1 The Bisection Method

Most of us have spent years solving equations in math classes. For example, most of us know that we can solve the equation $x^2 + x - 1 = 0$ either by factoring or by using the quadratic formula. For assignment 2 we’ll put the computer to work solving equations similar to this, equations that have the form $f(x) = 0$.

One of the most reliable methods for solving these equations is the bisection method. In the bisection method we start with two $x$ values $a$ and $b$. The two values should have the property that either $f(a) < 0$ and $f(b) > 0$ or $f(a) > 0$ and $f(b) < 0$. In other words, $f(a)$ and $f(b)$ have opposite signs. If the function $f(x)$ is reasonably nice (e.g., a polynomial), then we’re guaranteed that $f(c) = 0$ for some $c$ between $a$ and $b$. See Figure 1.

The bisection method tries to focus in on $c$ by dividing the interval from $a$ to $b$ in half. In Figure 2 $e = (a + b)/2$, we can see that the interval from $a$ to $e$ contains $c$. Furthermore, the size of the interval from $a$ to $e$ is half as long as the original interval. If we repeat this process with $e$ replacing $b$ in our formula, we’ll get a new $e$ that’s even closer to $c$.

It’s easy to come up with a picture in which we need to replace $a$ with $e$ instead. (Draw a picture!) So how do we decide which should be replaced? Well, the key property that $a$ and $b$ have is that $f(a)$ and $f(b)$ have opposite signs: this guarantees us that $c$ is between them. So we want to make sure that when we replace $a$ or $b$ with $e$, the “new” $a$ and $b$ will have the property that $f(a)$ and $f(b)$ have opposite signs. How can we do this? Just check that their product is negative:

$$\text{if } f(a) \cdot f(e) < 0:$$
$$b = e$$
$$\text{else: } \# f(e) \cdot f(b) < 0$$
$$a = e$$

This gives us the start of an algorithm:

$$e = (a + b)/2$$
$$\text{while the estimate } e \text{ isn’t good enough:}$$
$$\text{if } f(a) \cdot f(e) < 0:$$
$$b = e$$
$$\text{else}$$
$$a = e$$
$$\# \text{ new estimate}$$
$$e = (a + b)/2$$

To turn this into Python code, we need to decide when to exit from the loop.

In general, the bisection method doesn’t find the exact solution. For example, if $f(x) = 3x - 1$, then $c = 1/3$ which can’t be represented exactly in binary. So we can’t try something like
Figure 1: A function with $f(c) = 0$

Figure 2: The interval from $a$ to $b$ has been bisected by $e$
while f(e) != 0:
    . . .

However, we always know that $c$ is between $a$ and $b$, and since $e$ is halfway between $a$ and $b$, we know that the distance from $c$ to $e$ is at most $(b - a)/2$. (Look at the $x$-axis in Figure 2.) In other words, the “error” in the estimate $e$ is $\leq (b - a)/2$.

So if we input a maximum acceptable error $tol$ we can repeat the loop until $(b - a)/2 \leq tol$. This gives us the following algorithm:

\[
e = (a + b)/2
\]
\[
max\_err = (b-a)/2
\]
\[
while max\_err > tol:
    if f(a)*f(e) < 0:
        b = e
    else
        a = e
    # new estimate
    e = (a + b)/2
    max\_err = (b-a)/2
\]

In case $f(e)$ is exactly zero, we can immediately exit from the loop using Python’s break statement:

\[
e = (a + b)/2
\]
\[
max\_err = (b-a)/2
\]
\[
while max\_err > tol:
    if f(e) == 0:
        break
    elif f(a)*f(e) < 0:
        b = e
    else
        a = e
    # new estimate
    e = (a + b)/2
    max\_err = (b-a)/2
\]

The break statement will jump out of the loop and continue executing the first statement in the program after the while loop.

If $a$ and $b$ are chosen so that $f(a)f(b) < 0$, and if $f(x)$ is a reasonably nice function, the bisection method is guaranteed to eventually find an approximate solution to $f(x) = 0$. However, it can take a very long time to find $c$. To avoid this problem, we should add an additional condition to the while statement: if the number of passes through the loop body is too large, we can leave the loop and our bisection function can return None.

2 Program 2

For programming assignment 2, you should write a Python program that uses the bisection method to solve an equation having the form $f(x) = 0$. Input to the program will be:

- $tol$: the upper bound on the error (a float),
• **max_iter**: the maximum number of times the program can execute the body of the **while** loop,

• **a**: the left endpoint of the initial interval, and

• **b**: the right endpoint of the initial interval.

These values will be entered in this order. You can assume that **tol** and **max_iter** are correct. That is, **tol** will be a positive float, and **max_iter** will be a positive int. The values **a** and **b** will be floats, and **a** will be less than **b**. However, they may not satisfy the condition \( f(a)f(b) < 0 \). If they don’t your program should print a message and prompt for new values for both **a** and **b**, and it should continue prompting for new values until the user enters values that satisfy the condition \( f(a)f(b) < 0 \).

After getting correct input your program should call a function **bisect** that returns the approximate solution to \( f(x) = 0 \), or **None** if the function required too many passes through the while loop. After the function completes the while loop, it should print the number of passes that were taken through the loop.

Your program should also have a function \( f(x) \) that is called by both the main program and the **bisect** function. It returns the value of \( f(x) \). You should experiment with different functions, but the program you turn in should use the function \( f(x) = x^2 - 2 \). Here are a few functions to experiment with:

- \( f(x) = x^3 + x + 1 \). \( f(-0.682327803828) \approx 0. \)
- \( f(x) = \sin(x) \). This is in the Python math module. \( f(\pi) = 0. \)
- \( f(x) = (x - 1)^{\frac{1}{3}}. \) \( f(1) = 0. \)

After the **bisect** function returns, your function should print a message if it failed to find an approximate solution. If it did find a solution, your program should print the approximate solution and the value of \( f(x) \) at the approximate solution.

### 3 Due Date

In order to receive full credit, your program must be in the **p2** subdirectory of your Subversion repository by 2:00 pm on Friday, October 7, and you must turn in a print out of your program by 5 pm on the 7th.

### 4 Grading

Your program will be graded on the basis of its correctness and its “static features.”

1. Correctness will be 50% of your grade. Does your program find a solution with the desired accuracy when the bisection method finds an approximate solution within the required number of passes? Does the program print a message when it doesn’t find the solution? Does it print the number of passes? Does it keep prompting for **a** and **b** until \( f(a)f(b) < 0 \)?

2. The following static features will be graded.

   (a) Function use will be 20% of your grade. Does your program use the two required functions — the bisection function and the function \( f(x) \). Do they perform as required?
(b) Documentation will be 10% of your grade. Does your header documentation include the author’s name, the purpose of the program, and a description of how to use the program? Are the identifiers meaningful? Are any obscure constructs clearly explained?

(e) Source format will be 10% of your grade. Is the indentation consistent? Have blank lines been used so that the program is easy to read?

(d) Quality of solution will be 10% of your grade. Does your solution contain unnecessary calculations? Is your solution too clever — e.g., has the solution been condensed to the point where it’s incomprehensible?

5 Collaboration

It is OK for you to discuss solutions to this program with your classmates. However, no collaboration should ever involve looking at one of your classmate’s source programs! It is usually extremely easy to determine that someone has copied a program, even when the individual doing the copying has changed identifiers and comments. If we discover that someone has copied a program, the authors of both programs will receive an F in the course.