牛顿法计算平方根

2022年12月19日 15:23

1.1.7 牛顿法计算平方根

计算机支持的数字运算只有加减乘除，计算π、$\sqrt[3]{5}$、 sin(x)等 一些计算任务可以转化为求解方程，例子：计算 $\sqrt{2}$这个任务，就可以转化为计算方程$x^2-2=0$的根

这时我们可以采用牛顿法

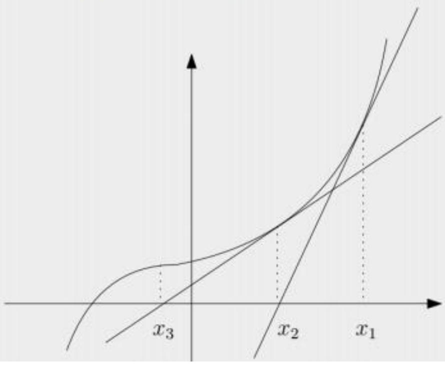
思路：不断逼近方程的根

注意，计算机在计算实数时，做不到百分之百精确，只能做到逼近。比如计算(+ 0.1 0.2)，结果并不是正好等于0.3。

牛顿法原理的图示

如何求解曲线$f(x)$对应的方程

- 猜测值x_1，从x_1计算x_2，从x_2算x_3,...直到算到足够接近方程解的值
- x_2：$f(x_1)$的切线和x轴的交点，可以看到x_1处竖线，x轴和$f(x_1)$的切线构成直角三角形
- x_3：$f(x_2)$的切线和x轴的交点，...
- x_1，x_2，...越来越靠近f(x)=0的根
- 当某个f(x_k)足够接近0时，认为x_k就是方程的根



牛顿法如何从x_i计算x_{i+1}

- 满足等式$f(x_i)+f'(x_i)(x_{i+1}-x_i)=0$
- 因此，$x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)}$

待求解的方程为$f(x)=x^2-n$

因此，导数$f'(x)=2x$

$$x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)}=x_i-\frac{x_i^2-n}{2x_i}=\frac{x_i+\frac{n}{x_i}}{2}$$

写程序时一般将繁琐的步骤拆开，分别定义，

;每一次计算下一个预测值前都要先判断当前预测值够不够精准

;如此形成循环

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

;计算下一个预测值
(define (improve guess x)
  (average guess (/ x guess)))

;平均值
(define (average x y)
  (/ (+ x y) 2))

;精度判断
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.0001))

;平方
(define (square x)
  (* x x))

;还需要将guess初始化，才能开始后续的计算，就使初始预测值为1.0好了
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

另外还可以用内部定义的方式改进：

```
(define (sqrt x)
    (define (good-enough? guess x)
          (< (abs (- (square guess) x)) 0.0001))
    (define (improve guess x)
          (average guess (/ x guess)))
    (define (average x y)
          (/ (+ x y) 2))
    (define (square x)
          (* x x))
    (define (sqrt-iter guess x)
          (if (good-enough? guess x)
               guess
               (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

或者再改进，不必将输入的参数x传来传去：

```
(define (sqrt x)
    (define (good-enough? guess )
          (< (abs (- (square guess) x)) 0.0001))
    (define (improve guess )
          (average guess (/ x guess)))
```

```
  (define (average x y)
        (/ (+ x y) 2))
  (define (square x)
        (* x x))
  (define (sqrt-iter guess )
        (if (good-enough? guess )
             guess
             (sqrt-iter (improve guess x))))
(sqrt-iter 1.0 ))
```

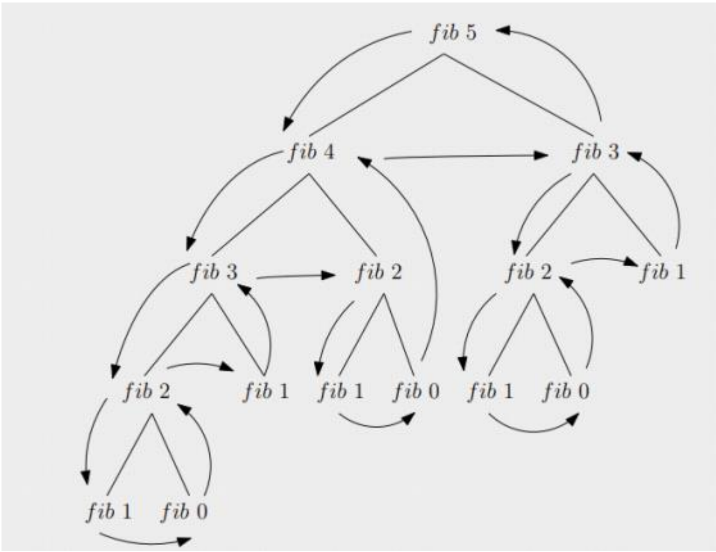
斐波那契数列

2022年12月20日　11:40

递归算法:

```
(define (fib n)
  (cond ( (= n 0) 0)
        ( (= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

例如求 (fib 5) 的过程:



- 求值(fib 0)和(fib 1)：1个
- 求值(fib 2)：(fib 0)+(fib 1)，1+1=2=(fib 3)个
- 求值(fib 3)：(fib 1)+(fib 2)，1+2=3=(fib 4)个
- 求值(fib 4)：(fib 2)+(fib 3)，前者(fib 3)个，后者(fib 4)个，共(fib 3)+(fib 4)=(fib 5)个
- 不难证明，对更大的n，求值(fib n)：(fib n-1)+(fib n)=(fib n+1)个
- 如果可以给出(fib n+1)的数值，我们就可以估算计算(fib n)所需的计算次数的下界

也就是说，(fib n)的计算次数随n增长的速度近似于 (fib n) 的增长速度。所以它连(fib 43)都要算好几十秒。

迭代算法:

```
;arguments: idx, fib(idx), fib(idx+1)
```

```
(define (fib-iter idx curValue nextValue idxBound)
  (if (= idx idxBound)
      curValue
      (fib-iter (+ idx 1)
                nextValue
                (+ curValue nextValue)
                idxBound)))

(define (fib n)
  (fib-iter 0 0 1 n))
```

使用迭代法计算(fib 5)的二维表格如下

idx	curValue=fib(idx)	nextValue=fib(idx+1)	idxBound
0	0	1	5
1	1	0+1=1	5
2	1	1+1=2	5
3	2	1+2=3	5
4	3	2+3=5	5
5	5	3+5=8	5

(fib 5)的计算结果为5

换零钱方式的统计

2022年12月28日　　18:04

问题：现有半美元、四分之一美元、10美分、5美分和1美分共5种硬币。若将1美元换成零钱，共有多少种不同方式？

http://www.cnblogs.com/DarkMaster/p/3903751.html

将总数为a的现金换成n种硬币的不同方式的数目等于

1.将现金数a换成除第一种硬币之外的所有其他硬币的不同方式数目，加上

2.将现金数a-d（至少有一个d面值的硬币）换成所有种类的硬币的不同方式数目，其中的d是第一种硬币的币值

也就是说：1美元换成五种零钱 = （包含半美元的所有换法） + （不包含半美元的换法）

递归必须要有停止条件，在这里的递归停止条件是：

a=0 return 1 （a = 0说明分配完了，得到一种分配方法）

a<0 or n=0 return 0 （a<0或者n=0说明未分配成功）

```
#lang sicp
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0)(= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 50)
        ((= kinds-of-coins 2) 25)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 5)
        ((= kinds-of-coins 5) 1)))
```

求幂

2022年12月29日　　20:37

递归:

```scheme
(define (expt base n)
  (if (= n 0)
      1
      (* base (expt base (- n 1)))))
```

迭代:

```scheme
(define (expt base n)
  (define (expt-iter base curN product nBound)
    (if (= curN nBound)
        product
        (expt-iter base
                   (+ curN 1)
                   (* base product)
                   nBound)))
  (expt-iter base 0 1 n))
```

以(expt 5 4)为例，二维表格如下

base	curN	$product = base^{curN}$	nBound
5	0	1	4
5	1	5*1=5	4
5	2	5*5=25	4
5	3	5*25=125	4
5	4	5*125=625	4

(expt 5 4)的计算结果为625

递归和迭代的幂函数算法的时间增长阶（行数）都是线性的

在算法的思想不变的情况下，递归改为迭代不能加速算法的增长阶

二分法:

如何二分

- 如果已知$a^{\frac{n}{2}}$，如何求a^n？进行一步平方运算即可
- $a^n = (square\ a^{\frac{n}{2}})$，先递归求出$a^{\frac{n}{2}}$的值b，而后(square b)=b*b即为所得
- $a^8 = (square\ a^4) = (square\ (square\ (square\ a)))$

对应的递归等式如下。注意递归等式有三个分支

$$a^n = \begin{cases} 1 & if\ n = 0 \\ (square\ a^{\frac{n}{2}}) & if\ n\ is\ even \\ a * a^{n-1} & otherwise \end{cases}$$

代码：

```
(define (fast-expt b n)
  (define (square x)(* x x))
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

时间增长阶：

以c为偶数时的情况为例：

令k为二分的次数，c=2^k ，k=$\log_2 c$

先展开：(fast-expt a c)-->执行cond判断-->(square (fast-expt b (/ n 2)))-->执行下一次cond判断-->……-->(square ... (square a)每次展开需要2行

展开所需时间 = 2*$\log_2 c$ 。

再收缩：(square ... (square (square a))-->(square ... (square (a*a))-->(square ... (square a^2)-->(square ... ($a^2 * a^2$)-->...每次收缩需要2行

收缩所需时间 = 2*$\log_2 c$ +1。

所以，fast-expt的运行时间为4*log2(c)+1，时间增长阶为Θ($\log_2 c$)，为对数

二分法确实减小了时间增长阶：线性时间 ⇒ 对数时间

最大公约数

2022年12月30日 17:26

最大公约数：两个数的公因子中最大的那个

朴素的算法：

用迭代算法从min(a,b)递减到2，遇到的第一个公约数就是a和b的最大公约数

算法的时间增长阶：线性

例如，如果a和b互质，则计算会从min(a,b)开始扫描，一直扫描到2

如：

```
#lang sicp
(define (f a b)
  (define (f a b c)
    (cond ((= c 1) 1)
          ((and (= 0 (remainder a c)) (= 0 (remainder b c))) c)
          (else (f a b (- c 1)))))
  (f a b (min a b)))

(define (min a b)
  (if (> a b)
      b
      a))
```

欧几里得算法：

定义过程(gcd a b)计算a和b的最大公约数

假定a > b。令a=c*b+d

不妨令x是a和b的最大公约数。如果b和d存在大于x的公因数y， 则y也是a的因数，不成立。因此，b和d的公因数小于等于x

令a=m*x，b=n*x，则m*x=c*n*x+d。因此，d=(m-c*n)*x，b和d的最大公因数为x

a和b的最大公约数=b和d的最大公约数

递归等式：(gcd a b)=(gcd b d)，参数变小了

递归出口：当a是b的倍数时，(gcd a b)=(gcd b 0)=b

例子：gcd(108,40)=gcd(40,28)=gcd(28,12)=gcd(12,4)=gcd(4,0)=4

代码实现：

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

素数检测

2023年1月17日　13:32

方法一：寻找因子

找出数n大于1的最小因子，程序写为检查从2开始的连续整数，看是否能整除n：

```
;找出数n大于1的最小因子
(define (smallest-divisor n)
  (define (find-divisor n test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divides? test-divisor n) test-divisor)
          (else (find-divisor n (+ test-divisor 1)))))
  (find-divisor n 2))

(define (square x)
  (* x x))
;判断是否能整除
(define (divides? a b)
  (= (remainder a b) 0))
;判断是否为素数
(define (prime? n)
  (= n (smallest-divisor n)))
```

红字部分解释：如果n不是素数，有n=a*b，那么a和b肯定是一个小于$\sqrt[2]{n}$，一个大于$\sqrt[2]{n}$，所以如果说当test-divisor增加到$\sqrt[2]{n}$时还没有出现因子，那就可以说明其大于1的最小因子就只有它自己

步数增长阶：该算法只需要在1到$\sqrt{n}$之间检查因子，所以其步数增长阶为$\Theta(\sqrt{n})$

方法二：费马检查

不动点

2023年2月8日 20:15

一个函数f(x)的不动点是一个值a，使得f(a)=a

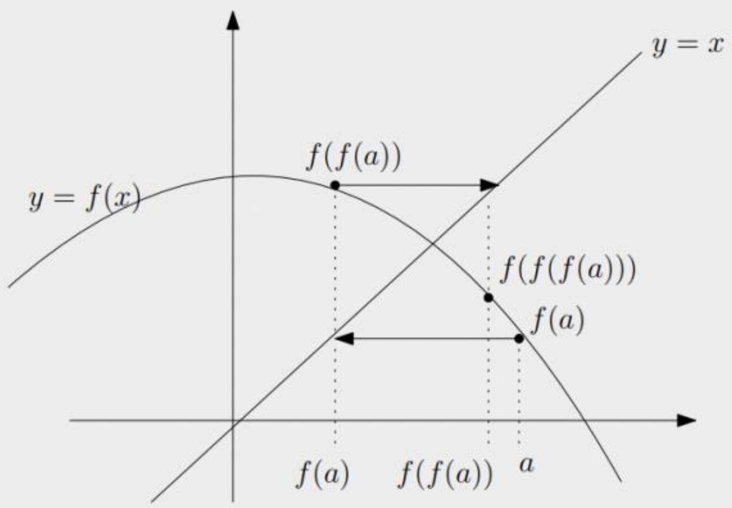
$2x-1$的不动点是1：$2x-1=x$，进而$x=1$

x^2-1的不动点是$\frac{1+\sqrt{5}}{2}$和$\frac{1-\sqrt{5}}{2}$：$x^2-1=x$，进而$x^2-x-1=0$

$\frac{a}{x}$的不动点是$\sqrt{a}$和$-\sqrt{a}$：$\frac{a}{x}=x$，进而$x^2=a$

不动点的计算方法

- 计算a、f(a)、f(f(a))、...，直到收敛
- 最终收敛到y=f(x)和y=x的交点b，此时b=f(b)




```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess))) ;let语句使得(f guess)只需求值一次，而不是两次
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

用不动点计算牛顿法，并计算平方根：

$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$

当算出的某个x_i和x_{i+1}足够接近时，认为x_{i+1}就是方程的根。此时$\frac{f(x_i)}{f'(x_i)}$足够小

$x - \frac{f(x)}{f'(x)}$的不动点就是方程f(x)=0的解

因为此时$x = x - \frac{f(x)}{f'(x)}$

把解方程f(x)=0转化为计算$x - \frac{f(x)}{f'(x)}$的不动点

- 参数为待解方程f
- 构造一个过程newton-transform(x)，求值结果是一个计算$x - \frac{f(x)}{f'(x)}$的过程
- 计算newton-transform过程的不动点即可

(newtons-method g guess)

- 计算方程g(x)=0的解，猜测的初始解为guess
- 由(newton-transform g)负责构造计算$x - \frac{g(x)}{g'(x)}$的过程，由fixed-point负责计算$x - \frac{g(x)}{g'(x)}$的不动点

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x 0.0001)) (g x)) 0.0001)))

(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))

(define (newton-method g guess)
  (fixed-point (newton-transform g) guess))

(define (square x) (* x x))
```

```
(define (sqrt x)
  (newton-method (lambda (guess) (- (square guess) x)) 1.0))
```

注意(lambda (guess) (- (square guess) x))其实就是求平方根的函数：x^2-n

不动点+平均阻尼计算平方根：

fixed-point算法并不总生效

- 如果f(f(x))=x，则数列x,f(x),f(f(x)),...是数列x,f(x),x,f(x),...，无法收敛

这样的情况可能发生

- 例如，计算a的平方根
- a的平方根是$g(x)=\frac{a}{x}$的不动点
- g(g(x))=$\frac{a}{g(x)}=\frac{a}{\frac{a}{x}}=x$

解决方法

- 每一轮不是从x计算f(x)，而是计算一个即足够接近f(x)又有不同的数字
- 例如，计算$\frac{x+f(x)}{2}$

这时计算平方根：

(average-dump f)：求值结果是一个计算$\frac{x+f(x)}{2}$的过程
average-dump称为平均阻尼过程
每一轮使用average-dump处理$\frac{x}{y}$

```
(define (fixed-point f first-guess)
  (define tolerance 0.00001)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (average a b) (/ (+ a b) 2))

(define (average-dump f)
  (lambda (x) (average x (f x))))
```

```
(define (sqrt x)
  (fixed-point (average-dump (lambda (y) (/ x y)))
               1.0))
```

两种方法总结：

注意这几个函数里面都是将y作为猜测值，x为输入值（要求平方根的数）

不动点+average-dump计算平方根$\sqrt{x}$

- 从$\frac{y}{x}$开始
- 做不动点的过程是(average-dump (lambda (y) (/ x y)))
- 求值结果是一个过程，将y映射到$\frac{y+\frac{x}{y}}{2}$

不动点+牛顿法计算平方根$\sqrt{x}$

- 从$y^2 - x$开始
- 做不动点的过程是(newton-transform (lambda(y)(- (square y) x)))
- 求值结果是一个过程，将y映射到$y - \frac{f(y)}{f'(y)} = y - \frac{y^2-x}{2*y} = \frac{y+\frac{x}{y}}{2}$

很巧合地，在计算平方根时，这两个计算方法完全一样

- 后者需给出方程，前者需给出对应的不动点的函数

方法一：不动点计算，使用average-dump处理原函数x=n/x(写成这样不容易误会)

方法二：不动点计算，使用newton-transform处理原方程x^2-n

这两者可以再抽象为一个模式：

使用不动点计算一个经过处理的过程(fixed-point-of-transform g transform guess)

代码：

;不动点+使用transform变换输入函数g+初始猜测为guess：

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))

(define (sqrt1 x)
  (fixed-point-of-transform (lambda (y) (/ x y))
                            average-dump
                            1.0))

(define (sqrt2 x)
  (fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-transform
                            1.0))
```

有理数的算术运算

2023年2月9日　　12:25

引入：
计算机无法准确地保存诸如1/3这样的循环小数
计算机也无法准确地计算0.1+0.2的结果，进而无法比较0.1+0.2和0.5-0.2是否相等
让我们实现一个可以完成这些任务的有理数类型

每个有理数被视为一个分数x/y
`(make-rat n d)`：构造有理数n/d
`(numer x)`：有理数x的分子部分
`(denom x)`：有理数x的分母部分
`(add-rat x y)`：求值结果是有理数x和y的和
`(sub-rat x y)`：求值结果是有理数x和y的差
`(mul-rat x y)`：求值结果是有理数x和y的积
`(div-rat x y)`：求值结果是有理数x和y的商

基于make-rat、numer和denom三个过程，实现其他过程。
一开始思考的时候，我们甚至还不知道make-rat、numer和denom到底如何实现。我们只是假定有人已经实现好了(make-rat n d)，其求值结果是某种东西，里面包含了n和d的信息，并且可以分别使用numer和denom取出这些信息。（之后用(cons n d)来实现）
这就是按愿望思维，先不管基础的东西到底如何去运算。

计算方法如下：

$$\frac{n_1}{d_1}+\frac{n_2}{d_2}=\frac{n_1d_2+n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1}-\frac{n_2}{d_2}=\frac{n_1d_2-n_2d_1}{d_1d_2}$$

$$\frac{n_1}{d_1}\cdot\frac{n_2}{d_2}=\frac{n_1n_2}{d_1d_2}$$

$$\frac{n_1/d_1}{n_2/d_2}=\frac{n_1d_2}{d_1n_2}$$

$$\frac{n_1}{d_1}=\frac{n_2}{d_2}\quad \text{当且仅当}\quad n_1d_2=n_2d_1$$

只要将它们用刚才定义的操作表达出来，再将其make-rat就可以了

进行化简改进前的代码:

```
;将分子分母储存在序对里
(define (make-rat n d) (cons (/ n g) (/ d g))))
(define (numer x) (car x))
(define (denom x) (cdr x))

;加减乘除以及判断相等的实现方法（翻译上图）
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))

(define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y))))

(define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (denom x) (numer y))))

(define (equal-rat? x y)
    (= (* (numer x) (denom y))
       (* (numer y) (denom x))))

;最后这个有理数打印成n/d的形式
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")



;有理数,在定义时除以最大公约数（在构造有理数时化简）
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))

(define (make-rat n d)
  (let ( (g (gcd n d)) )
    (cons (/ n g) (/ d g))))

(define (numer x) (car x))

(define (denom x) (cdr x))
```

;有理数在numer和denom时除以最大公约数（在使用时化简）

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))

(define (make-rat n d)
  (cons n d))

(define (numer x)
  (let ( (g (gcd (car x) (cdr x))) )
    (/ (car x) g)))

(define (denom x)
  (let ( (g (gcd (car x) (cdr x))) )
    (/ (cdr x) g)))
```

使用：

```
(define one-five (make-rat 1 5))
(define three-ten (make-rat 3 10))
(define five-six (make-rat 5 6))

(print-rat one-five)
(print-rat three-ten)
(print-rat five-six)
(print-rat (add-rat one-five three-ten))
(print-rat (mul-rat three-ten five-six))
```

我们有了三个不同的有理数实现，区别在于是否执行约分，以及何时执行约分