The University of Melbourne
SWEN90006: Software Testing and Reliability

Tutorial 4 Solutions

Semester 2, 2018

Introduction

The aim of this tutorial is for you to familiarise yourself with the various coverage criteria and analysis of the program for the various coverage criteria. When you get back to your revision you should try comparing the test cases that you derive for a program using different techniques.

The different type of program that we encounter this week is a numerical program. One of the challenges of numerical programs is that we can never be certain that we will get an *exact* answer to our computation. Instead what we typically require is an answer to within some *error* value[1]. Numerical programs are tricky to debug, because they are often used to *find* the answer to some problem in the first place. For example, solving some integration or differentiation problems is too hard to do by hand and so we use a *numerical* method to approximate the answer.

Working With the Program

The program implements the standard bisection method for root finding. The root-finding problem is expressed as follows:

> We are given a function $f(x)$ taking a real number and returning a real number[2]. The function is negative at some point x_0 and positive at some point x_1. Find the value x for which $f(x) = 0$ on an interval $[Lower, Upper]$. The point x is a root of f on the interval $[Lower, Upper]$.

As an example, consider the natural logarithm function, ln. The graph in Figure 1 shows the values of $ln(x)$ for various values of x. We can see that the value of $ln(x)$ is equal to 0 when $x = 1$. The bisection algorithm finds this value of x.

The idea behind the algorithm for finding roots is to look at the interval $[Lower, Upper]$ and bisect it (hence the name of the algorithm) and find the midpoint of the interval x_r. If we know that $f(Lower)$ is negative, and $f(Upper)$ is positive then there must be root in the interval, provided that the function is continuous. If the value of f at x_r is positive then the root must be in the interval $[Lower, x_r]$. If the value of f at x_r is negative then the root must be in the interval $[x_r, Upper]$. The algorithm should converge to the root because the length of the interval is getting smaller every time (in fact the length of the interval is halved every time). Does this sound familiar?

[1]Recall from the lecture notes that an error is the difference between a computed value and the exact value.
[2]That is a function $f : \mathbb{R} \to \mathbb{R}$.

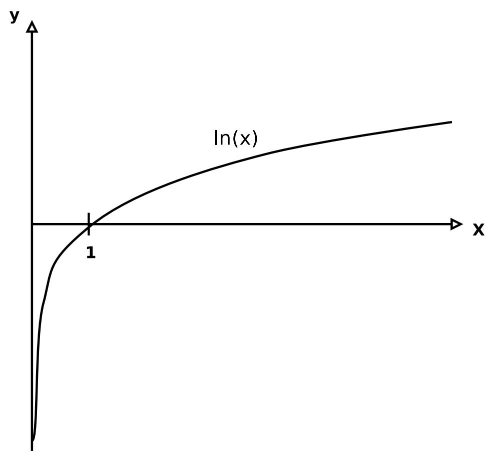


Figure 1: Root finding problems.

Your Tasks

Question 1

What is the input domain for the `Bisection` program below?

Answer If we ignore `f` then the input domain can be easily described from the *types* of the parameter variables. The function `Bisection` uses no other inputs other than those coming from the parameters so we have:

$$\texttt{double} \times \texttt{double} \times \texttt{double} \times \texttt{int}$$

where `double` is the set of double precision floating point numbers. Now, we think of `Bisection` as a function:

$$\texttt{Bisection} \ : \ \texttt{double} \times \texttt{double} \times \texttt{double} \times \texttt{int} \rightarrow \texttt{double}$$

This answer is acceptable up to a point but there are some hidden traps here.

(i) The first is to understand what we mean by "double" (or *double precision floating point numbers*) and exactly what set of numbers this defines.

(ii) The second is that the set of floating point numbers may differ on different machines – although we would hope that they follow the standards for floating point numbers.

(iii) The third is that we do not get *exact* answers and that we can only ever compute some functions *to the nearest floating point number.* Most *numerical computations* have theoretical error bounds.

As far as this tutorial is concerned, the `Bisection` function has avoided the problem of determining errors by passing in the error bound (the `error` parameter) to which the root should be calculated.

For your information – you will not need to know this for the exam! The set of floating point numbers is defined by giving four essential parameters.

The set of floating point numbers is usually defined according to IEEE-Std. 754. The essence of IEEE-Std. 754 is the following definition, but all that you need to remember is that IEEE-Std. 754 defines the set of double precision floating point numbers according to the four parameters below.

1. The base (or radix) B;

2. The precision p;
3. The minimum exponent m; and
4. The maximum exponent M.

The set of *normalised floating point numbers* determined by the parameters (B, p, m, M) consists of 0 and all the numbers x such that

$$x = +/- B^e \times 0.d_1 \ldots 0.d_p$$

where d_i are base B digits, $m \leq e \leq M$ and $d_1 \neq 0$. Double precision floating point numbers, of course, just double the number or bits in the precision p compared with that of single precision floating point numbers. As far as the definition of the set of floating point numbers is concerned, nothing changes from what was given above.

Now, how should you handle the function parameter `f`? There is no clear answer to this so we are looking for some reasonable suggestions.

The problem here is again that the set of functions that match the parameter specification is extremely large.

Step 1 The first question to ask is: what set of functions are we looking at?

The parameter specification requires a function

```
double  *f(double),
```

that is a function that takes a `double` as a parameter and returns a `double` as a result. What functions of this `type` can we think of?

- The identity function.
```
double Identity(double x) {
    return x;
};
```
- Functions with no zeroes. For example, functions that return a constant value such as:
```
double Constant(double x) {
    return 5;
};
```
or functions that do not cross zero anywhere such as:
```
double Quadratic(double x) {
    return x*x + 1;
};
```
- Functions that do cross zero – and there are many of these – for example, linear functions such as
```
double Linear(double x) {
    return x + 2;
};
```
or more complex polynomials such as:
```
double Polynomial(double x) {
    return (x - 3) * (x + 2) * ( x - 1);
};
```

Step 2 Choose good representatives of each set of functions, and ones that you can test; **OR**

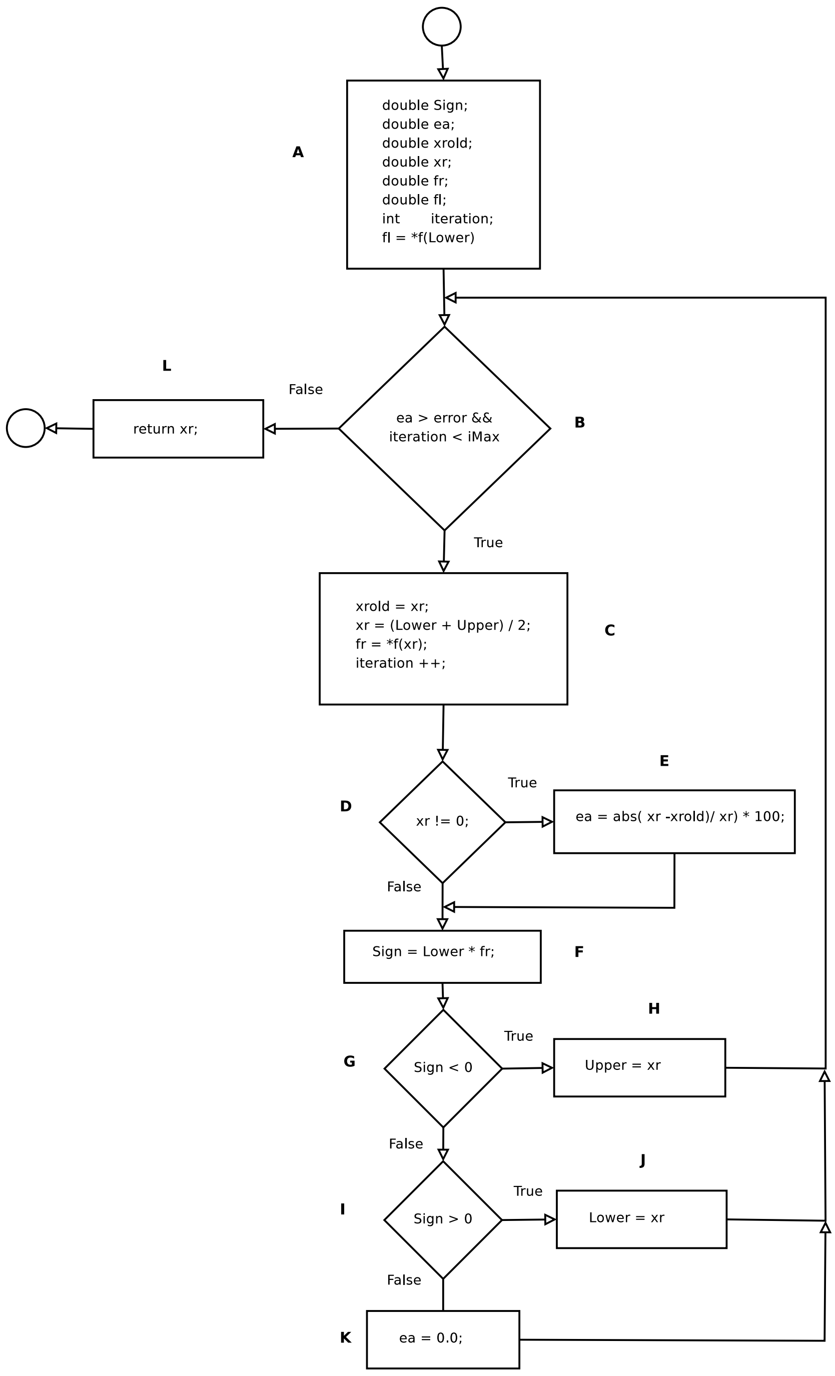
Step 3 Alternatively try and establish which set of functions *will actually be passed in to the* `Bisection` *function.* This information is typically found in the program specifications, in the program algorithm or in the text of the program itself.

Question 2

Draw the control-flow graph for the `Bisection` function. You may break the function up into basic blocks to simplify your CFG.

Recall that a *basic block* is a continuous sequence of statements where control flows from one statement to the next, a single point of entry, a single point of exit and no branches or loops.

Answer



Question 3

Suppose that we concentrated on the (nice and linear) function $f(x) = x - 2$. Derive a set of test cases that achieve:

(i) Statement coverage; and

(ii) Condition coverage.

Note, that you will have to determine what it means for the `Bisection` function to return the *correct* or *expected* output first.

Answer

This question requires some careful thought.

First, we analyse the CFG and determine if there are any infeasible paths. It turns out that there is are no infeasible paths, but that we will need a number of different test cases to create test suits to achieve the two different coverage criteria.

Now, we derive test cases that achieve (i) statement coverage; and (ii) condition coverage.

(i) For statement coverage we need a set of test cases that execute every statement at least once. Note that by choosing $f(x) = x - 2$ then the function is fixed but we can still vary `Lower`, `Upper`, `error` and `iMax`.

Using the input domain given in part (1) above, we choose the test input:

(`Lower = -1.0`, `Upper = 7.0`, `error = 1`, `iMax = 10`).

The expected output of this is a double value that is within 1% of 2. The 1% comes from the fact that `error = 1` is an input, and therefore, our result must be within that error range.

The above input executes the loop 3 times, with the following values along the way:

Iteration	`Lower`	`Upper`	`xr`	`ea`	`Sign`
1	-1.0	3.0	3.0	100.0	-3.0
2	1.0	3.0	1.0	200.0	3.0
3	1.0	3.0	2.0	0.0	0.0

From this, one can see that each statement down the centre of the CFG is executed, as are nodes E, H, and J. Node E is executed every time, which can be seen by looking at the values in the `xr` column. Looking at the `Sign` column, one can see that the true/false branches of nodes G and I are both executed as well, so the resulting statements in the branches are executed as well.

If we continue executing this test case the loop condition `ea > error` eventually becomes false (run the program and try it!). Thus we have executed all of the statements in both of the branches `Sign > 0.0` and `Sign < 0.0` and we have now terminated after meeting the condition `ea <= error`.

The only statement left is the statement at node K, which executes when `Sign == 0`. This can be covered by using the same test case as above, except setting `Lower` to 0.

(ii) The test cases above do not achieve condition coverage – we have not covered the loop condition `iteration < iMax` (node B) evaluating to false, nor have we covered the condition of `xr != 0` evaluating to false (node D).

First, we will derive a test case for the loop condition `iteration < iMax`. Here, we need the loop to iteration `iMax` times without finding a value within the error range. Clearly, the easiest way to do this is to use the above test case, which should loop 8 times, but reduce the value of `iMax` to less than 8. We choose 0. The expected output is undefined by the specification. In such a case, it is necessary to either determine the output using other means, or to have the specification revised.

Now, for the case of `xr != 0` evaluating to false. We require that the variable `xr` is zero. From the program code, we can see that this occurs when `(Lower + Upper) / 2` is zero.

The most straightforward test case is therefore a case in which they are initially zero, and in which the loop iterates at least once. We choose the following:

(`Lower = 0.0`, `Upper = 0.0`, `error = 1`, `iMax = 1`).

Again, the specification does not define the expected output of this.

The Program

```c
#include <stdio.h>
#include <math.h>

#define MAX_INT 65535

double Bisection(double Lower,         /* Lower bound of the interval. */
                 double Upper,         /* Upper bound of the interval. */
                 double error,         /* Allowed error. */
                 int iMax,             /* Bound on the number of iterations. */
                 double (*f)(double) ) /* The function. */
{
  double Sign = 0.0;   /* Test for the sign of the midpoint xr. */
  double ea = MAX_INT; /* Calculated error value. */
  double xrold = 0.0;  /* Previous estimate.  */
  double xr = 0.0;     /* Current x estimate for the root. */
  double fr = 0.0;     /* Current value of f. */
  double fl = 0.0;     /* Value of f at the lower end of the interval. */
  int iteration = 0;   /* For keeping track of the number of iterations. */

  fl = (*f)(Lower);
  while (ea > error && iteration < iMax) {

    /* Start by memorising the old estimate in xrold and then calculate
       the new estimate and store in fr */

    xrold = xr;
    xr = (Lower + Upper) / 2;
    fr = (*f)(xr);

    iteration++;

    /* Estimate the percentage error and store in ea. */

    if (xr != 0) {
      ea = fabs((xr - xrold)/xr) * 100;
    }

    /* To know whether fr has the same sign as f(Lower) or f(Upper) is easy:
       we know that f(Lower) is negative and we know that f(Upper) is positive.
       Multiple fr by f(Lower) and if the result is positive then fr must be
       negative. If the result is negative then fr must be positive. */

    Sign = (*f)(Lower) * fr;

    if (Sign < 0)
      Upper = xr;
    else if (Sign > 0)
      Lower = xr;
    else
      ea = 0;

    printf("iteration %d = (%f, %f, %f, %f, %f)\n", iteration,
           Lower, Upper, xr, ea, Sign);

  } /* end while */

  return xr;
}

int main()
{
  double f(double);

  double fx = Bisection(-1.0, 7.0, 1, 10, &f);
  printf("value = %f\n", fx);
}

double f(double x)
{
  return x - 2;
}
```