优化算法实现报告

Author: [lvwenlong\_lambda@qq.com](mailto:lvwenlong_lambda@qq.com)

Last Modified: 2016/06/20-10:04:27

## project 简介

这个 project 是杜建洪老师课程中介绍的优化算法的 c++ 实现。实现了如下算法：

* 各种一维查找算法，如斐波那契法、黄金分割法、外推法
* 基于 strong wolfe condition 的不精确线搜索方法
* 梯度下降法（Gradient Descent Method）
* 共轭梯度法（Conjugate Gradient Method）
* 牛顿法（Newton Method）
* 拟牛顿法法（Quasi Newton Method），包括 BFGS 算法与 DFP 算法。
* 单纯形法（Simplex Method）
* 鲍威尔法（Powell Method）

这个 project 使用 **CMake** 来 build，使用者需要在系统中事先安装 CMake，程序是在 ubuntu 操作系统下编写与测试，在 g++ 4.8 版本编译器与clang 3.7 中编译测试成功。CMake 也可以生成 Windows 下 Visual Studio 的工程文件，具体的使用请参考 CMake 手册。

使用者可以使用如下方式来编译：

cd /path/to/this/project/src  
mkdir out\_build  
cd out\_build  
cmake ..  
make  
cd ..

要安装 CMake，在 ubuntu 系统中，可以直接 sudo apt-get install cmake 来执行，在 Windows 系统中，可以去网站下载安装包。在其他系统中，可以查看 CMake 网站相关帮助。

在运行 cmake .. 命令时，可以通过下列的两个命令行选项来控制程序的行为：

* -DWRITE\_LOG=ON/OFF，是否在优化时对每个点进行记录，如果为 OFF 则只记录最终的最优点
* -DDEBUG\_OPTIMIZER=ON/OFF，是否开启debug模式，如果为 ON，则会使用统一的随机数发生器种子，这样保证每次运行，都得到相同的结果。

许多算法都需要矩阵运算，在 project 中，矩阵运算调用 [Eigen](http://eigen.tuxfamily.org/index.php?title=Main_Page) 实现

## 基本数据结构

这个 project 关注的重点在算法运行的迭代次数。因此，并没对算法运行时采用的数据结构进行优化。

用来表示函数输入参数、函数返回结果、以及待优化函数的数据类型定义如下：

// Input parameter for objective function  
typedef std::vector<double> Paras;  
// Result evaluated by objective function  
class Solution { // Para与evaluated result放在一个class中，方便(partial) sort  
 Paras \_solution;  
 std::vector<double> \_violation; // sum of constraint violation  
 double \_fom;  
  
 public:  
 Solution(const Paras& s, const std::vector<double>& cv, double fom) noexcept;  
 Solution() =delete;  
 double fom() const noexcept;  
 double sum\_violation() const noexcept;  
 const std::vector<double>& violations() const noexcept;  
 const Paras& solution() const noexcept;  
 Solution& operator=(const Solution&) =default;  
 bool operator<(const Solution& s) const noexcept   
 {   
 return \_fom < s.fom();   
 }  
 bool operator<=(const Solution& s) const noexcept   
 {   
 return \_fom <= s.fom();   
 }  
};  
// type signature of objective function  
typedef std::function<Solution(const Paras&)> ObjFunc;

使用std::vector<double>来表示待优化函数的输入参数，并将其typedef为Paras。

将输入参数、目标函数的值fom，以及约束violation打包成一个class Solution，这样会带来额外的拷贝开销，但好处是编程时更加方便，比如，可以很方便的对一组函数的解进行 排序，选出最好或者最差的解，如果把输入参数跟目标函数输出分开存储，则如果要对目标函数的解进行排序，则需要额外处理输入参数与目标函数输出的同步问题。

对于目标函数的表示，我采用了 c++11 中函数式编程的特性。在 c++11 中，可以用 lambda expression 来表示一个函数，这样表示的函数可以作为数据处理，可以作为另一个函数的输入参数，也可以作为一个函数的返回值。在这个 project 中，目标函数表示为一个输入为const Paras&，输出类型为Solution的函数。这个函数由用户定义，并作为 optimizer 的构造函数的一个参数。

## 一维优化算法

一维函数优化是优化算法的基本，即使是多元函数，在确定了下一步搜索方向之后，也往往在搜索方向上进行线搜索（line search），在这个 project 中，实现了 Fibonacci 法，黄金分割法和外推这三个优化算法。

首先定义一维函数优化的基类:

class Optimizer1D  
{  
 protected:  
 ObjFunc \_func;  
  
 public:  
 Optimizer1D(ObjFunc func) noexcept : \_func(func) {}  
 virtual Solution optimize() noexcept = 0;  
};

Optimizer1D 的构造函数接受一个 ObjFunc 类型，ObjFunc 即上一节介绍过的表示目标函数的类型，这里的目标函数必须是一维函数，否则，程序可能会出错。

Optimizer1D::optimize()是一个纯虚类，所有继承 Optimizer1D 类的派生类都需要实现这个方法，具体的一维优化算法就实现在这里。

### Fibonacci 法

Fibonacci 法的类型声明如下:

class FibOptimizer : public Optimizer1D  
{  
 const double \_lb;  
 const double \_ub;  
 const size\_t \_iter;  
  
 public:  
 FibOptimizer(ObjFunc f,double lb,double ub,size\_t iter) noexcept;  
 Solution optimize() noexcept;  
 ~FibOptimizer() {}  
};

Fibonacci 法需要提供一个一维目标函数，同时，需要提供搜索的下界与上界，Fibonacci 最终的精度随迭代次数指数下降，因此还需要提供一个迭代次数，设置迭代次数默认为 16。

Fibonacci 的基本思路是，希望在区间 内寻找函数 的最小值，则在 内找两个点 与 ，分别计算 与 ，比较 与 的值，若 ，则说明最小值在 区间内，若 , 则说明最小值在 区间内，然后依此递归。

Fibonacci 法靠 Fibonacci 数列来确定 与 的值，因为迭代次数 已经确定，因此可以事先计算出从 0 到 的 Fibonacci 数列，对于第 i 次迭代（从 0 开始），计算 ，然后，令 ，令 。 Fibonacci 法实现代码如下:

Solution FibOptimizer::optimize() noexcept  
{  
 // 1-D function  
 double a1 = \_lb;  
 double a2 = \_ub;  
 if (a1 > a2)  
 {  
 cerr << "Range error" << endl;  
 exit(EXIT\_FAILURE);  
 }  
 vector<double> fib\_list{1, 1};  
 if (\_iter > 2)  
 {  
 for (size\_t i = 2; i < \_iter + 1; ++i)   
 fib\_list.push\_back(fib\_list[i - 1] + fib\_list[i - 2]);  
 }  
 double y1, y2;  
 for(size\_t i = 0; i < \_iter - 1; ++i)  
 {  
 const double f1 = fib\_list[\_iter - 1 - i];  
 const double f2 = fib\_list[\_iter - i];  
 const double rate = f1 / f2;  
 const double a3 = a2 - rate \* (a2 - a1);  
 const double a4 = a1 + rate \* (a2 - a1);  
 const double y3 = \_func({a3}).fom();  
 const double y4 = \_func({a4}).fom();  
 if (y3 < y4)  
 {  
 a2 = a4;  
 y2 = y4;  
 }  
 else  
 {  
 a1 = a3;  
 y1 = y3;  
 }  
 }  
 return \_func({a1});  
}

### 黄金分割法

黄金分割法的类型声明如下，其类型声明以与 Fibonacci 法一致。

class GoldenSelection : public Optimizer1D  
{  
 const double \_lb;  
 const double \_ub;  
 const size\_t \_iter;  
  
 public:  
 GoldenSelection(ObjFunc f,   
 double lb,  
 double ub,  
 size\_t iter = 16) noexcept;  
 Solution optimize() noexcept;  
 ~GoldenSelection() {}  
};

黄金分割法的优化算法实现如下，它的思路与 Fibonacci 法一致，不同的是它使用黄金分割数 0.618 作为固定的区间收缩比例。

Solution GoldenSelection::optimize() noexcept  
{  
 // 1-D function  
 // function shoulde be convex function  
 double a1 = \_lb;  
 double a2 = \_ub;  
 if (a1 > a2)  
 {  
 cerr << "Range error" << endl;  
 exit(EXIT\_FAILURE);  
 }  
  
 const double rate = (sqrt(5) - 1) / 2;  
 double y1, y2;  
 for (size\_t i = \_iter - 1; i > 0; --i)  
 {  
 const double interv\_len = a2 - a1;  
 const double a3 = a2 - rate \* interv\_len;  
 const double a4 = a1 + rate \* interv\_len;  
 if (a3 == a4)  
 break;  
 else  
 {  
 assert(a3 < a4);  
 const double y3 = \_func({a3}).fom();  
 const double y4 = \_func({a4}).fom();  
 if (y3 < y4)  
 {  
 a2 = a4;  
 y2 = y4;  
 }  
 else  
 {  
 a1 = a3;  
 y1 = y3;  
 }  
 }  
 }  
 return y1 < y2 ? \_func({a1}) : \_func({a2});  
}

### 外推法

黄金分割法与 Fibonacci 法都需要事先知道最优点的范围，而外推法则可以适用于最优点范围不知道的情况，它先寻找一个最优点的范围，然后再去调用其他优化算法，比如黄金分隔法或二次插值法在找到的范围内进行优化。

下面是外推法的类声明以及算法实现：

class Extrapolation : public Optimizer1D  
{  
 const Paras \_init;  
 const double \_min\_len; // min extrapolation step  
 const double \_max\_len; // max extrapolation step  
 public:  
 Extrapolation(ObjFunc f,  
 Paras i,  
 double min\_len,  
 double max\_len  
 ) noexcept;  
 Solution optimize() noexcept;  
 ~Extrapolation() {}  
};  
  
Solution Extrapolation::optimize() noexcept  
{  
 // 1-D function  
 double step = \_min\_len;  
 double x1 = \_init[0];  
 double x2 = x1 + step;  
 double y1 = \_func({x1}).fom();  
 double y2 = \_func({x2}).fom();  
  
 double lb = x1;  
 double ub = x1 + \_max\_len;  
 if (y2 > y1)  
 {  
 step \*= -1;  
 ub = x1 - \_min\_len;  
 lb = x1 - \_max\_len;  
 x2 = x1 + step;  
 y2 = \_func({x2}).fom();  
 if (y2 > y1) return \_func({x1});  
 }  
 double factor = 2;  
 double x3 = x2 + factor \* step;  
 double y3 = \_func({x3}).fom();  
 double xa, xc;  
 double ya, yc;  
 if (y3 > y2)  
 {  
 xa = x1;  
 xc = x3;  
 ya = y1;  
 yc = y3;  
 }  
 else  
 {  
 while (y3 < y2 && (lb < x3 && x3 < ub))  
 {  
 factor \*= 2;  
 x3 += factor \* step;  
 if (x3 >= ub) x3 = ub;  
 if (x3 <= lb) x3 = lb;  
 y3 = \_func({x3}).fom();  
 }  
 double xtmp1 = x3 - factor \* step;  
 double xtmp2 = x3 - (factor / 2) \* step;  
 double ytmp1 = \_func({xtmp1}).fom();  
 double ytmp2 = \_func({xtmp2}).fom();  
 if (ytmp1 < ytmp2)  
 {  
 xa = x2;  
 xc = xtmp2;  
 ya = y2;  
 yc = ytmp2;  
 }  
 else  
 {  
 xa = xtmp1;  
 xc = x3;  
 ya = ytmp1;  
 yc = y3;  
 }  
 }  
  
 if (xa > xc)  
 {  
 std::swap(xa, xc);  
 std::swap(ya, yc);  
 }  
 const double len = xc - xa;  
 const size\_t gso\_iter = 2 + (log10(\_min\_len / len) / log10(0.618));  
 return GoldenSelection(\_func, xa, xc, gso\_iter).optimize();  
}

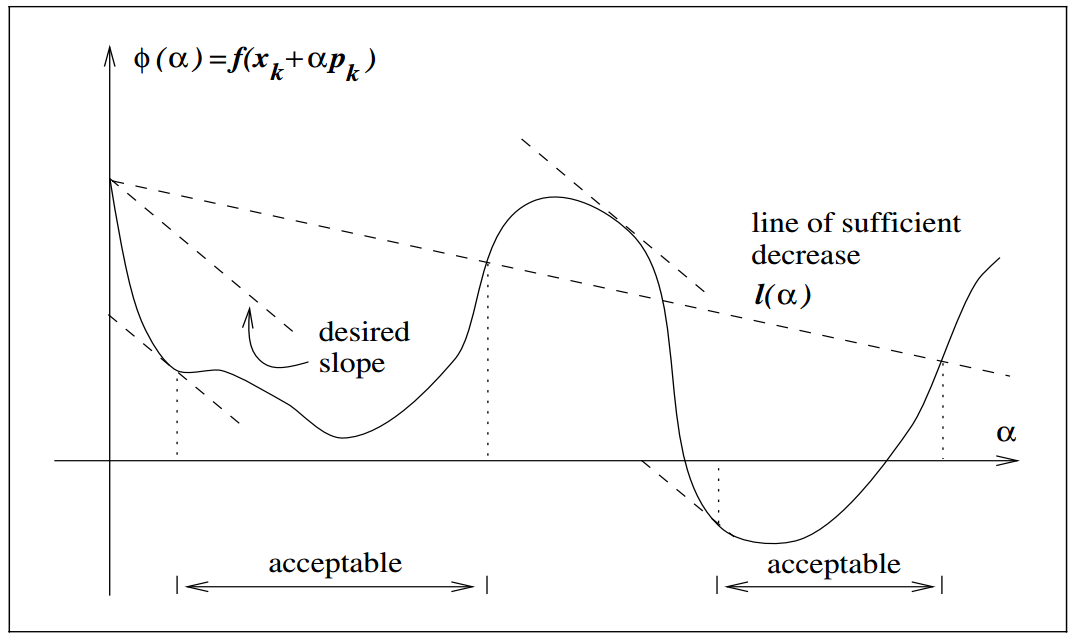
## 不精确线搜索

上一节实现的一维优化算法，都是期望找到在搜索方向上的最优点。但是，很多时候，找到严格意义上的最优点，往往需要很多次迭代; 而且, 因为搜索方向上的最优点并不是多元函数的最优点，在多元函数优化过程中，找到搜索方向上的最优点也没有必要。只要保证步长使得函数在搜索方向上下降足够多就可以了。因此，在实际的多元函数优化中，当需要确定在搜索方向上的步长时，常常并不采用精确的一元函数优化算法，而是规定一个“在搜索方向上足够下降”的标准，然后只要找到满足这样标准的点即可。

[**Strong wolfe condition**](https://www.wikiwand.com/en/Wolfe_conditions) 是一个常用的不精确线搜索的判据，其判据如下：

在上式中， 与 满足 ，其中，第一个不等式被称作 **sufficient decrease condition**，第二个不等式被称作 **curvature condition**。如果步长满足 sufficient decrease condition，则说明在步长处，函数已经有了足够的下降，而 curvature condition 则是要求函数在搜索方向上的梯度也有足够大的下降，因为很显然，如果在步长处函数的梯度仍然很大，则说明在这个方向上仍有进一步改变步长的余地。

Figure 1 是 strong wolfe condition 的一个例子，对于图中一维函数，只要最终步长选在位于"acceptable"的区间内即可。



Example of wolfe condition

本次 project 实现了寻找满足 strong wolfe condition 的搜索步长的算法，其基本思路是，先通过插值与外推的方法，尝试一系列递增的 trial step，找到一个满足 strong wolfe condition 的区间。再在这个区间内，进行二次或三次插值，直到找到满蓄 strong wolfe condition 的步长。

Strong wolfe condition 不精确线搜索算法代码，可以去 src/Optimizer/StrongWolfe.cpp 中查看。

## 多维函数优化

首先定义了两个基类，MultiDimOptimizer 与 GradientMethod，其中，MultiDimOptimizer 是一切多元函数优化算法的基类，在其中定义了一些辅助性质的成员变量与函数，如函数维度、最大迭代次数，最大与最小步长等。

GradientMethod 是MultiDimOptimizer 的一个派生类，它是所有基于梯度法的算法的基类，包括梯度下降法、共轭梯度法、牛顿法和拟牛顿法。

在 GradientMethod 中定义了一些与梯度有关的变量与函数，如求梯度的GradientMethod::get\_gradient，求 Hessian 矩阵的 GradientMethod::hessian，这两个函数都是虚函数，可以被派生类重载。MultiDimOptimizer 中还定义了一些与梯度相关的成员变量，如用数值法求梯度时用到的 \_epsilon。以及判定收敛（梯度为零）的最小梯度等。

class MultiDimOptimizer  
{  
protected:  
 const size\_t \_dim;  
 const size\_t \_max\_iter;  
 const double \_min\_walk;  
 const double \_max\_walk;  
 const std::string \_func\_name;  
 const std::string \_algo\_name;  
 std::ofstream \_log;  
  
 virtual Solution run\_func(const Paras&) noexcept;  
 virtual Solution run\_line\_search(  
 const Solution& s,  
 const Eigen::VectorXd& direction  
 ) noexcept;  
  
private:  
 ObjFunc \_func;  
 StrongWolfe \_line\_searcher;  
 size\_t \_eval\_counter;  
 size\_t \_linesearch\_counter;  
  
public:  
 void clear\_counter() noexcept   
 {   
 \_eval\_counter = 0;  
 \_linesearch\_counter = 0;  
 }  
 size\_t eval\_counter() noexcept { return \_eval\_counter; }  
 size\_t linesearch\_counter() noexcept   
 {   
 return \_linesearch\_counter;   
 }  
 MultiDimOptimizer(  
 ObjFunc f,  
 size\_t d,  
 size\_t max\_iter,  
 double min\_walk,  
 double max\_walk,  
 std::string func\_name,  
 std::string algo\_name) noexcept;  
 virtual ~MultiDimOptimizer(){}  
};  
class GradientMethod : public MultiDimOptimizer  
{  
protected:  
 const Paras \_init;  
 const double \_epsilon; // use \_epsilon to calc gradient  
 const double \_zero\_grad; // threshold for zero gradient   
  
 virtual Eigen::VectorXd get\_gradient(const Solution& s) noexcept;  
 virtual Eigen::MatrixXd hessian(  
 const Solution& point,  
 const Eigen::VectorXd& grad  
 ) noexcept;  
  
public:  
 GradientMethod(  
 ObjFunc f,  
 size\_t d,  
 Paras i,  
 double epsi,  
 double zgrad,  
 double minwalk,  
 double maxwalk,  
 size\_t max\_iter,  
 std::string fname,  
 std::string aname) noexcept;  
 virtual ~GradientMethod() { if(\_log.is\_open()) \_log.close(); }   
};

### 梯度下降法

梯度下降法假定函数在搜索域内总是一阶可导，对一个函数 ，给定一个初始点，梯度，则搜索方向，当梯度为零时判定收敛，此时，找到了函数在这个区域的极小值。

梯度下降法算法描述如下：

1. 对初始点 ，求出其梯度 ，若 ，或者达到 ，则判定收敛。
2. 搜索方向
3. 在搜索方向上做一维搜索，找出最优步长
4. ，置 ，转 step 1。

梯度下降法实现代码如下：

Solution GradientDescent::optimize() noexcept  
{  
 clear\_counter();  
 \_log << \_func\_name << endl;  
  
 Solution sol = run\_func(\_init);  
 VectorXd grad = get\_gradient(sol);  
 double grad\_norm = grad.lpNorm<2>();  
 double len\_walk = numeric\_limits<double>::infinity();  
 while (grad\_norm > \_zero\_grad   
 && eval\_counter() < \_max\_iter   
 && len\_walk > \_min\_walk)  
 {  
 // LOG is a macro used to record evaluated function input  
 LOG(sol, grad);  
 const Solution new\_sol = run\_line\_search(sol, -1 \* grad);  
 len\_walk = vec\_norm(new\_sol.solution() - sol.solution());  
 sol = new\_sol;  
 grad = get\_gradient(sol);  
 grad\_norm = grad.lpNorm<2>();  
 }  
 \_log << "=======================================" << endl;  
 write\_log(sol, grad);  
 \_log << "len\_walk: " << len\_walk << endl;  
 \_log << "eval: " << eval\_counter() << endl;  
 \_log << "line search: " << linesearch\_counter() << endl;  
 if (eval\_counter() >= \_max\_iter)   
 \_log << "max iter reached" << endl;  
 return sol;  
}

### 共轭梯度法

当目标函数在极值点附近的条件数（即 Hessian 矩阵最大特征值与最小特征值之比）过大时，梯度下降法在极值点附近会出现来回折叠现象，导致收敛较慢。共轭梯度法（Conjugate Gradient Method）可以克服这种问题，它选择共轭梯度方向作为搜索方向。

可以证明，如果目标函数在极值点附近是二次的，对于 N 维函数，则只需要 N 次一维查找，就可以找到极值点。当然上面的一维查找指的是精确的一维查找。如果使用不精确一维查找或者问题的阶数高于二阶，N 维问题需要的查找次数会大于N。

对于一个 N 维矩阵 ，如果向量 ，，满足 ，则这两个向量对于矩阵 共轭。N维空间中，共有N个互相共轭的向量。共轭梯度法第一步以梯度方向为搜索方向，而后每一步的搜索方向都与之前的搜索方向互相共轭，如此搜索 N 步。如果 N 步之后，仍然没有找到极值点。则再以梯度方向为搜索方向，再搜索 N 步。如此循环，直至找到极值点。

共轭梯度法算法描述如下:

1. ， 为初始点，计算梯度，若 或者达到最大迭代次数，则算法终止。否则，选择搜索方向
2. 搜索方向上做一维搜索，找到最优步长
   * 。
3. 计算 点的梯度 ，计算:
   * 。
4. ，若，则转step 2，否则，若 ，则算法终止，否则转 step 1。

共轭梯度法的实现代码如下:

Solution ConjugateGradient::optimize() noexcept  
{  
 clear\_counter();  
 \_log << \_func\_name << endl;  
  
 Solution sol = run\_func(\_init);  
 VectorXd grad = get\_gradient(sol);  
 VectorXd conj\_grad = grad;  
 double grad\_norm = grad.lpNorm<2>();  
 double len\_walk = numeric\_limits<double>::infinity();  
 assert(sol.solution().size() == \_dim);  
 while (grad\_norm > \_zero\_grad &&   
 eval\_counter() < \_max\_iter &&   
 len\_walk > \_min\_walk)  
 {  
 conj\_grad = grad;  
 for (size\_t i = 0; i < \_dim; ++i)  
 {  
 LOG(sol, grad, conj\_grad);  
 const Solution new\_sol = run\_line\_search(  
 sol,  
 -1 \* conj\_grad);  
 VectorXd new\_grad = get\_gradient(new\_sol);  
 double beta = pow(new\_grad.lpNorm<2>()/grad.lpNorm<2>(),2);  
  
 len\_walk = vec\_norm(new\_sol.solution() - sol.solution());  
 sol = new\_sol;  
 conj\_grad = new\_grad + beta \* conj\_grad;  
 grad = new\_grad;  
 grad\_norm = grad.lpNorm<2>();  
 if (!(grad\_norm > \_zero\_grad)) break;  
 }  
 }  
 \_log << "=======================================" << endl;  
 write\_log(sol, grad, conj\_grad);  
 \_log << "len\_walk: " << len\_walk << endl;  
 \_log << "eval: " << eval\_counter() << endl;  
 \_log << "line search: " << linesearch\_counter() << endl;  
 if (eval\_counter() >= \_max\_iter)   
 \_log << "max iter reached" << endl;  
 return sol;  
}

### 牛顿法

梯度下降法与共轭梯度法都是利用函数的梯度，而牛顿法利用函数的二阶导（Hessian 矩阵），因而能够达到比梯度下降法与共轭梯度法更快的收敛速度。

牛顿法算法描述如下：

1. 对于初始点 ，求出梯度 ，若 ，则判定收敛，算法终止。
2. 求出 Hessian 矩阵 ，计算搜索方向 。
3. 在搜索方向上做一维搜索，计算最优步长
4. ，转 step 1。

牛顿法的实现代码如下:

Solution Newton::optimize() noexcept  
{  
 clear\_counter();  
 \_log << "func: " << \_func\_name << endl;  
 Solution sol = run\_func(\_init);  
 VectorXd grad = get\_gradient(sol);  
 MatrixXd hess = hessian(sol, grad);  
 double grad\_norm = grad.lpNorm<2>();  
 double len\_walk = numeric\_limits<double>::infinity();  
 while (grad\_norm > \_zero\_grad &&   
 eval\_counter() < \_max\_iter &&   
 len\_walk > \_min\_walk)  
 {  
 VectorXd direction = -1\*hess.colPivHouseholderQr().solve(grad);  
 double judge = grad.transpose() \* direction;  
 double dir = judge < 0 ? 1 : -1;  
 LOG(sol, grad, hess);  
 direction \*= dir;  
 Solution new\_sol = run\_line\_search(sol, direction);  
 len\_walk = vec\_norm(new\_sol.solution() - sol.solution());  
 sol = new\_sol;  
 grad = get\_gradient(sol);  
 hess = hessian(sol, grad);  
 grad\_norm = grad.lpNorm<2>();  
 }  
 \_log << "=======================================" << endl;  
 write\_log(sol, grad, hess);  
 \_log << "len\_walk: " << len\_walk << endl;  
 \_log << "iter: " << eval\_counter() << endl;  
 \_log << "line search: " << linesearch\_counter() << endl;  
 \_log << "eigenvalues of hess: " << endl   
 \_log << hess.eigenvalues() << endl;  
 if (eval\_counter() >= \_max\_iter)   
 \_log << "max iter reached" << endl;  
 return sol;  
}

### 拟牛顿法：BFGS法与DFP法

Newton 法是二阶收敛，因此在理论上会比梯度法更快。但是如果目标函数无法直接给出 Hessian 矩阵，则要用有限差分的方法近似 Hessian 矩阵。对于N维的问题，其复杂度为 ，当目标函数的维度上升时，求 Hessian 矩阵的代价就会变得不可接受。拟牛顿法（Quasi-Newton Method）通过迭代的方法，来近似 Hessian 矩阵。常见的拟牛顿法有 DFP 法与 BFGS 法。

在牛顿法迭代过程中，Hessian 矩阵满足如下关系:

记 ，，则:

或者:

上面两式称作**拟牛顿条件**。如果 是正定的，则搜索方向 是一个下降方向。

选定初始的正定矩阵 ，对于空间中点 ，其梯度

对 DFP 法，G 矩阵如此确定

对BFGS

搜索方向

BFGS法的实现代码如下:

Solution BFGS::optimize() noexcept  
{  
 clear\_counter();  
 \_log << "func: " << \_func\_name << endl;  
  
 Solution sol = run\_func(\_init);  
 VectorXd grad = get\_gradient(sol);  
 MatrixXd quasi\_hess = MatrixXd::Identity(\_dim, \_dim);  
 double grad\_norm = grad.lpNorm<2>();  
 double len\_walk = numeric\_limits<double>::infinity();  
  
 while (grad\_norm > \_zero\_grad && eval\_counter() < \_max\_iter &&  
 len\_walk > \_min\_walk)  
 {  
 LOG(sol, grad, quasi\_hess);  
 const VectorXd direction =  
 -1 \* (quasi\_hess.colPivHouseholderQr().solve(grad));  
 const Solution new\_sol = run\_line\_search(sol, direction);  
 const VectorXd new\_grad = get\_gradient(new\_sol);  
 const vector<double> delta\_x =  
 new\_sol.solution() - sol.solution();  
 const VectorXd ev\_dg = new\_grad - grad;  
 const Map<const VectorXd> ev\_dx(&delta\_x[0], \_dim, 1);  
 len\_walk = vec\_norm(delta\_x);  
 if (len\_walk > 0)  
 {  
 quasi\_hess +=  
 (ev\_dg \* ev\_dg.transpose()) /  
 (ev\_dg.transpose() \* ev\_dx) -  
 (quasi\_hess \* ev\_dx \* ev\_dx.transpose() \*  
 quasi\_hess) /  
 (ev\_dx.transpose() \* quasi\_hess \* ev\_dx);  
  
 sol = new\_sol;  
 grad = new\_grad;  
 grad\_norm = grad.lpNorm<2>();  
 }  
 }  
 \_log << "=======================================" << endl;  
 write\_log(sol, grad, quasi\_hess);  
 \_log << "len\_walk: " << len\_walk << endl;  
 \_log << "eval: " << eval\_counter() << endl;  
 \_log << "line search: " << linesearch\_counter() << endl;  
  
 if (eval\_counter() >= \_max\_iter)  
 \_log << "max iter reached" << endl;  
 return sol;  
}

DFP法的实现代码如下：

Solution DFP::optimize() noexcept  
{  
 clear\_counter();  
 \_log << "func: " << \_func\_name << endl;  
  
 Solution sol = run\_func(\_init);  
 VectorXd grad = get\_gradient(sol);  
 double grad\_norm = grad.lpNorm<2>();  
 double len\_walk = numeric\_limits<double>::infinity();  
 MatrixXd quasi\_hess\_inverse = MatrixXd::Identity(\_dim, \_dim);  
  
 while (grad\_norm > \_zero\_grad && eval\_counter() < \_max\_iter &&  
 len\_walk > \_min\_walk)  
 {  
 LOG(sol, grad, quasi\_hess\_inverse);  
 VectorXd dvec = -1 \* (quasi\_hess\_inverse \* grad);  
#ifdef WRITE\_LOG  
 const double judge = grad.transpose() \* dvec;  
 \_log << "judge: " << judge << endl;  
 if (judge > 0) \_log << "judge greater than zero" << endl;  
#endif  
 const Solution new\_sol = run\_line\_search(sol, dvec);  
 const VectorXd new\_grad = get\_gradient(new\_sol);  
 const vector<double> delta\_x =  
 new\_sol.solution() - sol.solution();  
 const VectorXd ev\_dg = new\_grad - grad;  
 len\_walk = vec\_norm(delta\_x);  
 const Map<const VectorXd> ev\_dx(&delta\_x[0], \_dim, 1);  
 if (len\_walk > 0)  
 {  
 quasi\_hess\_inverse +=  
 (ev\_dx \* ev\_dx.transpose()) /  
 (ev\_dx.transpose() \* ev\_dg) -  
 (quasi\_hess\_inverse \* ev\_dg \* ev\_dg.transpose() \*  
 quasi\_hess\_inverse) /  
 (ev\_dg.transpose() \* quasi\_hess\_inverse \* ev\_dg);  
  
 sol = new\_sol;  
 grad = new\_grad;  
 grad\_norm = grad.lpNorm<2>();  
 }  
 }  
 \_log << "=======================================" << endl;  
 write\_log(sol, grad, quasi\_hess\_inverse);  
 \_log << "len\_walk: " << len\_walk << endl;  
 \_log << "eval: " << eval\_counter() << endl;  
 \_log << "line search: " << linesearch\_counter() << endl;  
 if (eval\_counter() >= \_max\_iter)  
 \_log << "max iter reached" << endl;  
 return sol;  
}

### 单纯形法

算法参数:

单纯形法算法描述如下:

1. 初始化，选取 个初始点，并对其求值，组成集合
2. 若达到最大迭代次数，或者达到收敛条件，算法终止。
3. 中的结果进行排序，选出最差结果 ，第二差的结果 ，以及最好的结果 。
4. 算 中，除了 以外的其他所有点的中点
5. 算反射点 。
6. 若 ，则用 在 中更新 ，并转 step 2
7. 若 ，则计算 ，并用 与 中较小的一组解更新 ，并转 step 2
8. 计算 。
9. 若 ，则用 更新 ，并转 step 2
10. 否则，更新所有点，将所有点向 靠拢，对 中的任意点 ，更新 。

单纯形法实现的代码如下：

double NelderMead::update\_sols(size\_t idx,  
 const Solution& new\_sol) noexcept  
{  
 assert(\_sols.size() == \_dim + 1);  
 assert(idx <= \_dim);  
 const double walk\_len =  
 vec\_norm(new\_sol.solution() - \_sols[idx].solution());  
 \_sols[idx] = new\_sol;  
 return walk\_len;  
}  
Solution NelderMead::optimize() noexcept  
{  
 clear\_counter();  
 \_log << \_func\_name << endl;  
 \_sols.clear();  
 \_sols.reserve(\_dim + 1);  
 for (size\_t i = 0; i < \_dim + 1; ++i)  
 \_sols.push\_back(run\_func(\_inits[i]));  
 double walk\_len = numeric\_limits<double>::infinity();  
 while (eval\_counter() < \_max\_iter && walk\_len > \_min\_walk)  
 {  
 // 1. order  
 std::sort(\_sols.begin(), \_sols.end(), std::less<Solution>());  
 const Solution& worst = \_sols[\_dim];  
 const Solution& sec\_worst = \_sols[\_dim - 1];  
 const Solution& best = \_sols[0];  
  
 // 2. centroid calc  
 Paras centroid(\_dim, 0);  
 for (size\_t i = 0; i < \_dim; ++i)  
 centroid = centroid + \_sols[i].solution();  
 centroid = 1.0 / static\_cast<double>(\_dim) \* centroid;  
  
 // 3. reflection  
 Solution reflect = run\_func(  
 centroid + \_alpha \* (centroid - worst.solution()));  
 LOG(reflect);  
 if (best <= reflect && reflect < sec\_worst)  
 {  
 walk\_len = update\_sols(\_dim, reflect);  
 }  
 else if (reflect < best) // 4. expansion  
 {  
 Solution expanded = run\_func(  
 centroid + \_gamma \* (reflect.solution() - centroid));  
 LOG(expanded);  
 const Solution& new\_sol =  
 expanded < reflect ? expanded : reflect;  
 walk\_len = update\_sols(\_dim, new\_sol);  
 }  
 else // 5. contract  
 {  
 assert(!(reflect < sec\_worst));  
 Solution contracted = run\_func(  
 centroid + \_rho \* (worst.solution() - centroid));  
 LOG(contracted);  
 if (contracted < worst)  
 {  
 walk\_len = update\_sols(\_dim, contracted);  
 }  
 else // 6. shrink  
 {  
#ifdef WRITE\_LOG  
 \_log << "shrink: " << endl;  
#endif  
 walk\_len = 0;  
 for (size\_t i = 1; i < \_dim + 1; ++i)  
 {  
 Paras p = best.solution() -  
 \_sigma \* (\_sols[i].solution() -  
 best.solution());  
 double tmp\_walk = update\_sols(i, run\_func(p));  
 walk\_len = max(tmp\_walk, walk\_len);  
 LOG(\_sols[i]);  
 }  
 }  
 }  
 }  
 std::sort(\_sols.begin(), \_sols.end(), std::less<Solution>());  
 \_log << "=========================================" << endl;  
 write\_log(\_sols[0]);  
 return \_sols[0];  
}

### 鲍威尔法

鲍威尔法的实现代码如下：

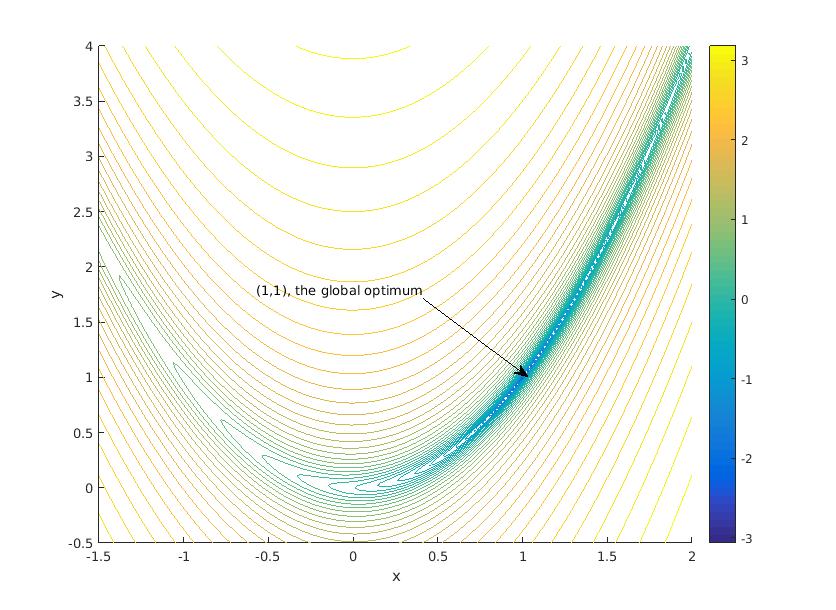
Solution Powell::optimize() noexcept  
{  
 clear\_counter();  
 Solution sol = run\_func(\_init);  
 double walk\_len = numeric\_limits<double>::infinity();  
  
 // initial search directions are axes  
 vector<VectorXd> search\_direction(\_dim, VectorXd(\_dim));  
 for (size\_t i = 0; i < \_dim; ++i)  
 {  
 for (size\_t j = 0; j < \_dim; ++j)  
 search\_direction[i][j] = i == j ? 1.0 : 0.0;  
 }  
 while (eval\_counter() < \_max\_iter && walk\_len > \_min\_walk)  
 {  
 double max\_delta\_y = -1 \* numeric\_limits<double>::infinity();  
 size\_t max\_delta\_id;  
 Paras backup\_point = sol.solution();  
 for (size\_t i = 0; i < \_dim; ++i)  
 {  
 LOG(sol);  
 Solution search\_sol =  
 run\_line\_search(sol, search\_direction[i]);  
 if (sol.fom() - search\_sol.fom() > max\_delta\_y)  
 {  
 max\_delta\_y = sol.fom() - search\_sol.fom();  
 max\_delta\_id = i;  
 }  
 sol = search\_sol;  
 }  
 Paras new\_direc = sol.solution() - backup\_point;  
 VectorXd new\_direc\_vxd = Map<VectorXd>(&new\_direc[0], \_dim);  
 walk\_len = new\_direc\_vxd.lpNorm<2>();  
 search\_direction[max\_delta\_id] = new\_direc\_vxd;  
 }  
 \_log << endl  
 << "==========================================" << endl;  
 write\_log(sol);  
 return sol;  
}

$\pagebreak$

## Benchmark

使用 Rosenbrock 函数来比较不同优化算法的性能。Rosenbrock 函数如下定义：

Figure 2 为 Rosenbrock 函数的等高线图，为增强显示效果，图中函数值对 10 取对数，即



Contour of

将 设为初始点 （单纯形法除外，单纯形法初始需要 N + 1 个点，N 为维度），程序运行结果以及图示见下表以及下图。其中 NumLineSearch 表示程序做一维搜索的次数，即为图上点的数量。而对于梯度法以及鲍威尔法，确定搜索方向后，要找到下一个点，还需要在一维搜索上话费数次函数执行，表中 NumEvaluation 一列表示算法优化过程中实际的函数执行次数。可以看出，梯度下降法表现最差，而单纯形法表现最好。

需要说明的是，表中数据只是不同算法对 Rosenbrock 函数以及 这一个初始点的性能，换一个比较函数，换一个初始点，都可能会有不同的结果。

Compare of different algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Fom | NumLineSearch | NumEvaluation |
| GradientDescent | 1.42e-4 | 639 | 6516 |
| ConjugateGradient | 5.29e-9 | 48 | 694 |
| Newton | 1.01e-5 | 11 | 200 |
| DFP | 2.68e-6 | 25 | 367 |
| BFGS | 3.82e-6 | 29 | 430 |
| Simplex | 2.44e-11 | 0 | 170 |
| Powell | 9.46e-11 | 70 | 739 |

