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 求二元函数极小值 指导老师 吴乐秦
实验项目编号 3 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 5 月 17 日上午 ~5 月 19 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二元函数极小值的自定义函数。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**和**共轭梯度法**

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. 共轭梯度法

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

$$\mathrm{acc}_{\text{当前}} = \frac{\mathrm{acc}}{|\mathrm{thn} - \mathrm{ans}|} \times 100\% \tag{2}$$

反映了搜索的准确度。其中偏差量

$$\mathrm{dev} = \frac{\max(0, |\mathrm{thn} - \mathrm{ans}| - \mathrm{acc})}{\mathrm{acc}} \times 100\% \tag{3}$$

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

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`

6.1.1. 测试过程：

6.1.2. 测试分析：

6.2. 对于简单函数的快速搜索：

见附录`LGNtest.cpp`

6.2.1. 测试过程：

6.2.2. 测试分析：

7. 附录

7.1. 代码

7.1.1. 核心`core.h`

7.1.2. 测试代码

7.1.2.1. `TOFtest.cpp`

7.1.2.2. `CMFtest.cpp`

7.1.2.3. `LGNtest.cpp`

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