20211046
Hyeonjang An

Homework 2

EC6301
Numerical Opimization
September 26, 2021

Problem

Discuss thier comparative performance for at least four different problems you generate.

1. Target functions and derivative
   Function1 is general quaratic function, which has derivative as cubic. function2 is log and function3 is trigonometric function. function4 is the minus signed version of gaussian function, which is usally used as kernel. It's σ value is set as 1.4

function	original function	derivation of function	interval
function1	$f(x) = x^4 + 2x^3 - 3x^2 - 10x + 7$	$f'(x) = 4x^3 + 6x^2 - 6x - 10$	[-5, 5]
function2	$f(x) = x \ln(x)$	$f'(x) = \ln(x) + 1$	[0.1, 5]
function3	$f(x) = \sin(x) + x^2 - 10$	$f'(x) = \cos(x) + 2x$	[-5, 5]
function4	$f(x) = -\exp(-\frac{x^2}{\sigma^2})$	$f'(x) = \frac{x}{\sigma^2}\exp(-\frac{x^2}{2\sigma^2})$	[-1, 1]

2. Conditions
   - Within the interval, all functions are continuous and the first order derivative of those are also continuous.
   - In the case of bracketing method (bisection & regular falsi)
     Within the interval the function has the value zero. This can be calculated analytically.
   - In the case of straight line method (Newton's & secant)
     For the comparision pairness, the initial points of each methods are same as the maximum point of interval, which is used in bracketing method.

3. Peformance comparision

	bisection	Newton's	secant	regular falsi
function1	2072ns	546ns	789ns	11241ns
function2	2639ns	189ns	892ns	5778ns
function3	2428ns	376ns	400ns	1310ns
function4	105ns	213ns	36.9ns	238ns

4. Analysis
   Apparently, the convergence of Newton's method is the fastest. Moreover, the overhead of regular falsi method is bigger than I thought. In the case of function1, regular falsi method has the slowest convergence rate.
   The speical thing is function 4. In the case of gaussian function, Newton's method has the slowest. I think that it is because of the calculation overhead from derivation.

Implementation

Implement the method of bisection , Newtons's, secant, regular falsi.

1. Optimizing Method Class

```
#include <limits>
#include <functional>
#include <cassert>
#include <boost/math/constants/constants.hpp>

namespace numerical_optimization
{

using function_t = std::function<float(const float&)>;
using boundary_t = std::pair<float, float>;

constexpr float  MIN = 1e-4;// std::numeric_limits<float>::min();
constexpr float  MAX = std::numeric_limits<float>::max();
constexpr float  GOLDEN_RATIO = 1.f/boost::math::constants::phi<float>();
constexpr size_t FIBONACCI_MAX = 46;

class Method
{
public:
    Method(function_t f):function(f) {
        boundary = seeking_bound(5);
    };

    // assignment 1
    float bisection(float start, float end);
    float newtons(float x);
    float secant(float x1, float x0);
    float regular_falsi(float start, float end);
    float regular_falsi_not_recur(float start, float end);

    // assignment 2
    float fibonacci_search(size_t N=FIBONACCI_MAX);
    float fibonacci_search(float start, float end, size_t N);
    float golden_section(size_t N=FIBONACCI_MAX);
    float golden_section(float start, float end, size_t N);

public: // for debugging, originally protected
    function_t function;
    boundary_t boundary;
    const size_t iter = 10000000; // termination condition

private:
    // for convenience
    bool  near_zero(float x) {
        return x==0 || -MIN<function(x)&&function(x)<MIN;
    }

    // for fibonacci_search
    std::vector<int> construct_fibonacci(size_t N) const;
    std::pair<float, float> seeking_bound(float step_size);
    int random_int() const;
};

};
```

2. Seeking bound

```
        fibonacci[i+2] = fibonacci[i] + fibonacci[i+1];

    return fibonacci;
}

boundary_t Method::seeking_bound(float step_size) {
    boundary_t result;

    std::vector<float> x(iter); x[1] = float(random_int());
    float d = step_size;

    float f0 = function(x[1]-d);
    float f1 = function(x[1]);
    float f2 = function(x[1]+d);

    if (f0>=f1 && f1>=f2) {
        x[0] = x[1] - d;
        x[2] = x[1] + d;
        d = d;
    } else if (f0<=f1 && f1<=f2) {
        x[0] = x[1] + d;
        x[2] = x[1] - d;
        d = -d;
    } else if (f0>=f1 && f1<=f2) {
        result = std::make_pair(x[1]-d, x[1]+d);
    }

    // now default
    function_t increment = [](const float& f){ return std::pow(2, f); };
    for(size_t k=2; k<iter; k++) {
        x[k+1] = x[k] + increment(k) * d;

        if(function(x[k+1])>=function(x[k]) && d>0) {
            result = std::make_pair(x[k-1], x[k+1]);
            break;
        }
        else if(function(x[k+1])>=function(x[k]) && d<0) {
            result = std::make_pair(x[k+1], x[k-1]);
            break;
        }
    }
    return result;
}
// random function for boundary seeking
int Method::random_int() const {
    // threshold
    constexpr int scale = 100000;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(
        std::numeric_limits<int>::min()/scale,
        std::numeric_limits<int>::max()/scale
        );
    return distrib(gen);
}

}
```

3. Fibonacci search

```
        b.first*((float)F[N-2]/(float)F[N]) + b.second*((float)F[N-1]/(float)F[N])
    );

    for(size_t n=N-1; n>1; n--) {

        // unimodality step
        if(function(x.first)>function(x.second)) {
            b.first = x.first;

            // only one calculation needed
            x.first  = x.second;
            x.second = b.first*((float)F[n-2]/(float)F[n]) + b.second*((float)F[n-1]/(flo

        } else if(function(x.first)<function(x.second)) {
            b.second = x.second;

            // only one calculation needed
            x.second = x.first;
            x.first  = b.first*((float)F[n-1]/(float)F[n]) + b.second*((float)F[n-2]/(flo
        }

    }
    return (b.first + b.second)/2;
```

4. Golden section search

```
float Method::golden_section(float start, float end, size_t N) {
    boundary_t b = std::minmax(start, end);
    float length = b.second - b.first;

    boundary_t x = std::make_pair(
        b.second - GOLDEN_RATIO*length,
        b.first  + GOLDEN_RATIO*length
    );

    for(size_t n=N; n>1; n--) {

        // unimodality step
        if(function(x.first)>function(x.second)) {
            b.first = x.first;

            length = b.second - b.first;

            // only one calculation needed
            // x = std::make_pair(x.second, b.first + GOLDEN_RATIO*length);
            x.first  = x.second;
            x.second = b.first + GOLDEN_RATIO * length;

        } else if(function(x.first)<function(x.second)) {
            b.second = x.second;

            length = b.second - b.first;

```