Numerical Methods

Xuemei Chen

New Mexico State University

(Last updated on February 6, 2020)

Contents

Notations

Before this course, you need to be familiar with

- Calculus I
- Calculus II: Integrations. Sequences and Series.
- It is not necessary, but helpful to know some multivariate Calculus
- Some basic Linear Algebra.

Notations

- $\{x|$ descriptions of $x\}$ is the set of all x that meets the description after the ":" sign. For example, $\{x|x^2-1<0\}$ is the set of all number x such that $x^2-1<0$. We can solve this inequality further and see that $\{x|x^2-1<0\}=(-1,1)$

  Remark: Some books use colon instead of verticle bar as $\{x:$ descriptions of $x\}$
- $\in$: belongs to/is contained in. For example, $a \in \{x|x^2-1<0\}$ means that a is a number satisfying $a^2-1<0$.
- A vector in $\mathbb{R}^n$ is a column vector in default.
- In all exercises at the end of each chapter, * means extra credit problems.

Chapter 1

Floating Point Arithmetic

The arithmetic performed by a calculator or computer is different from the arithmetic that we use in our algebra or calculus class. In the ideal world, we permit numbers with an infinite number of digits. For example, $\sqrt{2}$ is the unique positive number that, when multiplied by itself, produces the integer 2. It is an irrational number, which means that there are infinitely many digits associated with $\sqrt{2}$ when represented in decimals. In the computational world, we are only able to assign a fixed and finite number of digits to each number.

Try add up 0.1 and 0.2 in Python, you will get

```
>>> 0.1 + 0.2
0.30000000000000004
```

This can cause serious issues and create bugs. See Section 5.1 of [2]. The above error is caused by rounding in floating point numbers, with which numerical analysis is traditionally concerned.

1.1 Floating Point Representation

Floating point numbers are represented in terms of a base β, a precision p, and an exponent e. For example, in base 10, 0.34 can be represented as 3.4×10^{-1} or 34×10^{-2}. In order to guarantee uniqueness, we take a floating point number to have the specific form.

$$\pm\,(d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}) \times \beta^e, \tag{1.1}$$

where each d_i is an integer in $[0, \beta)$ and $d_0 \neq 0$. Such a representation is said to be a *normalized floating point number*. In this representation, $d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}$ is called *mantissa* (or *significand*). Here are a few examples where $\beta = 10, p = 5$ and e is in the range $-10 \leq e \leq 10$:

$$1.2345 \times 10^8, \qquad 3.4000 \times 10^{-2}, \qquad -5.6780 \times 10^5$$

In the above examples, we are using base $\beta = 10$ to ease into the material. Computers work more naturally in binary (base 2). When $\beta = 2$, each digit d_i is 0 or 1 in equation (1.1). I will leave it to the readers to review binary representation, but some examples are listed below:

- Conversion from binary to decimal:
  a. $11.0101_2 = 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4} = 2 + 1 + 0.25 + 0.0625 = 3.3125$
  b. $1100_2 = 1.1_2 \times 2^3 = (1\cdot 2^0 + 1\cdot 2^{-1})\cdot 2^3 = 12$
  c. $0.01_2 = 1.00_2 \times 2^{-2} = 0.25$

- Conversion from decimal to binary:

  a. $41 = 2^5 + 9 = 1 * 2^5 + 1 * 2^3 + 1 * 2^0 = 101001_2$

  b. To figure out the binary representation for 0.1, we first observe that $2^{-4} = 0.0625$ is the biggest component of 0.1. The leftover $0.1 - 0.0625 = 0.0375 > 2^{-5}$. So $0.1 = 2^{-4} + 2^{-5} + 0.00625 = 2^{-4} + 2^{-5} + 2^{-4} * 0.1 = 2^{-4} + 2^{-5} + 2^{-4}(2^{-4} + 2^{-5} + 2^{-4} * 0.1)$. This can keep going, so

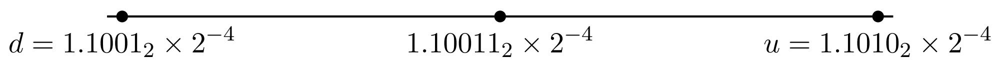
$$0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots = 0.0\overline{0011}_2 = 1.\overline{1001}_2 \times 2^{-4}. \qquad (1.2)$$

As seen, if a base (subscript) is not specified, then it is understood that it is a decimal number. The representation in (1.2) has to be rounded at some point, and this is causing the computation inaccuracy at the beginning of this chapter.

There are usually 4 rounding models: rounding down, rounding up, rounding towards 0, and rounding to nearest. rounding to the nearest is, either round down or round up, whichever is closer. In case of a tie, we choose the one whose least significant (rightmost) bit is smaller. Given a real number x, we denote its floating point representation by $\mathrm{fl}(x)$. For convenience of illustration, we use examples where we retain 4 digits after the binary point (this means the representation has precision $p = 5$).

Example 1.1. Round $0.1 = 1.1001100\cdots_2 \times 2^{-4}$ to the nearest so that there are only 4 digits after the binary point. This means that in this machine, $p = 5$.

Rounding up will be $u = 1.1010_2 \times 2^{-4}$. Rounding down will be $d = 1.1001_2 \times 2^{-4}$. To see which one is nearest, we can compute the midpoint of u and d.


$d = 1.1001_2 \times 2^{-4}$
$1.10011_2 \times 2^{-4}$
$u = 1.1010_2 \times 2^{-4}$

0.1 is clearly bigger than the middle point, so we pick $1.1010_2 \times 2^{-4}$ with rounding to the nearest. We can write $\mathrm{fl}(0.1) = 1.1010_2 \times 2^{-4}$ in this particular machine.

Question: In this same machine, meaning we still have $p = 5$ and we still use rounding to the nearest, what is $\mathrm{fl}(1.10011_2 \times 2^{-4})$? (Hint: in binary, we break the tie by choosing the one whose least significant (rightmost) bit is 0.)

In the model where $\beta = 10, p = 5$, 1 is represented as 1.0000×10^0, the number that is closest to 1 (and bigger than 1) is 1.0001×10^0. There is a gap 10^{-4}. This gap is relative. The closest number to 1.0000×10^4 is 1.0001×10^4 (taking the bigger one again). This means we cannot represent any number between 10000 and 10001 with this machine. The gap (=1) is a lot bigger now, but the relative gap is still $1/10000 = 10^{-4}$.

In general, with base β and precision level p, this gap (between 1 and its closest number) is defined to be the *machine precision* or *machine epsilon* ϵ_m.[1] We have $\epsilon_m = \beta^{1-p}$. Machine epsilon is related to the *round-off error*, which is the error that results from replacing a number with its floating point form. The approximation x^* to x has *absolute error* $|x - x^*|$ and *relative error* $\dfrac{|x - x^*|}{|x|}$, provided that $x \neq 0$.

Example 1.2. Let us use base 10 with $p = 3$, in which case the machine precision is $\epsilon_m = 10^{-2}$. If we use the rounding to nearest method, 1.345 becomes 1.34 (both 1.34 and 1.35 are equally

[1]Some sources define the machine precision to be half of this gap.

close, we break the tie by choosing the number whose right most bit is smaller). Relative error $=\left|\frac{1.345-1.34}{1.345}\right| < \frac{0.005}{1} = 0.5 * \epsilon_m$ (The relative error is about 0.003.)

The number 1.345×10^4 will be rounded to 1.34×10^4. The absolute error is very big, but the relative error is $\left|\frac{1.345 \times 10^4 - 1.34 \times 10^4}{1.345 \times 10^4}\right| = \left|\frac{1.345-1.34}{1.345}\right|$, the same as before.

Example 1.3. Following Example 1.1, the absolute error in representing 0.1 is $|0.1 - \text{fl}(0.1)|$, which is less than half of the difference between u and d. The relative error is

$$\frac{|0.1 - \text{fl}(0.1)|}{0.1} < \frac{2^{-1}(u-d)}{d} = \frac{2^{-5} \cdot 2^{-4}}{1.1001_2 \times 2^{-4}} < 2^{-5}.$$

If we use rounding to the nearest, one has

$$\frac{|\text{fl}(x) - x|}{|x|} \le 0.5\epsilon_m. \tag{1.3}$$

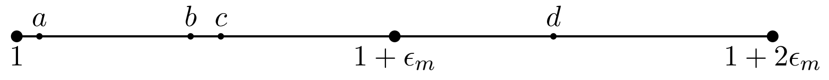
In other words, we may write

$$\text{fl}(x) = x(1+\epsilon), \quad \text{where } |\epsilon| < 0.5\epsilon_m.$$

The idea of machine epsilon is that

- Two numbers may be indistinguishable in a machine if their relative error is less than $0.5\epsilon_m$;
- Two numbers are definitely distinguishable in a machine if their relative error is more than ϵ_m.

Let's consider four numbers a, b, c, d as indicated below:



Using rounding to the nearest, $\text{fl}(a) = \text{fl}(b) = 1$ and a, b are within $0.5\epsilon_m$ relative error. $\text{fl}(b) \neq \text{fl}(c)$ even though they are within $0.5\epsilon_m$ relative error. $\text{fl}(c) = \text{fl}(d)$ even though they are almost ϵ_m apart. The only way to guarantee that two numbers have different representations is make sure they are relatively ϵ_m apart, like b and d.

1.2 Floating Point Operations

On a computer, all computations are reduced to $+, -, \times$, or $/$. Due to rounding, floating point arithmetic are different from the classical ones, and we will denote these *floating point operations (flop)* by $\oplus, \ominus, \otimes, \oslash$.

Example 1.4 (Addition)**.** Suppose we retain 4 digits after the binary point. To add up two numbers with different powers, we first need to adjust one of the numbers to align the mantissa (use bigger power), as:

$1.1000_2 \times 2^1 \oplus 1.1001_2 \times 2^{-1} = 1.1000_2 \times 2^1 \oplus \text{fl}(0.011001_2) \times 2^1 = 1.1000_2 \times 2^1 \oplus 0.0110_2 \times 2^1 = \text{fl}(1.1110_2) \times 2^1 = 1.1110_2 \times 2^1$.

Fundamental axiom of floating point arithmetic: Let $*$ be one of the operations $(+, -, \times, /)$, and let $\circledast$ be its floating point analogue, then for all **floating point numbers** x, y, we have that

$$x \circledast y = (x * y)(1+\epsilon), \text{ for some } \epsilon \text{ such that } |\epsilon| \le \epsilon_m.$$

or equivalently

$$\left| \frac{x \circledast y - x * y}{x * y} \right| \le \epsilon_m.$$

We will not prove this, but we will quickly check this axiom in one example.

Example 1.5. We assume a machine, base 2, can only have 4 digits precision after the binary point, rounding to the nearest. Let $x = 1.1010_2 \times 2^{-4}$ and $y = 1.1010_2 \times 2^{-3}$ be two floating point numbers. (x is the floating point representation of 0.1 by Example 1.1 and y is the floating point representation of 0.2.)

Similar to Example 1.4, we first rewrite x as $0.11010_2 \times 2^{-3}$, which rounds to $0.1110_2 \times 2^{-3}$ (break the tie by picking the one whose right most bit is 0).

$x \oplus y = \text{fl}(0.1110_2 \times 2^{-3} + 1.1010_2 \times 2^{-3}) = \text{fl}(10.1000_2 \times 2^{-3}) = 1.0100_2 \times 2^{-2} = 0.3125.$

The relative error for this addition is

$$\left| \frac{x \oplus y - (x+y)}{x+y} \right| = \left| \frac{1.0100_2 \times 2^{-2} - 1.001110_2 \times 2^{-2}}{1.001110_2 \times 2^{-2}} \right| = \frac{0.000010_2}{1.001110_2} < 2^{-5} < 2^{-4} = \epsilon_m.$$

Finally, we present the example of adding 0.1 and 0.2, but in a "very outdated" machine.

Example 1.6. If we assume a machine, base 2, precision 5. This is how it adds up 0.1 and 0.2.

Step 1: 0.1 is rounded to $\text{fl}(0.1) = 1.1010_2 \times 2^{-4}$ as shown in Example 1.1.

Step 2: $\text{fl}(0.2) = 1.1010_2 \times 2^{-3}$ since 0.2 is twice of 0.1.

Step 3: $\text{fl}(0.1) \oplus \text{fl}(0.2) = 0.3125$ as shown in Example 1.5.

The overall relative error is $|0.3 - 0.3125|/0.3 \approx 0.042$.

To sum up, if we use $m(\cdot)$ to indiate the machine output using floating pont numbers, then
$m(x+y) = \text{fl}(x) \oplus \text{fl}(y) = (x(1+\epsilon_1) + y(1+\epsilon_2))(1+\epsilon_3)$

1.3 IEEE 754 Standard

In 1985, the Institute for Electrical and Electronic Engineers (IEEE) published the *Binary Floating Point Arithmetic Standard 754-1985* which is followed by all microcomputer manufacturers. The double precision real numbers require a 64-bit representation. The first bit is a sign indicator, denoted s. This is followed by an 11-bit exponent c, and a 52-bit binary fraction f (also named mantissa or significand). See the following example:

$$0.1 = +1.\underbrace{1001100110011001100110011001100110011001100110011010_2}_{\text{52 digits mantissa}} \times 2^{-4},$$

This is `0.1000000000000000055511151231257827021181583404541015625` in decimal. In the IEEE 754 standard, $p = 53, \epsilon_m = 2^{-52} \approx 2.22 \times 10^{-16}$. To retrieve this machine epsilon, type `eps = numpy.finfo(float).eps` in Python. Moreover, execute `1 + 0.4*eps`, what do you get?

The default IEEE standard is rounding to nearest. The 11-bit exponent can represent $2^{11} = 2048$ numbers. The numbers 0 and 2047 are reserved for other use (like $0, \infty$, NaN), so c can only take on integers ranging from 1 to 2046. In order to include both positive and negative exponents, we will subtract 1023 from c and the actual exponent can be all integers from -1022 to 1023. To summarize, a double precision number has the form

$$(-1)^s * 2^{c-1023} * (1+f) \tag{1.4}$$

Consider the machine number

$$\underbrace{0}_{s} \quad \underbrace{10000000011}_{\text{11-bit } c} \quad \underbrace{1011100100010000000000000000000000000000000000000000}_{\text{52-bit } f},$$

converted to decimal, $c = 2^{10} + 2^1 + 2^0 = 1027$, $f = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12}$, so this number in decimal form is

$$(-1)^0 * 2^{1027-1023} * (1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12}) = 27.56640625.$$

The smallest positive number that can be represented is $s = 0, c = 1, f = 0$, which is $2^{-1022} \approx 2.225 \times 10^{-308}$.

1.4 Errors in Scientific Computing

Example 1.7. The quadratic formula states that the roots of $ax^2 + bx + c = 0$ are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \tag{1.5}$$

Let us apply this formula to the equation $x^2 + 80x + 1 = 0$, whose roots are approximately $x_1 = -0.01250195, x_2 = -79.98750$.

If the roots are computed in a machine with $\beta = 10, p = 4$. We would get

$$\text{fl}(x_1) = \frac{-80.00 + \text{fl}(\sqrt{6400 - 4.000})}{2.000} = \frac{-80.00 + 79.97}{2.000} = -0.015$$

whose relative error is about 0.0025/0.0125=20%. This is quite large of an error. This big roundoff error is caused by subtracting two nearly equal number. (In this example, we have b^2 much larger than $4ac$, so b and $\sqrt{b^2 - 4ac}$ are nearly equal.)

On the other hand, the calculation of x_2 involves the addition of two nearly equal numbers, which presents no problem as

$$\text{fl}(x_2) = \frac{-80.00 - \text{fl}(\sqrt{6400 - 4.000})}{2.000} = \frac{-80.00 - 79.97}{2.000} = -79.98$$

with a small relative error $\approx 0.01/79.99 = 1.25 \times 10^{-4}$.

There is a way to get a more accurate approximation for x_1 in this same machine!

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} = \frac{-2c}{b + \sqrt{b^2 - 4ac}}. \tag{1.6}$$

This is an equivalent formula for x_1, but makes a big difference in a computer since (1.6) **avoids subtracting two nearly equal numbers**. By (1.6), we have

$$\text{fl}(x_1) = \frac{-2.000}{80.00 + \text{fl}(\sqrt{6400 - 4.000})} = \frac{-2.000}{80.00 + 79.97} = -0.01250.$$

This is a much better approximation.

Due to rounding, subtracting two nearly equal numbers will cause a big relative error. Such an error is called cancellation error and we should try to avoid it if possible. As shown in Example 1.7, rationalizing a formula could be used when appropriate.

Chapter 1 Exercises

1. Convert binary to decimal.
   (a) 110_2
   (b) 11.0101_2

2. Convert decimal to binary
   (a) 11.5
   (b) 1.3125

3. Find the **normalized** floating point representation
   (a) 10.345, base 10, $p = 4$, rounding to the nearest.
   (b) 10.395, base 10, $p = 4$, rounding to the nearest.
   (c) 295, base 10, $p = 2$, rounding to the nearest.
   (d) 1.0101_2, base 2, $p = 3$, rounding up.
   (e) 1.0101_2, base 2, $p = 3$, rounding down.
   (f) 1.0101_2, base 2, $p = 3$, rounding to the nearest.
   (g) 1.0101_2, base 2, $p = 4$, rounding to the nearest.

4. Let us use base 10 with $p = 2$. If we use the rounding to the nearest method, what is the relative error of 1.23? What is the relative error of 10.61?

5. Following Exercise 3(g), what is the machine epsilon? what is the relative error in rounding this number? Does the rule (1.3) check out?

6. Directly add in binary (No rounding involved).
   (a) $0.111_2 + 11.101_2$
   (b) $1.1010_2 \times 2^{-4} + 1.1010_2 \times 2^{-3}$.

7. What is the biggest number that can be represented in IEEE 754 standard using double precision? Express this number in the form of $2^a - 2^b$.

8. In a machine of $\beta = 2, p = 3$, compute the machine result of 0.1+0.2 (rounding to the nearest).

9. In a machine of $\beta = 10, p = 4$, compute the machine result of $6400.1 + 4.12$ (rounding to the nearest).

10. This problem is from Chapter 5 problem 15 of [2].

    In the 1991 Gulf War, the Patriot misslie defense system failed due to roundoff error. The troubles stemmed from a computer that performed the tracking calculations with an internal clock whose integer values in tenths of a second were converted to seconds by multiplying by a 24-bit binary approximation to 0.1:

$$0.1 \approx 0.00011001100110011001100_2$$

    (a) Convert the binary number above to a fraction. Call it x.

(b) Compute the absolute difference between 0.1 and x.

(c) What is the time error in seconds after 100 hours of operation (i.e., the value of $|360,000-3,600,000x|$)?

On February 25, 1991, a Patriot battery system, which was to protet the Dhahran Air Base, had been operating for over 100 consecutive hours. The roundoff error caused the system not to track an incoming Scud missle, which slipped through the defense system and detonated on US Army barracks, killing 28 American soldiers.

11. Find the most accurate approximation of the roots of $x^2 - 75.7x + 1$ with $p = 4$. Compute the relative errors for both roots.

12. Given $p(x) = 1 + 2x + 3x^2 + 3x^3 + 4x^4$,

    (a) Find the nested form of $p(x)$.

    (b) How many flops are needed to evaluate this polynomial at one point, in the original form, and in the nested form?

Chapter 1 Python Exercises

13. (a) Define a vector v whose entries are [1, 0.01, 0.02, ... , 5]. Do not print.

    (b) Define a 10×10 matrix whose entries are the integers 1 to 100 ordered column-wise. Do not print. [Hint: look up `np.reshape`]

    (c) Plot $y = \cos x$ and $y = x$ on the interval [0,1] in the same graph.

14. Given the equation $x^2 - 100000x + 1 = 0$, use formula (1.5) and formula (1.6) to compute two different approximations of x_1. Print both values in the exponential notation with 15 digits after the decimal point. Which formula produces a better approximation of x_1?

15. Given $p(x) = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$.

    (a) Let $n(x)$ be the nested form of $p(x)$. Define $p(x), n(x)$, and $s(x) = (x-2)^9$. All 3 functions need to able to take in arrays.

    (b) $p(x) = n(x) = s(x)$ algebraically, but makes a difference in computers. Plot all three functions on the interval [1.92, 2.08] (with 0.001 increment). You need to place all 3 plots in a 1×3 grid. Be sure to give a title to each subplot to indicate which is which.

    (c) Explain the difference in three plots in (b).

Chapter 2

Solving nonlinear equations

If we recall the equations that we are able to solve, probably we would realize that there are very few. We can surely solve $x^2 - x - 1 = 0$ using the famous quadratic formula, but have you ever wondered formulas for finding roots of polynomials whose degree is higher than 2? Unfortunately, even as simple as a polynomial equation $x^5 - x - 1 = 0$, a math Ph.D feels hopeless to find an exact answer by analysis. In fact, there are no formula for finding roots of polynomials of degree bigger than 4. Equations like $\cos x = x$ do not have a closed form solution either.

A linear equation has the form $ax = b$. It is trivial to solve if x is a scalar. When x is a vector, we often write $Ax = b$, and is a major topic itself.

In this chapter, we discuss several iterative methods for solving nonlinear equations when x is a scalar. Direct methods are formula or procedures that will produce the exact solution. Iterative methods produce a sequence $\{x_1, x_2, \cdots, x_n, \cdots\}$ that (hopefully) converges to the true solution x.

2.1 The Bisection Method

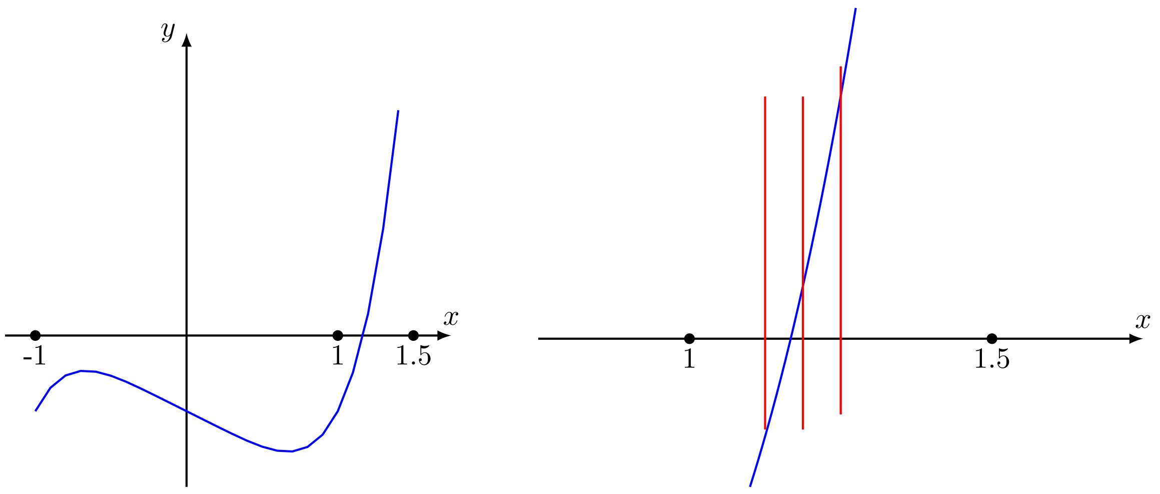


Figure 2.1: Bisection Method. The right is an enlargement of the left.

This is a simple, yet efficient method that is often covered in Calculus I as an application of the Intermediate Value Theorem (IVT). See Theorem **??**.

Take $f(x) = x^5 - x - 1$ as an example, whose graph is shown on the left of Figure 2.1. Since $f(1) = -1 < 0$ and $f(1.5) \approx 5.09 > 0$, then by the IVT, there is $c \in [1, 1.5]$ such that $f(c) = 0$, or equivalently, f has a root c in [1,1.5].

To have a more precise estimate of c, we evaluate f at the middle point $\frac{1+1.5}{2} = 1.25$. $f(1.25) \approx 0.8 > 0$. By the IVT again, we conclude that the root $c \in [1, 1.25]$. We can keep bisecting the interval and eventually find the root to a sufficient precision. See the right of Figure 2.1.

Note that the key of the Bisection method is:

1. The function needs to be continuous.
2. One needs to be given initial points a, b such that $f(a)$ and $f(b)$ have different signs.

The Bisection Algorithm

```
Input:    $a_0, b_0$ such that $f(a_0), f(b_0)$ have different signs.
Output:   an approximation of a root of $f$
Repeat:   for $n \geq 0$
          1. $p_n = \frac{a_n + b_n}{2}$
          2. if $f(a_n)f(p_n) < 0$, set $a_{n+1} = a_n, b_{n+1} = p_n$
             otherwsie, set $a_{n+1} = p_n, b_{n+1} = b_n$
          until a stopping criteria has been reached
```

Example 2.1. Find a solution of $2^{-x} = x$.

This is equivalent to finding a root of $f(x) = (\frac{1}{2})^x - x$. We have $f(0) = 1 - 0 > 0, f(1) = 0.5 - 1 < 0$, so we will let $a_0 = 0, b_0 = 1$. Therefore $p_0 = 1/2$.

$f(1/2) = \frac{1}{\sqrt{2}} - \frac{1}{2} > 0$, so $a_1 = 1/2, b_1 = 1, p_1 = 3/4$.

$f(3/4) = (0.5)^{3/4} - 3/4 < 0$, so $a_2 = 1/2, b_2 = 3/4, p_2 = 5/8$

After only two iterations, we get a coarse approximate of the root: $p_2 = 5/8 = 0.625$. For the reference, the actual root is 0.6411857445.

If we predetermine a tolerance level tol so that the algorithm will terminate if $|p_n - p_{n+1}| < \text{Tol}$, then this can be achieved when

$$|p_n - p_{n+1}| = \frac{b_n - a_n}{4} = \frac{b_0 - a_0}{2^{n+2}} < \text{Tol} \Longleftrightarrow n \geq \log_2(\frac{b_0 - a_0}{4\text{Tol}}) \tag{2.1}$$

If p is the actual root, then we have

$$|p_n - p| \leq \frac{b_n - a_n}{2} = 2|p_n - p_{n+1}| < 2\text{Tol}.$$

Therefore the number of iterations needed can be computed via (2.1).

2.2 The Newton's Method

Newton's method (also called the Newton-Raphson method) is a little more sophisticated: it involves derivatives, but it is still Calculus material. This is usually how a calculator or a computer finds a root.

If we are to find the root of $y = f(x)$, we start with a random initial guess x_0. Consider the the tangent line to the curve $y = f(x)$, at $P_0 = (x_0, f(x_0))$. The idea behind Newton's method is linearization. Since the tangent line (linearization of f) is close to f, then its x-intercept, x_2, is close to the x-intercept of the curve $y = f(x)$. We can easily find x_2.

The tangent line at the point $(x_0, f(x_0))$ is

$$y - f(x_0) = f'(x_0)(x - x_0).$$

Plugging in $y = 0$ to solve for the x-intercept:

$$0 - f(x_0) = f'(x_0)(x - x_0) \Longrightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

So $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$. We use x_1 as the next approximation and keep repeating this process, as shown in Figure 2.2, where we gain this iteration formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k \geq 0. \tag{2.2}$$

Note that we require $f'(x_k) \neq 0$ which is not a trivial condition.

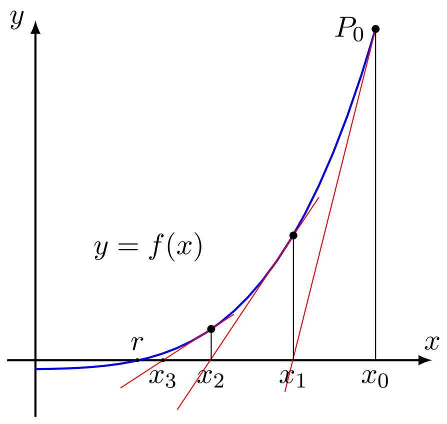


Figure 2.2: Newton's method

Example 2.2. Use the Newton's method to approximate $\sqrt{2}$.

Let $f(x) = x^2 - 2$, which makes $\sqrt{2}$ a root of $f(x)$. We can compute that $f'(x) = 2x$. Let $x_0 = 4$. We can simplify (2.2) to

$$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k} = \frac{x_k}{2} + \frac{1}{x_k}.$$

$x_1 = 2 + 1/4 = 9/4 = 2.25$
$x_2 = 9/8 + 4/9 = 113/72 = 1.569\dot{4}$
$x_3 = 1.569\dot{4}/2 + 1/1.569\dot{4} \approx 1.42189$

If we use the Bisection method for this problem starting at $a_0 = 0, b_0 = 4$, we then have $p_0 = 2, p_1 = 1, p_2 = 1.5, p_3 = 1.25$. This is a much worse approximation after the same number of iterations. See Table 2.1 for error comparison. This table also includes the secant method that will be introduced later.

The two drawbacks of the Newton's method are: (1) It requires the knowledge of the derivative function; (2) It may not converge if you have an unlucky initial guess x_0. For example, if one picks $x_0 = 0$ for finding root of $y = x^5 - x - 1$, the sequence will not converge to the root (try to draw tangent lines on your own). However, if the initial guess is close enough, the Newton's method does enjoy fast convergence (faster than the Bisection method as shown in Example 2.2) .

	Bisection	Newton	Secant
n	$\lvert p_n - \sqrt{2}\rvert$	$\lvert x_n - \sqrt{2}\rvert$	
0	0.585786438	2.585786438	
1	0.414213562	0.835786438	
2	0.085786438	0.155230882	0.414213562
3	0.164213562	0.007676801	0.080880229
4	0.039213562	0.000020723	0.014357866
5	0.023286438	0.000000000	0.000420459

Table 2.1: Accuracy Comparison of Example 2.2

Definition 2.3. A method that produces a sequence $\{x_n\}$ that converges to a number x *linearly* if there exists $L \in (0,1)$ such that for large values of n

$$|x_{n+1} - x| \leq L|x_n - x|.$$

The sequence converges to x *quadratically* if there exists $Q > 0$ such that for large values of n

$$|x_{n+1} - x| \leq Q|x_n - x|^2.$$

For example, the sequence $\{3^{-n}\}_{n=1}^{\infty}$ converges to 0 linearly. The sequence $\{b_n = 3^{-2^n}\}_{n=1}^{\infty}$ converges quadratically to 0.

Chapter 2 Exercises

Python Exercises

1. Bisection and Newton comparison.
   (a) Use the Bisection method to do Example 2.1, with the stopping criteria $|p_{n+1} - p_n| \leq$1e-5, and initial interval $a_0 = 0, b_0 = 1$. How many iterations was run? (meaning what is the n value when $|p_{n+1} - p_n| \leq$1e-5 is reached?) How is this compared to the theoretical value in (2.1)?
   (b) Use the Newton's method to do Example 2.1, with the stopping criteria $|x_{n+1} - x_n| \leq$1e-5, and initial value $x_0 = 1$. How many iterations was run?
2. Newton's method fails for finding a root of $y = x^5 - x - 1$ with $x_0 = 0$. Print the first 6 approximations to illustrate this.

Other Exercises

Bibliography

[1] Richard L. Burden, and J. Douglas Faires. *Numerical analysis.* Cengage Learning, 9th (2010).

[2] Anne, Greenbaum, and Timothy P. Chartier. *Numerical methods: design, analysis, and computer implementation of algorithms.* Princeton University Press, 2012.

[3] Ab Mooijaart, Matthijs Warrens, and Eeke van der Burg, *An introduction to matrix algebra with Matlab*, manuscript, 2006

[4] Lloyd N. Trefethen, and David Bau III. *Numerical linear algebra.* Vol. 50. Siam, 1997.