Homework 2

Problem

Discuss comparative study in terms of convergence speed between search algorithms for at least four optimization problems you generated accordingly.

1. Target functions

   - From function1 to function4, they are same as assignment1. function1 is general quaratic function. function2 is log and function3 is trigonometric function. function4 is the minus signed version of gaussian function. function5 and function6 are newly added for testing nondifferentiable cases.

	functions	performance	
		fibonacci search	golden section search
function1	$f(x) = x^4 + 2x^3 - 3x^2 - 10x + 7$	14643ns	14453ns
function2	$f(x) = x\ln(x)$	6123ns	5552ns
function3	$f(x) = \sin(x) + x^2 - 10$	5604ns	5120ns
function4	$f(x) = -\exp(-\frac{x^2}{\sigma^2})$	5569ns	5023ns
function5	$f(x) = \lvert x - 0.3\rvert$	5020ns	4075ns
function6	$f(x) = \lvert\ln(x)\rvert$	5011ns	4496ns

2. Conditions

   - The bound is determined by the seeking bound algorithm. The initial random values to search bound are chosen by random_int function.

3. Analysis

   - The maximum iteration is set by 46, because of the limitation for maximum fibonacci sequence value. The maximum integer value is now 2,147,483,647, but the 47th fibonacci value is 2,971,215,073. If more complicated Implementation is added, the fibonacci sequence could be larger. But currently didn't. Therefore, the maximum iteration is limited as 46, and the performance is related to this.
   -

Homework 2

Implementation

1. Class: Optimizing Method

```
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/1.618033988749895f;
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);

    // for convienience
    boundary_t get_bound() const;
private:
    function_t function;
    boundary_t boundary;
    const size_t iter = 10000000; // termination condition

    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;
    boundary_t seeking_bound(float step_size);
    int random_int() const;
```

2. Seeking bound

   •

```
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 2^x incremental function
    function_t increment = [](const float& f){ return std::pow(2, f); };
    for(size_t k=2; k<iter-1; 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
```

Homework 2

```cpp
    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);
}

boundary_t Method::get_bound() const {
```

3. Fibonacci search

- Construction of Fibonacci
- Due to the maximum integer value is limited by 214748364, the maximum index of fibonacci sequence is currently 46. If in the case of unsigned or long integer, it could be changed

```cpp

std::vector<int> Method::construct_fibonacci(size_t N) const {
    // cannot over 46 the integer range
    N = std::min(N, FIBONACCI_MAX);
    std::vector<int> fibonacci(N);

    fibonacci[0] = 1;
    fibonacci[1] = 1;

    for(size_t i=0; i<N-2; i++)
        fibonacci[i+2] = fibonacci[i] + fibonacci[i+1];

    return fibonacci;
```

- Fibonacci search

```cpp

float Method::fibonacci_search(float start, float end, size_t N) {
    std::vector<int> F = construct_fibonacci(N);

```

```
    // indexing
    N = F.size()-1;
    boundary_t b = std::minmax(start, end);
    float length = b.second - b.first;

    boundary_t x = std::make_pair(
        b.first*((float)F[N-1]/(float)F[N])
        + b.second*((float)F[N-2]/(float)F[N]),
        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 = std::make_pair(
                x.second,
                b.first*((float)F[n-2]/(float)F[n])
                + b.second*((float)F[n-1]/(float)F[n])
                );
        } else if(function(x.first)<function(x.second)) {
            b.second = x.second;

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

    }
    return (b.first + b.second)/2;
}
// combined with seeking bound
float Method::fibonacci_search(size_t N) {
    return fibonacci_search(boundary.first, boundary.second, N);
```

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-1; n>1; n--) {

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

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

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

            // only one calculation needed
            length = b.second - b.first;
            x = std::make_pair(b.second - GOLDEN_RATIO*length, x.first);
        }
    }
    return (b.first + b.second)/2;
}
// combined with seeking bound
float Method::golden_section(size_t N) {
    return golden_section(boundary.first, boundary.second, N);
}
```