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　　求一维函数最小值　　指导老师　吴乐秦　
实验项目编号　　2　　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 4 月 15 日上午 ~4 月 18 日晚上 温度 33℃湿度 95%

目录

1. 实验目的

实现利用迭代方法计算一维函数最小值的自定义函数。函数能处理最基本的异常，并比较这些方法在收敛速度上的表现。

2. 实验原理与理论分析

本次实验选用**最速下降法**，**牛顿法**和**割线法**。

2.1. 最速下降法

对于当前搜索点x_k，有梯度$d_k = -\nabla f(x_k)$。取合适的步长因子$\alpha_k s.t. f(x_k + \alpha_k d_k) < f(x_k)$ 则

$$x_{k+1} = x_k + \alpha_k d_k \tag{1}$$

2.2. 牛顿法

对于二次可微函数$f(x)$，取二次 Taylor 展开

$$f(x_k + s) \approx q(k)(s) = f(x_k) + \nabla f(x_k)^T s + \frac{1}{2} s^T \nabla^2 f(x_k) s \tag{2}$$

将上式右侧极小化，有迭代方程

$$x_{k+1} = x_k - \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) \tag{3}$$

2.3. 割线法

利用两次迭代值x_k，x_{k-1}在导函数图像上与 x 轴形成的交点作为新的迭代点，近似替代牛顿法中导函数的作用，即

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{\nabla f(x_n) - \nabla f(x_{n-1})} \nabla f(x_n) \tag{4}$$

3. 代码框架

编码利用 `C++` 完成，遵循 `C++17` 标准

规定命名空间 `lineSearch` 内的函数原型

```
std::pair<double,double> find_mininum(
  double (*func)(double x),  // 原函数
  double (*dfunc)(double x), // 一阶导函数
  double l = -_inf,          // 下界
  double r = _inf,           // 上界
  double acc = _acc,         // 搜索精度
  double x = 0.0,            // 初始点
  int mod = DESCENT          // 搜索方法
  double (*ddfunc)(double x) = nullptr,// 二阶导函数(可选)
)
```

其中:

参数	用途	默认值
`func`	目标优化函数	无，必须提供
`dfunc`	目标函数一阶导	无，必须提供
`ddfunc`	目标函数二阶导	空函数
`l`	函数下界	10^{299}
`r`	函数上界	-10^{299}
`acc`	搜索精度	10^{-3}
`x`	初始搜索点	0
`mod`	搜索模式	`DESCENT`（最速下降法）

返回值为一个 `std::pair<double,double>` 类型对象，分别存储了搜索到的 x_{km} 与对应的最小函数值 $f_{\min}$

关于模式选择，命名空间 `ODSearch` 内提供了三个可选模式:

`DESCENT`	最速下降法
`NEWTON`	牛顿法
`SECANT`	割线法

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
`Illegal Initial Value`	不合法的初始搜索点
`Derivative Function NOT Provided`	选择牛顿法时未提供二阶导
`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：

```
auto ans = ODSearch::find_mininum(f, df, l, r, acc, 0.0,
ODSearch::DESCENT);
//用最速下降法进行搜索
lineSearch::find_mininum(f,df,114,514,0.0019,0.0,lineSearch::SECANT,
ddf)
//用割线法搜索函数 f 的[114,514]区间, 从初始值 x 出发, 精度为 0.0019
```

4. 核心代码构成

完整代码见 7.附录

4.1. 最速下降法

```
    double alpha = 0.1;//初始步长因子
    double curx = x;//当前搜索点
    double fmin = func(x);//当前函数值最小值
    double grad = dfunc(x);//当前梯度

    while(abs(grad) > acc)
    {
      //二分线性搜索确定可选步长因子
      while(!(func(curx - alpha * grad) < func(curx)))
        alpha = alpha / 2.0;
      fmin = func(curx - alpha * grad);
      curx -= alpha * grad;
      grad = dfunc(curx);
      alpha = 0.1;
    }
    return {curx,fmin};
```

4.2. 牛顿法

```
/*
 *@brief 计算近似二阶泰勒的 build in lambda function
*/
auto Taylor = [&](double xk) -> double
{
  return dfunc(xk) / (ddfunc(xk) * ddfunc(xk));
};

double curx = x;//当前搜索点
double fmin = func(x);//当前函数值最小值
double grad = dfunc(x);//当前梯度

while(abs(grad) > acc)
{
  fmin = func(curx - Taylor(curx));
  curx -= Taylor(curx);
  grad = dfunc(curx);
}
return {curx,fmin};
```

4.3. 割线法

```
double curx = x;//当前搜索点
double prfx = (l + x)/ 2.0;//上一个搜索点
double fmin = func(x);//当前函数值最小值
double grad = dfunc(x);//当前梯度

/*
 *@brief 计算替代二阶导的割线斜率的 build in lambda function
*/
auto getSec = [&]() -> double{
  return (curx - prfx) * dfunc(curx) / (dfunc(curx) -
dfunc(prfx));
};


while(abs(grad) > acc)
{
```

```
        fmin = func(curx - getSec());
        curx -= getSec();
        grad = dfunc(curx);
    }
    return {curx,fmin};
```

5. 正确性测试

见附录 `TOFtest.cpp`

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

```
ODSearch::find_mininum(f,df,l,r,acc,0.0, ODSearch::DESCENT);
ODSearch::find_mininum(f,df,l,r,acc,0.0, ODSearch::NEWTON, ddf);
ODSearch::find_mininum(f,df,l,r,acc,0.0, ODSearch::SECANT, ddf);
```

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

$$\mathrm{acc}_{\text{当前}} = \frac{\mathrm{acc}}{|\mathrm{thn} - \mathrm{ans}|} \times 100\% \tag{5}$$

反映了搜索的准确度。其中偏差量

$$\mathrm{dev} = \frac{\max(0, |\mathrm{thn} - \mathrm{ans}| - \mathrm{acc})}{\mathrm{acc}} \times 100\% \tag{6}$$

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:

```
Cases1----
< search data > l:-3.38 r:3.96 acc:1e-09
< Theoretical > ans:0.24 acc:inf
[DESCENT] ans:0 at:0.24 acc:inf dev:0%
[NEWTON ] ans:1.9984e-19 at:0.24 acc:223.696 dev:0%
[SECANT ] ans:0 at:0.24 acc:inf dev:0%

----Test Cases2----
< search data > l:-1.76 r:3.4 acc:1e-07
< Theoretical > ans:1.38 acc:inf
[DESCENT] ans:0 at:1.38 acc:inf dev:0%
[NEWTON ] ans:1.69145e-15 at:1.38 acc:243.148 dev:0%
[SECANT ] ans:1.97215e-31 at:1.38 acc:2.2518e+10 dev:0%

----Test Cases3----
< search data > l:-1.64 r:2.58 acc:0.0001
< Theoretical > ans:0.56 acc:inf
[DESCENT] ans:0 at:0.56 acc:inf dev:0%
[NEWTON ] ans:1.16825e-09 at:0.559966 acc:292.571 dev:0%
[SECANT ] ans:0 at:0.56 acc:inf dev:0%

----Test Cases4----
< search data > l:-1.12 r:3.38 acc:0.001
< Theoretical > ans:1.24 acc:inf
[DESCENT] ans:0 at:1.24 acc:inf dev:0%
[NEWTON ] ans:9.16481e-08 at:1.2397 acc:330.323 dev:0%
[SECANT ] ans:4.93038e-32 at:1.24 acc:4.5036e+14 dev:0%

----Test Cases5----
< search data > l:-2.5 r:2.86 acc:1e-05
< Theoretical > ans:0.74 acc:inf
[DESCENT] ans:0 at:0.74 acc:inf dev:0%
[NEWTON ] ans:7.96863e-12 at:0.739997 acc:354.249 dev:0%
[SECANT ] ans:0 at:0.74 acc:inf 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:

见附录 `CMFtest.cpp`

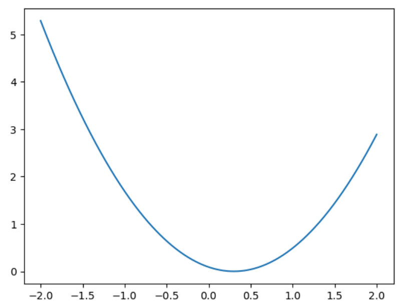
这一项测试针对在绝大部分实用数学模型的求解时的场景，即目标函数十分复杂难以计算，需要尽量减少计算函数值的次数，是最主要的应用环境，考察算法解决复杂问题的能力。

6.1.1. 测试过程:

测试函数:

$$f(x) = (x - 0.3)^2 + \min\left(\left(\frac{1}{x - 0.3}\right)^{10^5}, 0.001\right) \tag{7}$$

图像如下:



其导函数与二阶导函数

$$\nabla f(x) \simeq 2x - 0.6\min\left(\left(\frac{1}{x - 0.3}\right)^{10^5}, 0.001\right) \tag{8}$$

$$\nabla^2 f(x) \simeq 2 + \min\left(\left(\frac{1}{x - 0.3}\right)^{10^5}, 0.001\right) \tag{9}$$

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 50 次搜索，统计总时间消耗。

测试参数：$l = -1 \times 10^5, r = 1 \times 10^5, x_0 = 15, \mathrm{acc} = 1 \times 10^{-5}$

搜索结果正确无误：

```
[DESCENT] ans:0 at:0.3 acc:inf dev:0%
[NEWTON ] ans:9.31323e-12 at:0.300003 acc:327.68 dev:0%
[SECANT ] ans:0 at:0.3 acc:inf dev:0%
```

时间测试结果：

```
[DESCENT] cost:1.98s
[NEWTON ] cost:1.101s
[SECANT ] cost:0.028s
```

各项迭代次数：

```
[DESCENT] tc:74
[NEWTON ] tc:22
[SECANT ] tc:21
```

6.1.2. 测试分析：

测试结果比较符合预期，牛顿法由于其对二阶导函数的频繁使用降低了其速度，而割线法兼顾了较小的迭代次数和较少的调用次数。

6.2. 对于简单函数的快速搜索：

见附录`LGNtest.cpp`

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。

6.2.1. 测试过程：

测试函数：

$$f(x) = x + \frac{0.35}{x} \tag{10}$$

其导函数与二阶导函数为：

$$\nabla f(x) = \frac{20x^2 - 7}{20x^2} \tag{11}$$

$$\nabla^2 f(x) = \frac{7}{10x^3} \tag{12}$$

测试函数性质：在$\mathbb{R}^+$上为单谷函数，最小值在$x \simeq 0.5916079778272$处取得，$f(-0.48) \simeq 1.1832159566199$

测试内容：给定相同的测试参数，分别进行1×10^5次搜索，统计总时间消耗。

测试参数：$l = 0, r = 10^3, x_0 = 0.3, \mathrm{acc} = 1 \times 10^{-3}$

搜索结果正确无误：

```
[DESCENT] ans:1.18322 at:0.591313 acc:338.813 dev:0%
[NEWTON ] ans:1.18322 at:0.591347 acc:382.695 dev:0%
[SECANT ] ans:1.18322 at:0.591321 acc:349.049 dev:0%
```

时间测试结果：

```
[DESCENT] cost:0.023s
[NEWTON ] cost:0.074s
[SECANT ] cost:0.083s
```

各项迭代次数：

```
[DESCENT] tc:114
[NEWTON ] tc:94
[SECANT ] tc:138
```

6.2.2. 测试分析：

从测试结果来看，在小规模函数与小规模区间下，牛顿法与割线法速度相似，最速下降法虽迭代次数略大于牛顿法，但由于其较小的计算复杂度，时间上仍占据优势

7. 附录

7.1. 代码

7.1.1. 核心`core.h`

```
```

```
/**
 * @author
 * JNU,Guo Yanpei,github@GYPpro
 * https://github.com/GYPpro/optimizeLec
 * @file
 * /optimizeLec/WEEK2/core.h
 * @brief
 * a functional lib solving One-dimensional search
 */

#ifndef _ONE_DIMENSIONAL_SEARCH_
#define _ONE_DIMENSIONAL_SEARCH_

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

#include <iostream>


namespace ODSearch{

  const int DESCENT = 1;      //最速下降法
  const int NEWTON = 2;       //牛顿法
  const int SECANT = 3;         //割线法

  const double _inf = 1e299;    //最坏边界
  const double _acc = 1e-3;   //默认精度



  // /*
  //   * @britf
  //   * Set Descent Rate of DESCTNE method
  //   * */
  // void setDescentAlpha(double _a)
  // {
  //   alpha = _a;
  // }

      /**
    * @attention
```

```
   * function will return -1 and throw exceptions while getting
illegal input
   * @brief
   * Finding the minnum num of a One-dimensional function
   */
  std::pair<double,double> find_mininum(
    double (*func)(double x),  // 原函数
    double (*dfunc)(double x), // 一阶导函数
    double l = -_inf,          // 下界
    double r = _inf,           // 上界
    double acc = _acc,           // 搜索精度
        double x = 0.0,                   // 初始点
    int mod = DESCENT,           // 搜索方法
    double (*ddfunc)(double x) = nullptr// 二阶导函数(可选)
  ) {
    if (l > r) {throw "Illegal Range Execption";return {-1,-1};}
        if (x < l || x > r) {throw "Illegal Initial Value";return
{-1,-1};}
        if (mod == NEWTON && ddfunc == nullptr) {throw
"Derivative Function NOT Provided";return {-1,-1};}
        switch (mod)
        {
        case DESCENT:
        {
      double alpha = 0.1;//初始步长因子
      double curx = x;//当前搜索点
      double fmin = func(x);//当前函数值最小值
      double grad = dfunc(x);//当前梯度

      // int tc = 0;
      while(abs(grad) > acc)
      {
        //二分线性搜索确定可选步长因子
        while(!(func(curx - alpha * grad) < func(curx)))
          alpha = alpha / 2.0;
        fmin = func(curx - alpha * grad);
        curx -= alpha * grad;
        grad = dfunc(curx);
        alpha = 0.1;
        // tc ++;
      }
```

```
    // std::cout << "tc:" << tc << "\n";
    return {curx,fmin};
      } break;

      case NEWTON:
      {
    /*
     *@brief 计算近似二阶泰勒的 build in lambda function
    */
    auto Taylor = [&](double xk) -> double
    {
      return dfunc(xk) / (ddfunc(xk) * ddfunc(xk));
    };

    double curx = x;//当前搜索点
    double fmin = func(x);//当前函数值最小值
    double grad = dfunc(x);//当前梯度

    // int tc = 0;
    while(abs(grad) > acc)
    {
      fmin = func(curx - Taylor(curx));
      curx -= Taylor(curx);
      grad = dfunc(curx);
      // tc ++;
    }
    // std::cout << "tc:" << tc << "\n";
    return {curx,fmin};
      } break;

      case SECANT:
      {

    double curx = x;//当前搜索点
    double prfx = (l + x)/ 2.0;//上一个搜索点
    double fmin = func(x);//当前函数值最小值
    double grad = dfunc(x);//当前梯度

    /*
     *@brief 计算替代二阶导的割线斜率的 build in lambda function
    */
    auto getSec = [&]() -> double{
```

```
        return (curx - prfx) * dfunc(curx) / (dfunc(curx) -
dfunc(prfx));
      };

      // int tc = 0;

      while(abs(grad) > acc)
      {
        fmin = func(curx - getSec());
        curx -= getSec();
        grad = dfunc(curx);
        // tc ++;
      }
      // std::cout << "tc:" << tc << "\n";
      return {curx,fmin};

        } break;


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

7.1.2. 测试代码

7.1.2.1. `TOFtest.cpp`

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

#include <iostream>
```

```cpp
#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);
}
double df(double a)
{
    return 2 * a - 2 * dev;
}
double ddf(double a)
{
    return 2.0;
}
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";

        auto ans = ODSearch::find_mininum(f, df, l, r, acc, 0.0,
ODSearch::DESCENT);
        cout << "[DESCENT] ans:" << ans.second << " at:" <<
ans.first << " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.0,
ODSearch::NEWTON, ddf);
        cout << "[NEWTON ] ans:" << ans.second << " at:" <<
ans.first << " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.0,
ODSearch::SECANT, ddf);
        cout << "[SECANT ] ans:" << ans.second << " at:" <<
ans.first << " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
    }
    system("pause");
}
```

7.1.2.2. `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 = 50;       // 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;
}
double df(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 2 * x - 0.6;
}
double ddf(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 2.0 + pf;
}
int main()
{
    // cout << f();
    int tc = 0;
    double l = -1e5, r = 1e5, acc = 1e-5;
    double thn = 0.3;

    auto ans = ODSearch::find_mininum(f, df, l, r, acc, 15,
ODSearch::DESCENT);
    cout << "[DESCENT] ans:" << ans.second << " at:" <<
ans.first<< " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
```

```
    ans = ODSearch::find_mininum(f, df, l, r, acc, 15,
ODSearch::NEWTON, ddf);
    cout << "[NEWTON ] ans:" << ans.second << " at:" <<
ans.first<< " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
    ans = ODSearch::find_mininum(f, df, l, r, acc, 15,
ODSearch::SECANT, ddf);
    cout << "[SECANT ] ans:" << ans.second << " at:" <<
ans.first<< " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
    int begin = clock();
    while (N > tc++)
    {

        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.5,
ODSearch::DESCENT);
    }
    int end = clock();
    tc = 0;
    cout << "[DESCENT] cost:" << double(end-begin)/CLOCKS_PER_SEC
<< "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.5,
ODSearch::NEWTON, ddf);
    }
    end = clock();
    tc = 0;
    cout << "[NEWTON ] cost:" << double(end-begin)/CLOCKS_PER_SEC
<< "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.5,
ODSearch::SECANT, ddf);
    }
    end = clock();
```

```
    tc = 10;
    cout << "[SECANT ] cost:" << double(end-begin)/CLOCKS_PER_SEC
<< "s" << "\n";
    cout << ans.first << "\n";
    system("pause");
}
```

7.1.2.3. `LGNtest.cpp`

```
/**
 * @file LGNtest.cpp
 * @brief 大量数据测试
 */

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

int N = 1e5;       // test case
double dev = 0.5916079778; // deviation

double f(double x)
{
    return x + 0.35/x;
}
double df(double x)
{
    return (20.0 * x * x - 7.0) / (20.0 * x * x);
}
double ddf(double x)
{
    return 7.0 / (10.0 * x * x * x);
}
int main()
{
    int tc = 0;
    double l = 0, r = 1.5, acc = 1e-3;
    double thn =0.5916079778;
    // double ans;
```

```
    auto ans = ODSearch::find_mininum(f, df, l, r, acc, 0.2,
ODSearch::DESCENT);
    cout << "[DESCENT] ans:" << ans.second << " at:" <<
ans.first<< " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
    ans = ODSearch::find_mininum(f, df, l, r, acc, 0.2,
ODSearch::NEWTON, ddf);
    cout << "[NEWTON ] ans:" << ans.second << " at:" <<
ans.first<< " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
    ans = ODSearch::find_mininum(f, df, l, r, acc, 0.2,
ODSearch::SECANT, ddf);
    cout << "[SECANT ] ans:" << ans.second << " at:" <<
ans.first<< " acc:" << (acc / abs(thn - ans.first)) * 100 << "
dev:" << max(0.0, abs(thn - ans.first) - acc) / acc * 100 << "%
\n";
    int begin = clock();
    while (N > tc++)
    {

        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.5,
ODSearch::DESCENT);
    }
    int end = clock();
    tc = 0;
    cout << "[DESCENT] cost:" << double(end-begin)/CLOCKS_PER_SEC
<< "s" << "\n";
    begin = clock();
    while (N > tc++)
    {

        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.5,
ODSearch::NEWTON, ddf);
    }
    end = clock();
    tc = 0;
    cout << "[NEWTON ] cost:" << double(end-begin)/CLOCKS_PER_SEC
<< "s" << "\n";
    begin = clock();
    while (N > tc++)
```

```
    {

        ans = ODSearch::find_mininum(f, df, l, r, acc, 0.5,
ODSearch::SECANT, ddf);
    }
    end = clock();
    tc = 10;
    cout << "[SECANT ] cost:" << double(end-begin)/CLOCKS_PER_SEC
<< "s" << "\n";
    cout << ans.first << "\n";
    system("pause");
}
```

7.2. 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`/WEE2`文件夹下