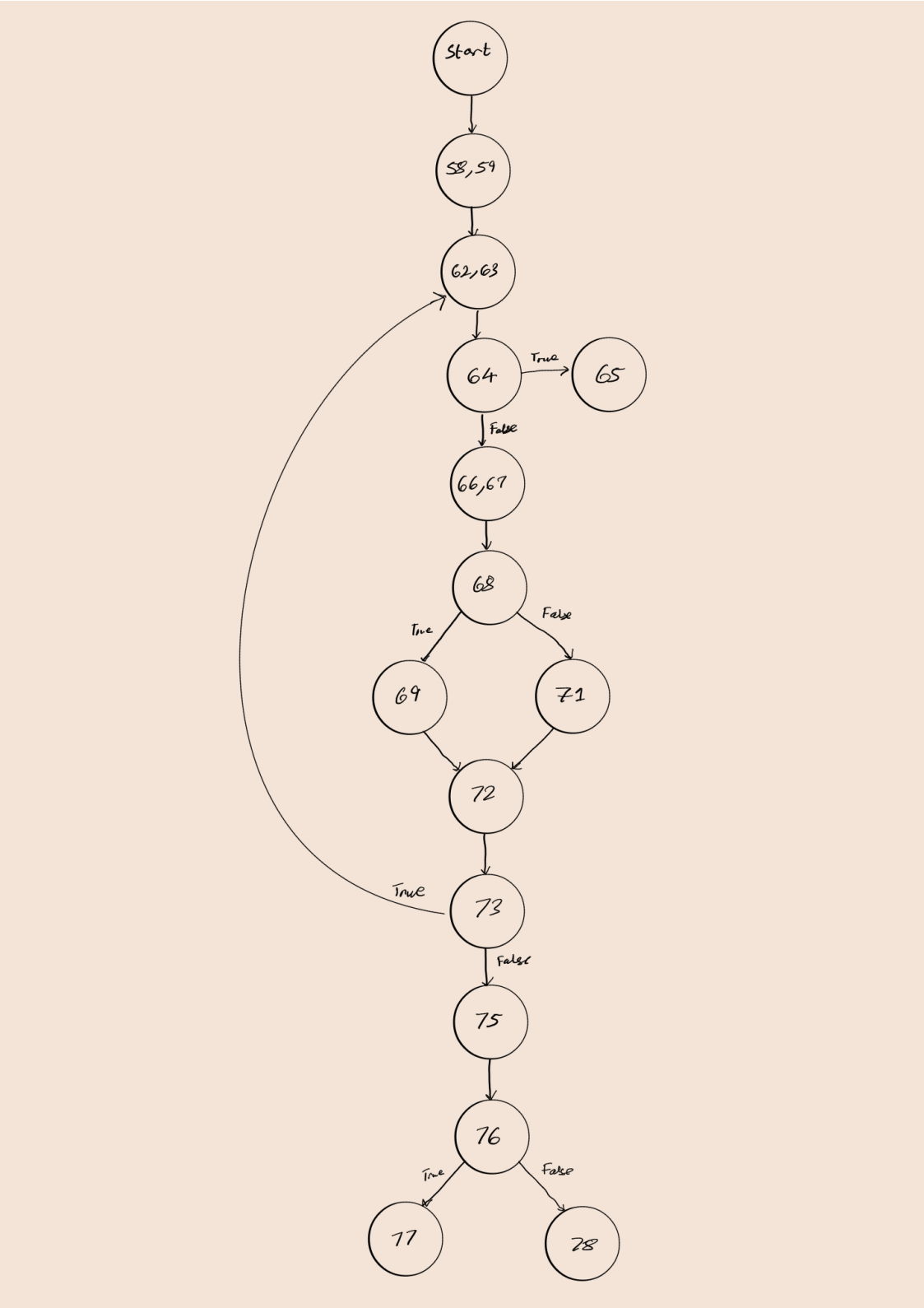

Figure 1: Control flow graph of the bisection method algorithm.

.2 MyBisect Implementation and Coverage

MyBisect Implementation and Polynomial Class

Listing 1: Implementation of MyBisect and Polynomial.

```
class MyBisect:

    # constructor overloading
    # based on args
    def __init__(self, *args):
        # given tolerance, maxIterations, function
        if len(args) == 3:
            self._tolerance = args[0]
            self._maxIterations = args[1]
            self._func = args[2]
        # given tolerance or maxIterations, function
        elif len(args) == 2:
            if isinstance(args[0], float):
                self._tolerance = args[0]
                self._maxIterations = 50
                self._func = args[1]
            else:
                # if arg is an integer
                self._tolerance = 0.000001
                self._maxIterations = args[0]
                self._func = args[1]
        elif len(args) == 1:
            # only given a function
            self._tolerance = 0.000001
            self._maxIterations = 50
            self._func = args[0]
        else:
            self._tolerance = 0.000001
            self._maxIterations = 50
            self._func = None

    # using property decorator
    # a getter function
    @property
    def tolerance(self):
        return self._tolerance

    # a setter function
    @tolerance.setter
    def tolerance(self, tol):
        if tol > 0:
            self._tolerance = tol
```

```
    # using property decorator
    # a getter function
    @property
    def max_iterations(self):
        return self._maxIterations

    # a setter function
    @max_iterations.setter
    def max_iterations(self, max_iter):
        if max_iter > 0:
            self._maxIterations = max_iter

    def run(self, x1, x2):
        iter_num = 1
        mid = 0

        while True:
            f1 = self._func(x1)
            f2 = self._func(x2)
            if f1 * f2 > 0:
                raise ValueError('Root Not Found')
            mid = (x1 + x2) / 2.0
            fmid = self._func(mid)
            if fmid * f1 < 0:
                x2 = mid
            else:
                x1 = mid
            iter_num += 1
            if not((abs(x1 - x2) / 2) >= self._tolerance and abs(fmid) > self._tole
                break
        print("iter_num: {}, max_iter: {}".format(iter_num, self._maxIterations))
        if iter_num >= self._maxIterations:
            raise ValueError('Root Not Found')
        return mid


class Polynomial:
    def __init__(self, *coefficients):
        """ input: coefficients are in the form a_n, ... a_1, a_0
        """
        self.coefficients = list(coefficients)  # tuple is turned into a list

    # The __repr__ and __str__ method can be included here,
    # but is not necessary for the immediately following code

    def __call__(self, x):
        res = 0
```

```
        for coeff in self.coefficients:
            res = res * x + coeff
        return res


def main():
    result = 0
    f = Polynomial(1, -1)  # x-1     ### example for x^2+x-1 use Polynomial(1,1,-1)
    b = MyBisect(f)
    try:
        result = b.run(-10, 10)
        print("Root found at:" + str(result))
    except ValueError as err:
        print(err.args)


if __name__ == "__main__":
    main()
```

MyBisect Coverage Reports

Figure 2: Coverage report for `MyBisect` (Part 1).

Documents \ ECE 322 \ Lab3 \ Lab3_1 \ mybisect.py: 88%

```
67              fmid = self._func(mid)
68              if fmid * f1 < 0:
69                  x2 = mid
70              else:
71                  x1 = mid
72              iter_num += 1
73              if not((abs(x1 - x2) / 2) >= self._tolerance and abs(fmid) > self._tolerance and iter_num <= self._maxIterations)
74                  break
75          print("iter_num: {}, max_iter: {}".format(iter_num, self._maxIterations))
76          if iter_num >= self._maxIterations:
77              raise ValueError('Root Not Found')
78          return mid
79
80
81  class Polynomial:
82      def __init__(self, *coefficients):
83          """ input: coefficients are in the form a_n, ...a_1, a_0
84          """
85          self.coefficients = list(coefficients)  # tuple is turned into a list
86
87      # The __repr__ and __str__ method can be included here,
88      # but is not necessary for the immediately following code
89
90      def __call__(self, x):
91          res = 0
92          for coeff in self.coefficients:
93              res = res * x + coeff
94          return res
95
96
97  def main():
98      result = 0
99      f = Polynomial(1, -1)  # x-1    ### example for x^2+x-1 use Polynomial(1,1,-1)
100     b = MyBisect(f)
101     try:
102         result = b.run(-10, 10)
103         print("Root found at:" + str(result))
104     except ValueError as err:
105         print(err.args)
106
107
108 if __name__ == "__main__":
109     main()
```

« prev ^ index » next coverage.py v7.12.0, created at 2025-11-18 15:23 -0700

Figure 3: Coverage report for `MyBisect` (Part 2).

.3 MathPackage Tests and Coverage

Unit Tests for MathPackage

Listing 2: Unit tests for `MathPackage`.

```
import unittest

from MathPackage import MathPackage


class TestMathPackage(unittest.TestCase):
    """Unit tests for MathPackage.random, max and min."""

    def test_random_generates_n_values_in_range(self):
        n = 5
        a = 1.0
        b = 3.0
        values = MathPackage.random(n, a, b)

        self.assertEqual(len(values), n)
        for v in values:
            self.assertGreaterEqual(v, a)
            self.assertLessEqual(v, b)

    def test_random_with_zero_n_returns_empty_list(self):
        values = MathPackage.random(0, 0.0, 1.0)
        self.assertEqual(values, [])

    def test_max_returns_largest_value_mixed_signs(self):
        values = [1.0, -5.0, 10.0, 3.5]
        result = MathPackage.max(values)
        self.assertEqual(result, 10.0)

    def test_max_returns_largest_value_all_negative(self):
        values = [-10.0, -3.0, -7.5]
        result = MathPackage.max(values)
        self.assertEqual(result, -3.0)

    def test_min_returns_smallest_value_mixed_signs(self):
        values = [1.0, -5.0, 10.0, 3.5]
        result = MathPackage.min(values)
        self.assertEqual(result, -5.0)

    def test_min_returns_smallest_value_all_positive(self):
        values = [2.0, 10.0, 3.5, 100.0]
        result = MathPackage.min(values)
        self.assertEqual(result, 2.0)
```

```
if __name__ == "__main__":
    unittest.main()
```

Mock-Based Tests for MathPackage

Listing 3: Mock-based tests for `MathPackage`.

```
import unittest
from unittest.mock import patch, call

from MathPackage import MathPackage


class TestMathPackageMocks(unittest.TestCase):

    # ---------- Mock for MathPackage.random ----------

    @patch.object(MathPackage, "random")
    def test_random_mock(self, mock_random):
        # Mock return value
        mock_random.return_value = [1.5, 2.0, 2.5, 2.2, 2.7]

        # Call method (this actually calls the mock)
        n = 5
        a = 1.0
        b = 3.0
        result = MathPackage.random(n, a, b)

        # Assertions on return value
        self.assertEqual(result, [1.5, 2.0, 2.5, 2.2, 2.7])

        # Mock assertions
        mock_random.assert_called()
        mock_random.assert_called_once()
        mock_random.assert_called_with(n, a, b)
        mock_random.assert_called_once_with(n, a, b)

        self.assertEqual(mock_random.call_count, 1)
        self.assertEqual(mock_random.call_args, call(n, a, b))
        self.assertEqual(mock_random.call_args_list, [call(n, a, b)])
        self.assertEqual(mock_random.method_calls, [])

    # ---------- Mock for MathPackage.max ----------

    @patch.object(MathPackage, "max")
    def test_max_mock(self, mock_max):
```

```
    # Mock return value
    mock_max.return_value = 10.0

    values = [1.0, -5.0, 10.0, 3.5]

    # Call method
    result = MathPackage.max(values)

    # Assertions on return value
    self.assertEqual(result, 10.0)

    # Mock assertions
    mock_max.assert_called()
    mock_max.assert_called_once()
    mock_max.assert_called_with(values)
    mock_max.assert_called_once_with(values)

    self.assertEqual(mock_max.call_count, 1)
    self.assertEqual(mock_max.call_args, call(values))
    self.assertEqual(mock_max.call_args_list, [call(values)])
    self.assertEqual(mock_max.method_calls, [])

# ---------- Mock for MathPackage.min ----------

@patch.object(MathPackage, "min")
def test_min_mock(self, mock_min):
    # Mock return value
    mock_min.return_value = -5.0

    values = [1.0, -5.0, 10.0, 3.5]

    # Call method
    result = MathPackage.min(values)

    # Assertions on return value
    self.assertEqual(result, -5.0)

    # Mock assertions
    mock_min.assert_called()
    mock_min.assert_called_once()
    mock_min.assert_called_with(values)
    mock_min.assert_called_once_with(values)

    self.assertEqual(mock_min.call_count, 1)
    self.assertEqual(mock_min.call_args, call(values))
    self.assertEqual(mock_min.call_args_list, [call(values)])
    self.assertEqual(mock_min.method_calls, [])
```

```
if __name__ == "__main__":
    unittest.main()
```

MathPackage Coverage Reports

Figure 4: Coverage report for `MathPackage` (overview).

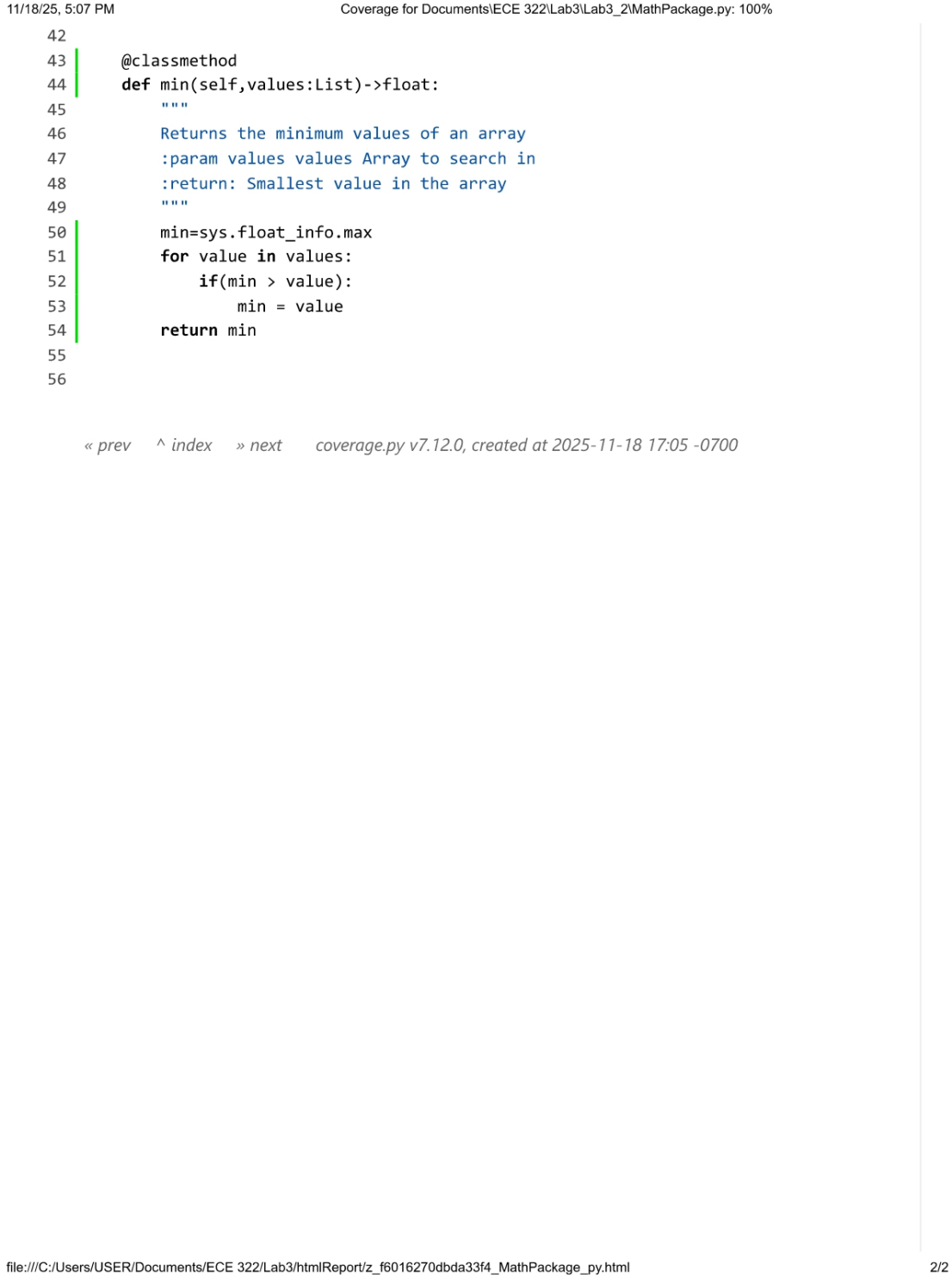

Figure 5: Coverage report for `MathPackage` (detailed view).

.4 Pairwise Test Generation Code

Listing 4: Example pairwise test generation with allpairspy.

```
from allpairspy import AllPairs

def getallpairspy(parameters):

    print("PAIRWISE:")
    for i, pairs in enumerate(AllPairs(parameters)):
        print("{:2d}: {}".format(i, pairs))

def main():
    parameters = [
        [0, 1, 2],
        [0, 1, 2],
        [0, 1, 2],
    ]

    getallpairspy(parameters)

if __name__ == "__main__":
    main()
```