Chapter 54

Zero finding

54.1 Root finding by bisection

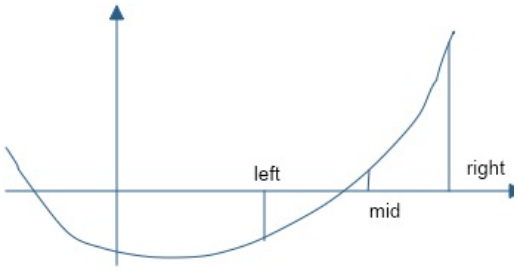

Figure 54.1: Root finding by interval bisection

For many functions f, finding their zeros, that is, the values x for which $f(x) = 0$, can not be done analytically. You then have to resort to numerical *root finding* schemes. In this project you will develop gradually more complicated implementations of a simple scheme: root finding by *bisection*.

In this scheme, you start with two points where the function has opposite signs, and move either the left or right point to the mid point, depending on what sign the function has there. See figure 54.1.

In section 54.2 we will then look at Newton's method.

Here we will not be interested in mathematical differences between the methods, though these are important: we will use these methods to exercise some programming techniques.

54.1.1 Simple implementation

> *Before doing this section, make sure you study chapter 7 about Functions, and chapter 10 about Vectors.*

Let's develop a first implementation step by step. To ensure correctness of our code we will use a Test-Driven Development (TDD) approach: for each bit of functionality we write a test to ensure its correctness before we integrate it in the larger code. (For more about TDD, and in particular the Catch2 framework, see section 68.2.)

54.1.2 Polynomials

First of all, we need to have a way to represent polynomials. For a polynomial of degree d we need $d+1$ coefficients:

$$f(x) = c_0 x^d + \cdots + c_{d-1} x^1 + c_d \tag{54.1}$$

We implement this by storing the coefficients in a `vector<double>`. We make the following arbitrary decisions

1. let the first element of this vector be the coefficient of the highest power, and
2. for the coefficients to properly define a polynomial, this leading coefficient has to be nonzero.

Let's start by having a fixed test polynomial, provided by a function `set_coefficients`. For this function to provide a proper polynomial, it has to satisfy the following test:

```
TEST_CASE( "coefficients are polynomial","[1]" ) {
  auto coefficients = set_coefficients();
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 54.1. Write a routine `set_coefficients` that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Solution to exercise 54.1. Hard-coded:

```
std::vector<double> coefficients{ 1, 1.5, 2, 2.5};
```

Read from user input:

```
vector<double> set_coefficients() {
  int degree;
  std::cout << "Degree?";
  std::cin >> degree ;
  std::cout << std::endl;

  std::vector<double> coefficients(degree+1);
  for (int c=0; c<=degree; c++) {
    std::cout << "coeff" << c << ":";
    std::cin >> coefficients.at(c);
  }
  std::cout << std::endl;
  return coefficients;
};
```

Above we postulated two conditions that an array of numbers should satisfy to qualify as the coefficients of a polynomial. Your code will probably be testing for this, so let's introduce a boolean function `proper_polynomial`:

- This function returns `true` if the array of numbers satisfies the two conditions;

- it returns `false` if either condition is not satisfied.

In order to test your function `proper_polynomial` you should check that

- it recognizes correct polynomials, and
- it fails for improper coefficients that do not properly define a polynomial.

Exercise 54.2. Write a function `proper_polynomial` as described, and write unit tests for it, both passing and failing.

Solution to exercise 54.2.

```
bool proper_polynomial( const std::vector<double>& coefficients ) {
  return coefficients.size()>0 and coefficients.front()!=0.;
};
```

```
TEST_CASE( "proper test","[2]" ) {
  auto coefficients = set_coefficients();
  REQUIRE( proper_polynomial(coefficients) );
  coefficients.at(0) = 0.;
  REQUIRE( not proper_polynomial(coefficients) );
  vector<double> zeroset;
  REQUIRE( not proper_polynomial(zeroset) );
}
```

Next we need polynomial evaluation. You can interpret the array of coefficients in (at least) two ways, but with equation (54.1) we proscribed one particular interpretation.

So we need a test that the coefficients are indeed interpreted with the leading coefficient first, and not with the leading coefficient last. For instance:

```
// correct interpretation: 2x^2 + 1
vector<double> second{2,0,1};
REQUIRE( proper_polynomial(second) );
REQUIRE( evaluate_at(second,2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( evaluate_at(second,2) != Catch::Approx(6) );
```

(where we have left out the `TEST_CASE` header.)

Now we write the function that passes these tests:

Exercise 54.3. Write a function `evaluate_at` which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

For bonus points, look up *Horner's rule* and implement it.

Solution to exercise 54.3.

```
double evaluate_at( const vector<double>& coefficients,double x) {
  double r=coefficients[0];
  for (int c=1; c<coefficients.size(); c++)
    r = x*r + coefficients[c];
  return r;
```

```
};
```

Comments:

- *Coefficients need to be passed in. Global variables are not acceptable.*

With the polynomial function implemented, we can start working towards the algorithm.

54.1.3 Left/right search points

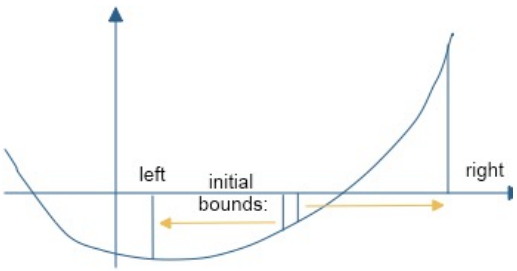

Figure 54.2: Setting the initial search points

Suppose x_-, x_+ are such that

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

that is, the function values in the left and right point are of opposite sign. Then there is a zero in the interval (x_-, x_+); see figure 54.1.

But how to find these outer bounds on the search?

If the polynomial is of odd degree you can find x_-, x_+ by going far enough to the left and right from any two starting points. For even degree there is no such simple algorithm (indeed, there may not be a zero) so we abandon the attempt.

Exercise 54.4. Make the following code work:

```
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

Solution to exercise 54.4.

```
bool is_odd( const vector<double>& coefficients ) {
  return coefficients.size()%2==0;
}
```

Comments:

- *The return type should be* `bool`*, not* `int`*.*

- *Passing the argument as const-ref is good, but probably not part of students' current knowledge.*

You could test the above as:

```
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Now we can find x_-, x_+: start with some interval and move the end points out until the function values have opposite sign.

Exercise 54.5. Write a function `find_outer` which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

How can you compute this test more compactly?
What is a good prototype for the function?
How do move the points far enough out to satisfy this condition?

Solution to exercise 54.5.

```
void find_outer( const vector<double>& coefficients,double &left,double &right) {
  if (is_odd(coefficients)) {
    left = -1; right = +1;
    while ( evaluate_at(coefficients,left)*evaluate_at(coefficients,right) > 0 ) {
      left *=2; right *= 2;
    }
  } else {
    throw("can not find outer for even");
  }
}
```

Comments:

- *Coefficients need to be passed in.*
- *Pass by reference is the easiest solution. Ambitious students can find* `std::pair`.
- *It is correct to decrement/increment the left/right point, but that increases setup time, and has no advantage in zero finding, since that works by bisection.*

Since finding a left and right point with a zero in between is not always possible for polynomials of even degree, we completely reject this case. In the following test we throw an exception (see section 23.2.2, in particularly 23.2.2.3) for polynomials of even degree:

```
right = left+1;
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_outer(second,left,right) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_outer(third,left,right) );
REQUIRE( left<right );
```

Make sure your code passes these tests. What test do you need to add for the function values?

54.1.4 Root finding

The root finding process globally looks as follows:

```
double find_zero( /* something */ ) {
  while ( /* left and right too far apart */ ) {
    // move bounds left and right closer together
  }
  return something;
}
```

The bounds moving code is probably tricky enough that you want to test it on its own.

Exercise 54.6. Write a function `move_bounds` and test it.

Implement some unit tests on this function.

Put it all together:

Exercise 54.7. Make this call work:

```
auto zero = find_zero(coefficients,left,right);
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << "\n";
```

Add an optional precision argument to the root finding function.

Design unit tests, including on the precision attained, and make sure your code passes them.

Solution to exercise 54.7.

```
double find_zero( const vector<double>& coefficients,double left,double right) {
  double
    left_val = evaluate_at(coefficients,left),
    right_val = evaluate_at(coefficients,right);
  assert( left_val *right_val<=0 );
  while (abs(left_val)>1.e-8 && abs(right_val)>1.e-8) {
    move_bounds(coefficients,left,left_val,right,right_val);
    assert( left_val *right_val<=0 );
    // cout << "values: " << left_val << " " << right_val << "\n";
  }
  return (left+right)/2;
}
```

54.1.5 Object implementation

Revisit the exercises of section 54.1.1 and introduce a `polynomial` class that stores the polynomial coefficients. Several functions now become members of this class.

Also update the unit tests.

How can you generalize the polynomial class, for instance to the case of special forms such as $(1+x)^n$?

54.1.6 Templating

In the implementations so far we used `double` for the numerical type. Make a templated version that works both with `float` and `double`.

Can you see a difference in attainable precision between the two types?