Numerical Methods

Xuemei Chen

University of San Francisco

(Math 375: Numerical Analysis in Fall 2017)

Contents

Preliminaries

Before this course, you need to be familiar with

- Calculus I
- Calculus II: especially Sequences and Series
- Linear Algebra
    1. Vectors, norm, dot product
    2. Linear Independence and Dependence
    3. Span, subspace, bases, dimension of a subspace/vector space
    4. Properties of orthogonal or orthonormal sets/bases; Gram-Schmidt algorithm
    5. Matrix multiplication; Determinant; Inverse
    6. Solving linear equations: algorithm, existence and uniqueness.
    7. Projections; Least square
    8. Eigenvalues and Eigenvectors
    9. SVD

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 $a^2-1<0$.

- s.t.: such that

Chapter 1

Solving nonlinear equations

If you recall the equations that you are able to solve, you would realize that there are very few. You know how to 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 hand (so you shouldn't feel bad). In fact, there are no formula for polynomials of degree bigger than 4. Equations like $\cos x = x$? does not have a closed form solution either.

When we need to solve nonlinear equations that frequently occur in every aspect of sciences and applications, we use iterative methods.

Bisection Method

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

Theorem 1.1 (Intermediate Value Theorem)**.** *If $f(x)$ is continuous on $[a, b]$, then for any m that is in between $f(a)$ and $f(b)$, there exists a number $c \in [a, b]$ such that $f(c) = m$.*

Take $f(x) = x^5 - x - 1$ as an example, whose graph is shown on the left of Figure 1.1. Since $f(1) = -1 < 0$ and $f(1.5) \approx 5.09 > 0$, then by IVT, there is $c \in [1, 1.5]$ such that $f(c) = 0$ and this c is exactly the root that we are looking for.

To have a more precise estimate of c, we evaluate f at the middle point $\dfrac{1 + 1.5}{2} = 1.25$. $f(1.25) \approx 0.8 > 0$. By 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 Figure 1.1.

Note that the key of this Bisection method are:

1. The function needs to be continuous.
2. One needs to be given initial points a, b such that $f(a)f(b) < 0$.

Newton's Method

Newton's method is a little more sophisticated: it involves derivatives, but it is still Calc I 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_1. 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

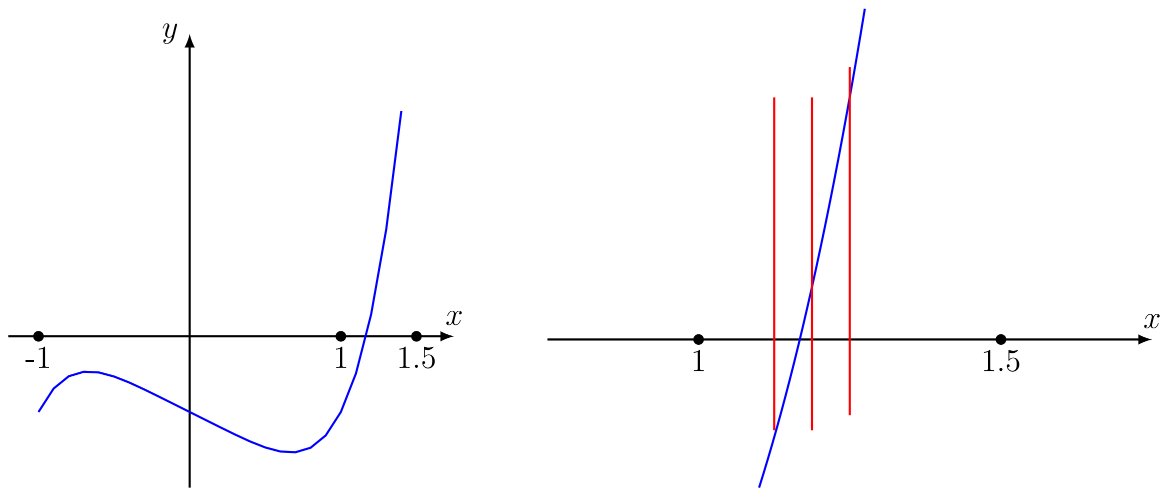


Figure 1.1: Bisection Method

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 1.2, where we gain this iteration formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, k \geq 0 \tag{1.1}$$

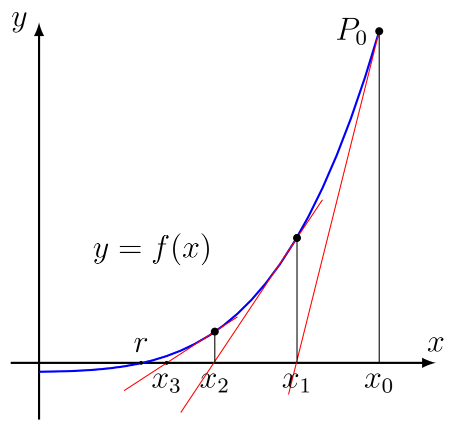


Figure 1.2: Newton's method

Example 1.2. Use Newton's method to approximate $\sqrt{2}$.

Let $f(x) = x^2 - 2$, which makes $\sqrt{2}$ a root of $f(x)$. $f'(x) = 2x$. Let $x_0 = 4$.

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

$x_1 = 2 + 1/4 = 9/4$
$x_2 = 9/8 + 4/9 = 113/72$
...

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 $x_1 = 0$ 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, we do have convergence if the initial guess is close enough.

Theorem 1.3 ([1, Theorem 4.3.1])**.** *If the second derivative of $f(x)$ is continuous, and x_0 is sufficiently close to a root r of f, then Newton's method converges to r and ultimately the convergence rate is quadratic.*

The next section introduces the secant method which is Newton's method without requiring derivatives.

Secant Method

The secant method is defined by taking $f'(x_n)$ to be

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

So the iteration formula for the secant method is:

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}.$$

Fixed point methods

x is called a fixed point of $\varphi(x)$ if $\varphi(x) = x$.

Example 1.4. Find fixed point of $\varphi(x) = x^2$.

This is to solve $x = x^2 \Longrightarrow x(x-1) = 0 \Longrightarrow x = 1, 0$

What about finding fixed points of $\varphi(x) = e^{-x}$ since we can't solve $e^{-x} = x$ by hand anymore? A common way is to iterate using the formula

$$x_{k+1} = \varphi(x_k), \qquad k \geq 0$$

which in this example will be $x_0 = 0, x_1 = \varphi(x_0) = 1, x_2 = \varphi(x_1) = e^{-1}, ...$

Finding the fixed point of $\varphi(x)$ is equivalent to finding the root of $\varphi(x) - x$, which means we can always use Newton's method to find a fixed point. Using the same example, we let

$f(x) = e^{-x} - x$, then $f'(x) = -e^{-x} - 1$. So the Newton' iteration is $x_{k+1} = x_k + \dfrac{e^{-x_k} - x_k}{e^{-x_k} + 1}$. Still letting $x_0 = 0$, we have $x_1 = 0 + 1/2 = 1/2$, which is already a better approximation than the fixed point iteration.

Theorem 1.5. *Assume $\varphi \in C^1$ and $|\varphi'(x)| < 1$ in some interval $[r - \delta, r + \delta]$ where r is a fixed point of φ. If $x_0 \in [r - \delta, r + \delta]$ and we use iteration $x_{k+1} = \varphi(x_k)$, then $\lim_{k\to\infty} x_k = r$.*

Proof. See page 95 of [1]. □

Newton's method (1.1) can be considered as a fixed point iteration. To be specific, (1.1) is equivalent to $x_{k+1} = \varphi(x_k)$ with $\varphi(x) = x - \dfrac{f(x)}{f'(x)}$. We can apply Theorem 1.5 to pick a good initial point for Newton's method.

Example 1.6. We are trying to use Newton's method to find the root of $f(x) = x^3 - 1$. The Newton's iteration is $x_{k+1} = \varphi(x_k)$, where $\varphi(x) = x - \dfrac{f(x)}{f'(x)} = x - \dfrac{x^3 - 1}{3x^2} = \dfrac{2}{3}x + \dfrac{1}{3x^2}$. $\varphi'(x) = \dfrac{2}{3} - \dfrac{2}{3}x^{-3}$.

$$|\varphi'(x)| < 1 \Longrightarrow \left|\frac{2}{3} - \frac{2}{3}x^{-3}\right| < 1 \Longrightarrow |\frac{1}{x^3} - 1| < \frac{3}{2} \Longrightarrow -\frac{3}{2} < \frac{1}{x^3} - 1 < \frac{3}{2} \Longrightarrow x^3 > \frac{2}{5} \text{ or } x^3 < -2$$

We can pick x_0 from the interval $(\left(\dfrac{2}{5}\right)^{1/3}, 2 - \left(\dfrac{2}{5}\right)^{1/3})$. Notice that the interval has to be centered around 1.

Remark 1.7. Note that Theorem 1.5 is only a sufficient condition. Even if $|\varphi'(x_0)| < 1$, x_0 can be still a fine initial point. For example, any x_0 can work in Example 1.6.

Chapter 1 Exercises

* means extra credit problems

1 Use Newton's method to approximate $\sqrt{2}$. Let $x_0 = 1$. Compute 2 iterates only.

2 Prove that Newton's method will converge to 0 given any initial value x_0 if we are solving $x^2 = 0$.

3 Write down the first three iterates of the secand method for solving $x^2 - 3 = 0$, starting with $x_0 = 0$ and $x_1 = 1$.

4 We can compute 1/3 by solving $f(x) = 0$ with $f(x) = x^{-1} - 3$.

(a) Write down the Newton iteration for this problem, and compute by hand the first 2 Newton iterates for approximating 1/3, starting with $x_0 = 0.5$.

(b) What happens if you start with $x_0 = 1$?

(c) *In the case of (b), show that the iterates $x_k \to -\infty$ as $k \to \infty$.

(d) Use the theory of fixed point iteration to determine an interval about 1/3 from which Newton's method will converge to 1/3.

5 Let function $\varphi(x) = (x^2 + 4)/5$.

(a) Find the fixed point(s) of $\varphi(x)$.

(b) Would the fixed point iteration, $x_{k+1} = \varphi(x_k)$, converge to a fixed point in the interval $[0, 2]$ for all initial gueses $x_0 \in [0, 2]$?

(c) *Find a function $f(x)$ such that its Newton iterations are $x_{k+1} = \varphi(x_k)$. (Hint: You need to solve a separable differential equation (Calc II material))

6 Compare Bisection method and Newton's method. List their pros and cons. Think of a way to combine Bisection method and Newton's method to overcome the drawbacks of the Newton's method.

Chapter 2

Floating point arithmetic

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 [1]. The above error is caused by rounding in floating point numbers, with which numerical analysis is traditionally concerned.

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.

$$\pm (d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)})\beta^e, \tag{2.1}$$

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*. 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 these examples, we are using base $\beta = 10$ to ease into the material. Computers work more naturally in binary (base 2). **Please review binary representation**. When $\beta = 2$, each digit d_i is 0 or 1 in equation (2.1). To review, we have

- $11.0101_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 2 + 1 + 0.25 + 0.0625 = 3.3125$
- $1100_2 = 1.1_2 \times 2^3 = (1 \cdot 2^0 + 1 \cdot 2^{-1}) \cdot 2^3 = 12$
- $0.01_2 = 1.00_2 \times 2^{-2} = 0.25$

The number 0.1 has an exact representation in base 10. However, 0.1 does not have a finite binary expansion. In fact

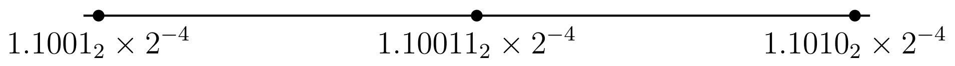
$$0.1 = 0.0\overline{0011}_2$$

meaning $0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots$. The representation has to be rounded at some point, and this is causing the computation inaccuracy at the beginning of this section.

There are usually 4 rounding models: Rounding down, Rounding up, Rounding towards 0, and rounding to nearest. The default IEEE standard is rounding to nearest, that is, either round down or round up, whichever is closer. In case of a tie, it is the one whose least significant (rightmost) bit is 0.

For convenience of illustration, we use examples where we retain 4 digits after the binary point. And we shall use the round to nearest model.

Example 2.1. $0.1 = 1.1001100\cdots_2 \times 2^{-4}$. But we have to round this number so that there are only 4 digits after the binary point. 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.



0.1 is clearly bigger than the middle point, so we pick $1.1010_2 \times 2^{-4}$ with rounding to the nearest.

Example 2.2 (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 significands (use bigger power), as:
$1.1000_2 \times 2^1 \oplus 1.1001_2 \times 2^{-1} = 1.1000_2 \times 2^1 \oplus fl(0.011001_2) \times 2^1 = 1.1000_2 \times 2^1 \oplus 0.0110_2 \times 2^1 = fl(1.1110_2) \times 2^1 = 1.1110_2 \times 2^1$.

Almost all machines today use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 double precision. This standard has 64 bits, with 1 bit for the sign, 11 for the exponent, and 52 for the significand.

Use the IEEE-754 floating point number to represent,

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

which is approximately `0.1000000000000000055511151231257827021181583404541015625` in decimal.

Fundamental Axiom of Floating Point Arithmetic

If we use $fl(x)$ to represent the floating point number of x, then we always obey that $fl(x) = x(1+\epsilon)$ such that $|\epsilon| \le \epsilon_m$, where ϵ_m is the machine precision. This can also be expressed as $\left|\dfrac{fl(x)-x}{x}\right| \le \epsilon_m$, and interpreted as "relative error is bounded by the machine precision."

Example 2.3. Let us use base 10 with 1 digit precision as an example, in which case the machine precision is $\epsilon_m = 10^{-1}$. If we use the rounding to nearest method,

1.23 becomes 1.2. Relative error=$\left|\dfrac{1.23-1.2}{1.21}\right| \approx 0.02$

10.61, is first expressed as 1.061*10, then round to 1.1*10. Relative error=$\left|\dfrac{10.61-11}{10.61}\right| \approx 0.04$

Not only the rounding needs to have small relative error, all the operations $(+, -\times, \div)$ should be stable in the same way:

Fundamental axiom of floating point arithmetic: Let $*$ be one of the operations $(+, -, \times, \div)$, 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

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

Let us check quickly check this axiom with the operation + on one example (insanity check).

Example 2.4. 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 approximately 0.1 by Example 2.1 and y is approximately 0.2 in decimal.)

Similar to Example 2.2, 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).

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

The relative error for this addition is

$$\left|\frac{x \oplus y - (x+y)}{x+y}\right| = \left|\frac{1.0100_2 \times 2^{-2} - 1.001110_2 \times 2^{-2}}{1.001110_2 \times 2^{-2}}\right| = \frac{0.000010_2}{1.001110_2} \approx 0.0256 < 2^{-4} = \epsilon_m.$$

Example 2.5. If we assume a machine, base 2, can only have 4 digits precision after the binary point. This is how it adds up 0.1 and 0.2.

Step 1: 0.1 is rounded to $1.1010_2 \times 2^{-4} = fl(0.1)$.
Step 2: 0.2 is rounded to $1.1010_2 \times 2^{-3} = fl(0.2)$.
Step 3: $fl(0.1) \oplus fl(0.2) = 0.3125$ as shown in Example 2.4.
The overall relative error is $0.0125/0.3 \approx 0.042$.

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

Chapter 2 Exercises

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

2. Directly add and subtract in binary (No rounding involved).
   (a) $0.111_2 + 11.101_2$
   (b) $1.0101_2 - 1.10_2$
   (c) $1.1010_2 \times 2^{-4} + 1.1010_2 \times 2^{-3}$ as in Example 2.4. Convert it to decimal with a calculator.

3. Compute the **normalized** floating point numbers
   (a) 10.3456, base 10, retain 4 digits after point, rounding up.
   (b) 1.0101_2, base 2, retain 2 digits after point, rounding up.
   (c) 1.0101_2, base 2, retain 2 digits after point, rounding down.
   (d) 1.0101_2, base 2, retain 2 digits after point, rounding to the nearest (decide which one is closer by finding the midpoint of answers in (b) and (c)).

4. Compute the machine result of 0.1+0.2 if we retain 2 digits precision after the binary point.

5. *In Python

```
>>>  (0.1 + 0.2) + 0.3
0.6000000000000001
>>>  0.1 + (0.2 + 0.3)
0.6
```

   Explain this phenomenon with a more limited machine where we only retain 2 digits after the binary point.

6. This problem is from Chapter 5 problem 15 of [1].

   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.

Bibliography

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