暨南大学本科实验报告专用纸

课程名称 运筹学 成绩评定

实验项目名称 求单峰函数最小值 指导老师 吴乐秦

实验项目编号 1 实验项目类型 设计性 实验地点 数学系机房

学生姓名 郭彦培 学号 2022101149

学院 信息科学技术学院 系 数学系 专业 信息管理与信息系统

实验时间 2024 年 3 月 21 日上午 ~3 月 23 日晚上 温度 21℃湿度 85%

目录

1. 实验目的

基于无导数二分法、黄金分割法和 Fibonacci 法实现对单谷函数的求解

2. 实验原理与理论分析

2.1. 二分法

第k步搜索区间$[l_k, r_k]$，微元ε，试探点

$$\begin{aligned} m_{k,1} &= \frac{l_k + r_k}{2} + \varepsilon \\ m_{k,2} &= \frac{l_k + r_k}{2} - \varepsilon \end{aligned} \tag{1}$$

有转移方程:

$$\begin{cases} r_{k+1} = m_k & f(m_{k,1}) > f(m_{k,2}) \\ l_{k+1} = m_k & f(m_{k,1}) \leq f(m_{k,2}) \end{cases} \tag{2}$$

2.2. 黄金分割法

第k步搜索区间$[l_k, r_k]$，缩短率$\tau = \frac{\sqrt{5}-1}{2}$，试探点

$$\begin{aligned} m_{k,1} &= l_k + (1-\tau)(r_k - l_k) \\ m_{k,2} &= l_k + \tau(r_k - l_k) \end{aligned} \tag{3}$$

有转移方程:

$$\begin{cases} r_{k+1} = m_{k,1} & f(m_{k,1}) > f(m_{k,2}) \\ l_{k+1} = m_{k,2} & f(m_{k,1}) \leq f(m_{k,2}) \end{cases} \tag{4}$$

特别的，对于新的试探点，有

$$\begin{aligned} m_{k+1,1} &= l_k + \tau^2(r_k - l_k) \\ \tau^2 &= 1 - \tau \end{aligned} \tag{5}$$

因此新试探点

$$m_{k+1,1} = m_{k,2} \tag{6}$$

不需要重新计算。另一侧试探点的情况于此同理。

2.3. 斐波那契法

斐波那契数列满足:

$$\begin{aligned} &F_0 = F_1 = 1, \\ &F_{k+1} = F_{k-1} + F_k, \qquad k = 1, 2, ... \end{aligned} \tag{7}$$

则令缩短率$\tau = \frac{F(n-k)}{F(n-k+1)}$

同样有试探点

$$\begin{aligned} m_{k,1} &= l_k + (1-\tau)(r_k - l_k) \\ m_{k,2} &= l_k + \tau(r_k - l_k) \end{aligned} \tag{8}$$

转移方程:

$$\begin{cases} r_{k+1} = m_{k,1} & f(m_{k,1}) > f(m_{k,2}) \\ l_{k+1} = m_{k,2} & f(m_{k,1}) \le f(m_{k,2}) \end{cases} \tag{9}$$

特别的，对于新的试探点，有

$$\begin{aligned} m_{k+1,1} &= l_{k+1} + \frac{F_{n-k-1}}{F_{n-k}}(r_{k+1} - l_{k+1}) \\ &= l_{k+1} + \frac{F_{n-k-1}}{F_{n-k}}(m_{k,1} - l_{k+1}) \\ &= l_k + \frac{F_{n-k-1}}{F_{n-k}}\left(l_k + \frac{F_{n-k}}{F_{n-k+1}}(r_k - l_k) - l_k\right) \\ &= l_k + \frac{F_{n-k}}{F_{n-k+1}}(r_k - l_k) = m_{k,2} \end{aligned} \tag{10}$$

因此不需要重复计算。

3. 代码框架

实现一个函数，接受目标函数的函数指针，初始搜索区间，方法选择器，返回求得的单谷函数的最小值。

规定命名空间`lineSearch`内的函数原型

```
double find_mininum(
  double (*func)(double x),// 目标单谷函数
  double l,                // 搜索区间左端点
  double r,                // 搜索区间右端点
  double acc,              // 搜索精度
  int mod                  // 搜索模式
);
```

其中，`func`为目标单谷函数，`l`、`r`分别为初始区间的两个端点，`acc`为搜索精度，`mod`为模式选择。

关于模式选择，命名空间`lineSearch`内提供了三个可选模式:

`BINARY`	无导数二分法
`GOLDEN_RATIO`	黄金分割法
`FIBONACCI`	Fibonacci 法

当参数不合法时，程序会抛出异常，并返回固定值`-1`:

`Illegal Range Execption`	区间不合法
`Unexpection Search Mod Exception`	未知的搜索模式
`Unknown Exception`	其他预料外错误

以下是一些函数调用例子:

```
lineSearch::find_mininum(f,-1,1,0.001,lineSearch::BINARY)
//用二分搜索函数 f 的[-1,1]区间，精度为 0.001
lineSearch::find_mininum(f,114,514,0.0019,lineSearch::FIBONACCI)
//用斐波那契法搜索函数 f 的[114,514]区间，精度为 0.0019
```

4. 核心代码构成

完整代码见 7.附录

4.1. 无导数二分法

```
    double inf = acc /INF_ACC_RATIO,
           //infinitesimal unit 模拟求导极小值
            x; //search index 搜索目标值
    double cul = l,//current left range 当前搜索左端点
           cur = r;//current right range 当前搜索右端点
    while(cur - cul > acc){
        double mid = ( cul + cur ) /2.0;//计算中间值
        if(func(mid + inf) > func(mid - inf)) cur = mid;
        // 若极小值在左侧，则将右边界左移
        else cul = mid;
        // 反之则右移
        x = cul;
    }
    return x;
```

4.2. 黄金分割法

```
    double x;       //search index 搜索目标值
    double cul = l,
           //current left range 当前搜索左端点
           cur = r;
           //current right range 当前搜索右端点
    double otl = func(r - GOLDEN_RATIO_VALUE * (r - l)),
           //overture point left 左侧试探点函数值
           otr = func(l + GOLDEN_RATIO_VALUE * (r - l));
           //overture point right 右侧试探点函数值
    while(cur - cul > acc){
        if(otl > otr) //最小值在左侧区间
        {
            cul = cur - GOLDEN_RATIO_VALUE * (cur - cul);
            //将区间左端点移动到原左侧黄金分割点处
            otl = otr;
            //原右侧试探点为新区间左侧试探点
            otr = func(cul + GOLDEN_RATIO_VALUE * (cur - cul));
            //计算新的右侧试探点值
        } else {    //反之...
            cur = cul + GOLDEN_RATIO_VALUE * (cur - cul);
            otr = otl;
            otl = func(cur - GOLDEN_RATIO_VALUE * (cur - cul));
        }
        x = cul;
    }
    return x;
```

其中，`GOLDEN_RATIO_VALUE`为黄金分割比，其值为$\frac{\sqrt{5}-1}{2}$

4.3. Fibonacci 法

```
    double x;       //search index 搜索目标值
    double cul = l,
           //current left range 当前搜索左端点
           cur = r;
           //current right range 当前搜索右端点
    double FN = (r - l) / acc;
           //inneed fibonacci value 斐波那契数列最大值
    int N; //total caculate times 最大计算次数
    std::vector<double> Fib(2,1);
                        //Fibonacci array 斐波那契数列

    while(Fib[Fib.size()-1] < FN)//递推计算斐波那契数列
        Fib.push_back(Fib[Fib.size()-1] + Fib[Fib.size()-2]);
    N = Fib.size() -1;

    double otl = func(l + Fib[N-2] / Fib[N] * (r - l)),
           //overture point left 左侧试探点函数值
           otr = func(l + Fib[N-1] / Fib[N] * (r - l));
           //overture point right 右侧试探点函数值

    for(int k = 0;k <= N - 2.0 ;k ++)
    {
        if(otl > otr) //最小值在左侧
        {
            cul = cul + Fib[N -k -2] / Fib[N -k] * (cur -cul);
            //将区间左端点移动到原左侧试探点处
            otl = otr;
            //原右侧试探点为新区间左侧试探点
            otr = func(cul +Fib[N -k -1] /Fib[N -k] *(cur -cul));
            //计算新的右侧试探点值
        } else {//反之...
            cur = cul +Fib[N -k -1] /Fib[N -k] *(cur -cul);
            otr = otl;
            otl = func(cul +Fib[N -k -2] /Fib[N -k] *(cur -cul));
        }
        x = cul;
    }
    return x;
```

5. 正确性测试

完整测试代码见 7.附录

5.1. 测试数据准备

测试用的目标函数为一个在`x`轴平移了`dev`的二次函数，即:

```
double dev = 0.03; // deviation
double f(double a)
{
    return (a - dev) * (a - dev);
}
```

测试程序将随机生成一系列的偏移值`dev`，和对应的合法搜索区间`l`,`r`、准确度`acc`，并分别调用

```
lineSearch::find_mininum(f, l, r, acc, lineSearch::BINARY);
lineSearch::find_mininum(f, l, r, acc, lineSearch::GOLDEN_RATIO);
lineSearch::find_mininum(f, l, r, acc, lineSearch::FIBONACCI);
```

随后分析并输出结果。

规定理论值为`thn`，当前答案为`ans`

下面是 10 次测试的结果，其中当前精准度

$$\mathrm{acc}_{\text{当前}} = \frac{\mathrm{acc}}{|\mathrm{thn} - \mathrm{ans}|} \times 100\% \tag{11}$$

反映了搜索的准确度。其中偏差量

$$\mathrm{dev} = \frac{\max(0, |\mathrm{thn} - \mathrm{ans}| - \mathrm{acc})}{\mathrm{acc}} \times 100\% \tag{12}$$

反应了搜索结果与目标的偏差是否在可接受范围内。

$\mathrm{acc} > 100\%$且$\mathrm{dev} = 0$时可以视为解是可接受的。

5.2. 测试结果

测试次数取 5 时输出如下:

```
----Test Cases1----
< search data > l:-3.38 r:3.96 acc:1e-09
< Theoretical > ans:0.24 acc:inf
[Binary search] ans:0.24 acc:641.04 dev:0%
[0.618  method] ans:0.24 acc:665.21 dev:0%
[  Fibonacci  ] ans:0.24 acc:238.612 dev:0%

----Test Cases2----
< search data > l:-1.76 r:3.4 acc:1e-07
< Theoretical > ans:1.38 acc:inf
[Binary search] ans:1.38 acc:246.724 dev:0%
[0.618  method] ans:1.38 acc:247.304 dev:0%
[  Fibonacci  ] ans:1.38 acc:267.991 dev:0%

----Test Cases3----
< search data > l:-1.64 r:2.58 acc:0.0001
< Theoretical > ans:0.56 acc:inf
[Binary search] ans:0.559956 acc:225.986 dev:0%
[0.618  method] ans:0.559955 acc:221.363 dev:0%
[  Fibonacci  ] ans:0.559919 acc:123.319 dev:0%

----Test Cases4----
< search data > l:-1.12 r:3.38 acc:0.001
< Theoretical > ans:1.24 acc:inf
[Binary search] ans:1.23986 acc:731.429 dev:0%
[0.618  method] ans:1.23937 acc:159.73 dev:0%
[  Fibonacci  ] ans:1.23942 acc:173.462 dev:0%

----Test Cases5----
< search data > l:-2.5 r:2.86 acc:1e-05
< Theoretical > ans:0.74 acc:inf
[Binary search] ans:0.739996 acc:273.067 dev:0%
[0.618  method] ans:0.739998 acc:427.762 dev:0%
[  Fibonacci  ] ans:0.739996 acc:281.095 dev:0%
```

可以看到对于不同的参数，程序的`acc`与`dev`均在可接受范围内，因此可以认为搜索算法实现正确。

6. 各方法不同情况下的性能表现与分析

完整测试代码见 7.附录

6.1. 对于复杂目标函数进行搜索：

这一项测试针对在绝大部分实用数学模型的求解时的场景，即目标函数十分复杂难以计算，需要尽量减少计算函数值的次数，是最主要的应用环境，考察算法解决复杂问题的能力。

6.1.1. 测试结果猜想：

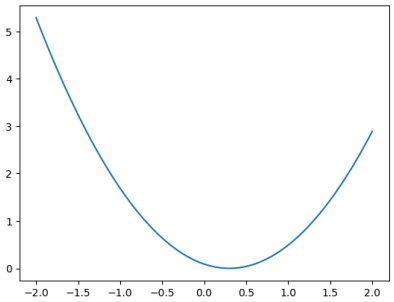
每次迭代搜索次数较少的 Fibonacci 法与黄金分割法将显著快于虽然下降率大但搜索次数较高二分法。

6.1.2. 测试过程：

测试函数：

$$f(x) = (x-0.3)^2 + \min\left(\left(\frac{1}{x-0.3}\right)^{10^5}, 0.001\right) \tag{13}$$

其图像如下：



测试函数性质：每次计算函数值会涉及无法被优化的10^5次乘方运算，可以较好的模拟实际模型中的复杂情况。数学上，$f(x)$在$\mathbb{R}$上为单谷函数，最小值在$x=0.3$处取得，$f(0.3)=0$

测试内容：给定相同的测试参数，分别进行 500 次搜索，统计总时间消耗。

测试参数：$l=-1\times10^5, r=1\times10^5, \text{acc}=1\times10^{-5}$

搜索结果：

```
    [Binary search] ans:0.299996 acc:282.772 dev:0%
    [0.618  method] ans:0.299997 acc:307.922 dev:0%
    [  Fibonacci  ] ans:0.299995 acc:197.102 dev:0%
```

时间测试结果:

```
    [Binary search] cost:10.689s
    [0.618  method] cost:7.233s
    [  Fibonacci  ] cost:7.108s
```

6.1.3. 测试分析:

根据测试结果来看，基本符合预期。二分查找由于大量额外的查找，导致时间消耗高于另外两种搜索，又由于斐波那契法的下降速率略快于黄金分割法，因此最终执行速率略快。

6.2. 对于简单单谷函数，进行大范围区间搜索:

这一项测试在简单场景较为实用，主要考验算法的收敛速度。

6.2.1. 测试结果猜想:

缩短率最小的二分法会是最快的，Fibonacci 次之，而黄金分割最慢。

6.2.2. 测试过程:

测试函数：$f(x) = (x - 0.48)^2$

测试函数性质：在$\mathbb{R}$上为单谷函数，最小值在$x = -0.48$处取得，$f(-0.48) = 0$

测试内容：给定相同的测试参数，分别进行 500 次搜索，统计总时间消耗。

测试参数：$l = -1 \times 10^{100}, r = 1 \times 10^{100}, \text{acc} = 1 \times 10^{-12}$

搜索结果:

```
    [Binary search] ans:-0.48 acc:40941.8 dev:0%
    [0.618  method] ans:-0.48 acc:682.364 dev:0%
    [  Fibonacci  ] ans:-0.48 acc:162.658 dev:0%
```

时间测试结果:

```
[Binary search] cost:0.001s
[0.618  method] cost:0.003s
[  Fibonacci  ] cost:0.026s
```

6.2.3. 测试分析：

从测试结果来看，二分法最快，与预期相同；而斐波那契法比黄金分割法慢，与预期不符。

猜测原因在于搜索范围极端大时斐波那契法需要申请额外的大量内存用于存储计算过程中需要使用的斐波那契数列，导致算法常数复杂度较大，且目标函数较为简单，导致节省查询次数收益很小。

6.3. 对于小区间和一般函数的快速搜索：

这一项测试针对在小规模数据的密集计算中常用的场景，即搜索区间较小，精度要求较低，函数较为简单。考察算法初期下降速率与常数级别优化。

6.3.1. 测试结果猜想：

黄金分割最快，二分法次之，斐波那契法最慢

6.3.2. 测试过程：

测试函数：$f(x)=\begin{cases}\sqrt{x+1.03} & (x\geq -0.48)\\ \sqrt{0.07-x} & (x<-0.48)\end{cases}$

测试函数性质：在$\mathbb{R}$上为单谷函数，最小值在$x=-0.48$处取得，$f(-0.48)\simeq 0.7416198463187$

测试内容：给定相同的测试参数，分别进行1×10^6次搜索，统计总时间消耗。

测试参数：$l=-1, r=1, \mathrm{acc}=1\times 10^{-3}$

搜索结果：

```
[Binary search] ans:-0.480469 acc:213.333 dev:0%
[0.618  method] ans:-0.48046 acc:217.357 dev:0%
[  Fibonacci  ] ans:-0.48065 acc:153.81 dev:0%
```

时间测试结果：

```
    [Binary search] cost:0.191s
    [0.618  method] cost:0.159s
    [  Fibonacci  ] cost:8.19s
```

6.3.3. 测试分析:

从测试结果来看，黄金分割法略微领先二分法，斐波那契法遥遥落后，与预期吻合。黄金分割的轻量级和计算次数较少的优势在本组样例中体现较为明显，而二分虽然下降最快，但计算次数有一定劣势。斐波那契则由于大量额外的常数计算导致在小规模样例中不讨好。

6.4. 测试结果总结

对于一般的函数而言，二分法与黄金分割法的速度已经足够快，而理论计算函数次数最小的斐波那契则由于额外消耗导致性能表现较差。而对于大型模型求解时的场景，二分法速度不甚理想，而黄金分割法与斐波那契法的差距较小。

因此，绝大部分情况下，选用实现简单、速度较快的黄金分割法是较为理想的选择。

同时，针对不同的搜索场景，需要灵活选择不同的算法。以下是一些常用场景的选项:

- 在选取某大型物理过程模拟的一个量时，若确定仿真目标与所确定的量呈现单谷函数，则可以选用斐波那契法，对应测试`6.1.`
- 在求解某数学问题时，需要快速地从某单谷的超多项函数（可能是泰勒展开等）中找到数值解，同样选用斐波那契法。
- 如果上述数学问题的函数值较容易计算，但搜索范围很大，则可以选择下降率最大的二分法，对应测试`6.2.`
- 在某些网络与数据库算法中需要快速确定某个参数或是定位内存中的元素时，则可以选择常数时间较少的黄金分割法，对应测试`6.3.`

7. 附录

7.1. 主体部分：`core.h`

```
/**
 * @author
 * JNU,Guo Yanpei,github@GYPpro
 * https://github.com/GYPpro/optimizeLec
 * @file
 * /optimizeLec/WEEK1/core.h
 * @brief
 * a functional lib solving linner search problem
 */

#ifndef _LINE_SEARCH_
#define _LINE_SEARCH_

#include <math.h>
#include <algorithm>
#include <vector>

namespace lineSearch{

  const int BINARY = 1;     //binary search method
  const int GOLDEN_RATIO = 2; //0.618 method
  const int FIBONACCI = 3;    //Fibonacci method

  const double GOLDEN_RATIO_VALUE = (sqrt(5.0) - 1.0)/2.0;

  double INF_ACC_RATIO = 10;    //the ratio between infinitesimal
unit in binary search method and given accuracy, which means inf
= acc / INF_ACC_RATIO;

  /**
   * @brief
   * caculate 2 value in Fibonacci Array (fb(n) and fb(n))at same
time using matrix multiplication and binary lifting method
   * use cacu_Fibonacci(n).first to get fb(n)
   */
  std::pair<int, int> cacu_Fibonacci(int n)
  {

    if (n == 0) return std::pair<int,int>(0,1);
```

```
    auto p = cacu_Fibonacci(n / 2);
    int c = p.first * (2 * p.second - p.first);
    int d = p.first * p.first + p.second * p.second;
    if (n & 1)
      return std::pair<int,int>(d, c + d);
    else
      return std::pair<int,int>(c, d);
  }

  /**
   * @attention
   * function will return -1 and throw exceptions while getting
illegal input
   * given a illegal input function is undefined behavior
   * @brief
   * Finding the minnum num of the input unimodal function at a
given accuracy
   * Usign segmentation interval method
   */
  double find_mininum(
    double (*func)(double x),// inputed unimodal function
    double l,         // left range
    double r,         // right range
    double acc,         // Search accuracy
    int mod           // Search mod
  ) {
    if (l > r) {throw "Illegal Range Execption";return -1;}
    switch (mod)
    {

    case BINARY:
    {
      double inf = acc /INF_ACC_RATIO, //infinitesimal unit
           x;               //search index
      double cul = l,//current left range while searching
           cur = r;//current right range while searching
      while(cur - cul > acc){
        double mid = ( cul + cur ) /2.0;
        if(func(mid + inf) > func(mid - inf)) cur = mid;//the
mininum located at the left range of middle value
        else cul = mid;//otherwise
        x = cul;
```

```
      }
      return x;
    } break;

    case GOLDEN_RATIO:
    {
      double x;                          //search index
      double cul = l,                    //current left range
while searching
           cur = r;                    //current right range while
searching
      double otl = func(r - GOLDEN_RATIO_VALUE * (r - l)),//
overture point left
           otr = func(l + GOLDEN_RATIO_VALUE * (r - l));//
overture point right
      while(cur - cul > acc){
        if(otl > otr) //the mininum located at the right range of
left overture point
        {
          cul = cur - GOLDEN_RATIO_VALUE * (cur - cul);
          otl = otr;
          otr = func(cul + GOLDEN_RATIO_VALUE * (cur - cul));
        } else {    //otherwise
          cur = cul + GOLDEN_RATIO_VALUE * (cur - cul);
          otr = otl;
          otl = func(cur - GOLDEN_RATIO_VALUE * (cur - cul));
        }
        x = cul;
      }
      return x;
    } break;

    case FIBONACCI:
    {
      double x;              //search index
      double cul = l,        //current left range while searching
           cur = r;        //current right range while searching
      double FN = (r - l) / acc; //inneed fibonacci value
      int N;             //total caculate times

      std::vector<double> Fib(2,1); //Fibonacci array
      while(Fib[Fib.size()-1] < FN)
```

```
        Fib.push_back(Fib[Fib.size()-1] + Fib[Fib.size()-2]);
      N = Fib.size() -1;

      double otl = func(l + Fib[N-2] / Fib[N] * (r - l)), //
overture point left
           otr = func(l + Fib[N-1] / Fib[N] * (r - l)); //
overture point right

      for(int k = 0;k <= N - 2.0 ;k ++)
      {
        // std::cout << cul << " " << cur << " k:" << k << "\n";
        if(otl > otr) //the mininum located at the right range of
left overture point
        {
          cul = cul + Fib[N - k - 2] / Fib[N - k] * (cur - cul);
          //move left range to left overture point
          otl = otr;
          otr = func(cul + Fib[N - k - 1] / Fib[N - k] * (cur -
cul));
        } else {
          cur = cul + Fib[N - k - 1] / Fib[N - k] * (cur - cul);
          otr = otl;
          otl = func(cul + Fib[N - k - 2] / Fib[N - k] * (cur -
cul));
        }
        x = cul;
      }
      return x;
    } break;

    default:
    {
      throw "Unexpection Search Mod Exception";
      return -1;
    }
      break;
    }
    throw "Unknown Exception";
    return -1;
  }
}
#endif
```

7.2. 正确性测试：`TOFtest.cpp`

```
/**
 * @file TOFtest.cpp
 * @brief True or False test
 */

#include <iostream>
#include <stdlib.h>
#include "core.h"
using namespace std;

int tc = 10; // test case
double dev = 0.03; // deviation

double f(double a)
{
    return (a - dev) * (a - dev);
}
int main()
{
    double l = -1,
           r = 1.0,
           acc = 0.001;
    double thn = 0.03;
    srand(1145);

    auto randint = [](int l,int r) -> int{
        return (int)((rand() * (r - l))/(RAND_MAX) + l);
    };

    while(tc --){

        dev = ((double)randint(1,100))/50.0;
        thn = dev;
        l = dev - ((double)randint(100,200))/50.0;
        r = dev + ((double)randint(100,200))/50.0;
        acc = pow(0.1,abs(randint(1,10)));

        cout << "\n----Test Cases" << 10 - tc<< "----\n";

        cout << "< search data > l:" << l << " r:" << r << "
acc:" << acc <<  "\n";
```

```
        cout << "< Theoretical > ans:" << thn << " acc:"
             << "inf\n";

        double ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
        cout << "[Binary search] ans:" << ans << " acc:" <<
(acc / abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn -
ans) - acc) / acc * 100 << "%\n";
        ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
        cout << "[0.618  method] ans:" << ans << " acc:" <<
(acc / abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn -
ans) - acc) / acc * 100 << "%\n";
        ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
        cout << "[  Fibonacci  ] ans:" << ans << " acc:" <<
(acc / abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn -
ans) - acc) / acc * 100 << "%\n";
    }
    system("pause");
}
```

7.3. 复杂函数测试：`CMFtest.cpp`

```
/**
 * @file CMFtest.cpp
 * @brief Complex Model Funtion test
 */


#include <iostream>
#include <stdlib.h>
#include "core.h"
#include <time.h>
using namespace std;

int N = 5000;       // test case
double dev = 0.03; // deviation
int k = 1e5;

double f(double x)
```

```
{
    double pf = x-0.3;
    for(int i = 0;i < k -1;i ++)
        pf *= (x-0.3);
    pf = min(pf,0.001);
    return (x-0.3) * (x-0.3) + pf;
}
int main()
{
    // cout << f();
    int tc = 0;
    double l = -1e5, r = 1e5, acc = 1e-5;
    double thn = 0.3;
    double ans;
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
    cout << "[Binary search] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
    cout << "[0.618  method] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
    cout << "[  Fibonacci  ] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    int begin = clock();
    while (N > tc++)
    {

        ans += lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
    }
    int end = clock();
    tc = 0;
    cout << "[Binary search] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    begin = clock();
    while (N > tc++)
```

```
    {

        ans +=lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
    }
    end = clock();
    tc = 0;
    cout << "[0.618  method] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans +=lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
    }
    end = clock();
    tc = 10;
    cout << "[  Fibonacci  ] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    cout << ans << "\n";
    system("pause");
}
```

7.4. 大区间测试：`LGAtest.cpp`

```
/**
 * @file LGAtest.cpp
 * @brief Large Arrange test
 */

#include <iostream>
#include <stdlib.h>
#include "core.h"
#include <time.h>
using namespace std;

int N = 500;        // test case
double dev = 0.03; // deviation

double f(double x)
{
```

```
    return (x + 0.48) * (x + 0.48);
}
int main()
{
    int tc = 0;
    double l = -1e100, r = 1e100, acc = 1e-12;
    double thn = -0.48;
    double ans;
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
    cout << "[Binary search] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
    cout << "[0.618  method] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
    cout << "[  Fibonacci  ] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    int begin = clock();
    while (N > tc++)
    {

        ans += lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
    }
    int end = clock();
    tc = 0;
    cout << "[Binary search] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans +=lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
    }
    end = clock();
```

```
    tc = 0;
    cout << "[0.618  method] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans +=lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
    }
    end = clock();
    tc = 10;
    cout << "[  Fibonacci  ] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    cout << ans << "\n";
    system("pause");
}
```

7.5. 大数量测试：`LNGtest.cpp`

```
/**
 * @file LGNtest.cpp
 * @brief Large Number test
 */

#include <iostream>
#include <stdlib.h>
#include "core.h"
#include <time.h>
using namespace std;

int N = 1e7;       // test case
double dev = 0.03; // deviation

double f(double x)
{
    return (x + 0.48) * (x + 0.48);
}
int main()
{
    int tc = 0;
    double l = -0.5, r = 1.5, acc = 1e-3;
```

```
    double thn = -0.48;
    double ans;
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
    cout << "[Binary search] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
    cout << "[0.618  method] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    ans = lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
    cout << "[  Fibonacci  ] ans:" << ans << " acc:" << (acc /
abs(thn - ans)) * 100 << " dev:" << max(0.0, abs(thn - ans) -
acc) / acc * 100 << "%\n";
    int begin = clock();
    while (N > tc++)
    {

        ans += lineSearch::find_mininum(f, l, r, acc,
lineSearch::BINARY);
    }
    int end = clock();
    tc = 0;
    cout << "[Binary search] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans +=lineSearch::find_mininum(f, l, r, acc,
lineSearch::GOLDEN_RATIO);
    }
    end = clock();
    tc = 0;
    cout << "[0.618  method] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    begin = clock();
    while (N > tc++)
    {
```

```
        ans +=lineSearch::find_mininum(f, l, r, acc,
lineSearch::FIBONACCI);
    }
    end = clock();
    tc = 10;
    cout << "[  Fibonacci  ] cost:" << double(end-begin)/
CLOCKS_PER_SEC << "s" << "\n";
    cout << ans << "\n";
    system("pause");
}
```

7.6. 代码仓库

全部代码、与 x86 可执行程序均同步在本人的`github`：

`https://github.com/GYPpro/optimizeLec`

本次实验报告存放在`/WEE1`文件夹下