https://github.com/hyeonjang/numerical-optimization
20211046
Hyeonjang An

Homework 2

EC6301
Numerical Opimization
September 26, 2021

Problem

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

1. Target functions and bound
   Bounds are computed automatically by seeking bound algorithm.
   From function1 to function4, they are same as assignment1. The function5 and function6 are newly added for testing nondifferentiable cases.

	functions
function1	$f(x) = x^4 + 2x^3 - 3x^2 - 10x + 7$
function2	$f(x) = x\ln(x)$
function3	$f(x) = \sin(x) + x^2 - 10$
function4	$f(x) = -\exp(-\frac{x^2}{\sigma^2})$
function5	$f(x) = \lvert x - 0.3\rvert$
function6	$f(x) = \lvert\ln(x)\rvert$

2. Peformance comparision

	fibonacci search	golden section search
function1	14643ns	14453ns
function2	6123ns	5552ns
function3	5604ns	5120ns
function4	5569ns	5023ns
function5	5020ns	4075ns
function5	5011ns	4496ns

3. Conditions
   As said before, the bound is determined by the seeking bound algorithm.

4. Analysis
   The maximum iteration is set by 46, because of the limitation for maximum fibonacci sequence value.

Implementation

1. Class: Optimizing Method

```
9  using function_t = std::function<float(const float&)>;
10 using boundary_t = std::pair<float, float>;
11
12 constexpr float  MIN = 1e-4;// std::numeric_limits<float>::min();
13 constexpr float  MAX = std::numeric_limits<float>::max();
14 constexpr float  GOLDEN_RATIO = 1.f/boost::math::constants::phi<float>();
15 constexpr size_t FIBONACCI_MAX = 46;
16
17 class Method {
18 public:
19     Method(function_t f):function(f) { boundary = seeking_bound(5); };
20
21     // assignment 1
22     float bisection(float start, float end);
23     float newtons(float x);
24     float secant(float x1, float x0);
25     float regular_falsi(float start, float end);
26     float regular_falsi_not_recur(float start, float end);
27
28     // assignment 2
29     float fibonacci_search(size_t N=FIBONACCI_MAX);
30     float fibonacci_search(float start, float end, size_t N);
31     float golden_section(size_t N=FIBONACCI_MAX);
32     float golden_section(float start, float end, size_t N);
33
34     // for convienience
35     boundary_t get_bound() const;
36 private:
37     function_t function;
38     boundary_t boundary;
39     const size_t iter = 10000000; // termination condition
40
41     bool  near_zero(float x) {
42         return x==0 || -MIN<function(x)&&function(x)<MIN;
43     }
44     // for fibonacci search
45     std::vector<int> construct_fibonacci(size_t N) const;
46     boundary_t seeking_bound(float step_size);
47     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; 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
```

```
242     constexpr int scale = 100000;
243 
244     std::random_device rd;
245     std::mt19937 gen(rd());
246     std::uniform_int_distribution<> distrib(
247         std::numeric_limits<int>::min()/scale,
248         std::numeric_limits<int>::max()/scale
249         );
250     return distrib(gen);
251 }
```

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

```
187 std::vector<int> Method::construct_fibonacci(size_t N) const {
188     // cannot over 46 the integer range
189     N = std::min(N, FIBONACCI_MAX);
190     std::vector<int> fibonacci(N);
191 
192     fibonacci[0] = 1;
193     fibonacci[1] = 1;
194 
195     for(size_t i=0; i<N-2; i++)
196         fibonacci[i+2] = fibonacci[i] + fibonacci[i+1];
197 
198     return fibonacci;
199 }
```

   - Fibonacci search

```
107 float Method::fibonacci_search(float start, float end, size_t N) {
108     std::vector<int> F = construct_fibonacci(N);
109 
110     // indexing
111     N = F.size()-1;
112     boundary_t b = std::minmax(start, end);
```

```
113     float length = b.second - b.first;
114 
115     boundary_t x = std::make_pair(
116         b.first*((float)F[N-1]/(float)F[N])
117         + b.second*((float)F[N-2]/(float)F[N]),
118         b.first*((float)F[N-2]/(float)F[N])
119         + b.second*((float)F[N-1]/(float)F[N])
120     );
121 
122     for(size_t n=N-1; n>1; n--) {
123 
124         // unimodality step
125         if(function(x.first)>function(x.second)) {
126             b.first = x.first;
127 
128             // only one calculation needed
129             x = std::make_pair(
130                 x.second,
131                 b.first*((float)F[n-2]/(float)F[n])
132                 + b.second*((float)F[n-1]/(float)F[n])
133                 );
134         } else if(function(x.first)<function(x.second)) {
135             b.second = x.second;
136 
137             // only one calculation needed
138             x = std::make_pair(
139                 b.first*((float)F[n-1]/(float)F[n])
140                 + b.second*((float)F[n-2]/(float)F[n]),
141                 x.first
142             );
143         }
144 
145     }
146     return (b.first + b.second)/2;
147 }
148 // combined with seeking bound
149 float Method::fibonacci_search(size_t N) {
150     return fibonacci_search(boundary.first, boundary.second, N);
151 }
```

4. Golden section search

```
153 float Method::golden_section(float start, float end, size_t N) {
154     boundary_t b = std::minmax(start, end);
155     float length = b.second - b.first;
156 
157     boundary_t x = std::make_pair(
158         b.second - GOLDEN_RATIO*length,
159         b.first  + GOLDEN_RATIO*length
160     );
161 
162     for(size_t n=N-1; n>1; n--) {
163 
164         // unimodality step
165         if(function(x.first)>function(x.second)) {
166             b.first = x.first;
167 
168             // only one calculation needed
169             length = b.second - b.first;
170             x = std::make_pair(x.second, b.first + GOLDEN_RATIO*length);
171 
172         } else if(function(x.first)<function(x.second)) {
173             b.second = x.second;
174 
175             // only one calculation needed
176             length = b.second - b.first;
177             x = std::make_pair(b.second - GOLDEN_RATIO*length, x.first);
178         }
179     }
180     return (b.first + b.second)/2;
181 }
182 // combined with seeking bound
183 float Method::golden_section(size_t N) {
184     return golden_section(boundary.first, boundary.second, N);
185 }
```