Biostatistics 615/815 Lecture 17: Single dimensional optimization

Hyun Min Kang

November 13th, 2012
The Minimization Problem
Specific Objectives

Finding global minimum

- The lowest possible value of the function
- Very hard problem to solve generally

Finding local minimum

- Smallest value within finite neighborhood
- Relatively easier problem
A quick detour - The root finding problem

- Consider the problem of finding zeros for $f(x)$
- Assume that you know
  - Point $a$ where $f(a)$ is positive
  - Point $b$ where $f(b)$ is negative
  - $f(x)$ is continuous between $a$ and $b$
- How would you proceed to find $x$ such that $f(x) = 0$?
#include <iostream>

class myFunc {  // a typical way to define a function object
public:
    double operator()(double x) const {
        return (x*x-1);
    }
};

int main(int argc, char** argv) {
    myFunc foo;
    std::cout << "foo(0) = " << foo(0) << std::endl;
    std::cout << "foo(2) = " << foo(2) << std::endl;
}
Root Finding with C++

```cpp
// binary-search-like root finding algorithm
double binaryZero(myFunc foo, double lo, double hi, double e) {
    for (int i=0; ++i) {
        double d = hi - lo;
        double point = lo + d * 0.5;  // find midpoint between lo and hi
        double fpoint = foo(point);  // evaluate the value of the function
        if (fpoint < 0.0) {
            d = lo - point;  lo = point;
        }
        else {
            d = point - hi;  hi = point;
        }
        // e is tolerance level (higher e makes it faster but less accurate)
        if (fabs(d) < e || fpoint == 0.0) {
            std::cout << "Iteration " << i << ", point = " << point
                        << ", d = " << d << std::endl;
            return point;
        }
    }
}
```
Improvements to Root Finding

Approximation using linear interpolation

\[ f^*(x) = f(a) + (x - a) \frac{f(b) - f(a)}{b - a} \]

Root Finding Strategy

- Select a new trial point such that \( f^*(x) = 0 \)
Root Finding Using Linear Interpolation

double linearZero (myFunc foo, double lo, double hi, double e) {
    double flo = foo(lo);   // evaluate the function at the end points
    double fhi = foo(hi);
    for(int i=0;;++i) {
        double d = hi - lo;
        double point = lo + d * flo / (flo - fhi);   // use linear interpolation
        double fpoint = foo(point);
        if (fpoint < 0.0) {
            d = lo - point;
            lo = point;
            flo = fpoint;
        } else {
            d = point - hi;
            hi = point;
            fhi = fpoint;
        }
        if (fabs(d) < e || fpoint == 0.0) {
            std::cout << "Iteration " << i << ", point = " << point << ", d = " << d << std::endl;
            return point;
        }
    }
}
Performance Comparison

**Finding \( \sin(x) = 0 \) between \(-\pi/4\) and \(\pi/2\)**

```cpp
#include <cmath>
class myFunc {
public:
    double operator()(double x) const { return sin(x); }
};
...
int main(int argc, char** argv) {
    myFunc foo;
    binaryZero(foo,0-M_PI/4,M_PI/2,1e-5);
    linearZero(foo,0-M_PI/4,M_PI/2,1e-5);
    return 0;
}
```

**Experimental results**

- `binaryZero()`: Iteration 17, point = \(-2.99606e-06\), d = \(-8.98817e-06\)
- `linearZero()`: Iteration 5, point = 0, d = \(-4.47489e-18\)
R example of root finding

```r
# use uniroot() function for root finding
> uniroot( sin, c(0-pi/4,pi/2) )  ## function and interval as arguments
$root
[1] -3.531885e-09

$f.root
[1] -3.531885e-09

$iter
[1] 4

$estim.prec
[1] 8.719466e-05
```
Summary on root finding

- Implemented two methods for root finding
  - Bisection Method: `binaryZero()`
  - False Position Method: `linearZero()`

- In the bisection method, the bracketing interval is halved at each step
- For well-behaved function, the False Position Method will converge faster, but there is no performance guarantee.
Back to the Minimization Problem

- Consider a complex function $f(x)$ (e.g. likelihood)
- Find $x$ which $f(x)$ is maximum or minimum value
- Maximization and minimization are equivalent
  - Replace $f(x)$ with $-f(x)$
Notes from Root Finding

- Two approaches possibly applicable to minimization problems
  - Bracketing
    - Keep track of intervals containing solution
  - Accuracy
    - Recognize that solution has limited precision
Notes on Accuracy - Consider the Machine Precision

- When estimating minima and bracketing intervals, floating point accuracy must be considered.
- In general, if the machine precision is $\epsilon$, the achievable accuracy is no more than $\sqrt{\epsilon}$.
- $\sqrt{\epsilon}$ comes from the second-order Taylor approximation

$$f(x) \approx f(b) + \frac{1}{2} f''(b)(x - b)^2$$

- For functions where higher order terms are important, accuracy could be even lower.
  - For example, the minimum for $f(x) = 1 + x^4$ is only estimated to about $\epsilon^{1/4}$. 
Outline of Minimization Strategy

1. Find 3 points such that
   - $a < b < c$
   - $f(b) < f(a)$ and $f(b) < f(c)$

2. Then search for minimum by
   - Selecting trial point in the interval
   - Keep minimum and flanking points
Part I: Finding a Bracketing Interval

- Consider two points
  - x-values $a, b$
  - y-values $f(a) > f(b)$
Root Finding

Minimization

Parabola

Boost

Summary

Bracketing in C++

#define SCALE 1.618

void bracket( myFunc foo, double& a, double& b, double& c) {
    double fa = foo(a);
    double fb = foo(b);
    double fc = foo(c = b + SCALE*(b-a) );
    while( fb > fc ) {
        a = b; fa = fb;
        b = c; fb = fc;
        c = b + SCALE * (b-a);
        fc = foo(c);
    }
    // after the loop, fb < fa and fb < fc will hold.
}
Part II: Finding Minimum After Bracketing

- Given 3 points such that
  - \( a < b < c \)
  - \( f(b) < f(a) \) and \( f(b) < f(c) \)
- How do we select new trial point?
What is the best location for a new point $X$?
What we want

We want to minimize the size of next search interval, which will be either from $A$ to $X$ or from $B$ to $C$

- If $f(X) < f(B)$, the next search interval will be $(B, C)$
- If $f(X) > f(B)$, the next search interval will be $(A, X)$
Minimizing worst case possibility

- **Formulae**

\[ w = \frac{b - a}{c - a} \]
\[ z = \frac{x - b}{c - a} \]

Segments will have length either \(1 - w\) or \(w + z\).

- **Optimal case**

\[
\begin{cases}
1 - w = w + z \\
\frac{z}{1-w} = w
\end{cases}
\]

- **Solve It**

\[ w = \frac{3 - \sqrt{5}}{2} = 0.38197 \]
The Golden Search
The Golden Ratio

Bracketing Triplet

A B C
The Golden Ratio

The number 0.38196 is related to the *golden mean* studied by Pythagoras.
The Golden Ratio

New Bracketing Triplet

Alternative New Bracketing Triplet

0.38196
Golden Search

- Reduces bracketing by ~40% after function evaluation
- Performance is independent of the function that is being minimized
- In many cases, better schemes are available
Golden Step

```c
#define GOLD 0.38196
#define ZEPS 1e-10       // precision tolerance
double goldenStep (double a, double b, double c) {
    double mid = ( a + c ) * .5;
    if ( b > mid )
        return GOLD * (a-b);
    else
        return GOLD * (c-b);
}
```
**Golden Search**

```cpp
double goldenSearch(myFunc foo, double a, double b, double c, double e) {
    int i = 0;
    double fb = foo(b);
    while (fabs(c-a) > fabs(b*e)) {
        double x = b + goldenStep(a, b, c);
        double fx = foo(x);
        if (fx < fb) {
            (x > b) ? (a = b) : (c = b);
            b = x; fb = fx;
        } else {
            (x < b) ? (a = x) : (c = x);
        }
        ++i;
    }
    std::cout << "i = " << i << " , b = " << b << " , f(b) = " << foo(b) << std::endl;
    return b;
}
```

Hyun Min Kang

Biostatistics 615/815 - Lecture 17

November 13th, 2012 28 / 49
A running example

Finding minimum of $f(x) = -\cos(x)$

```cpp
class myFunc {
public:
    double operator()(double x) const {
        return 0-cos(x);
    }
};

int main(int argc, char** argv) {
    myFunc foo;
    goldenSearch(foo,0-M_PI/4,M_PI/4,M_PI/2,1e-5);
    return 0;
}
```

Results

$i = 66, \ b = -4.42163e-09, \ f(b) = -1$
R example of minimization

```r
> optimize(cos, interval = c(0 - pi/4, pi/2), maximum = TRUE)

$maximum
[1] -8.648147e-07

$objective
[1] 1
```
Further improvements

- As with root finding, performance can improve substantially when local approximation is used.
- However, a linear approximation won’t do in this case.
Approximation Using Parabola
Better optimization using local approximation

- Root finding example
  - Binary search reduces the search space by constant factor $1/2$
  - Linear approximation may reduce the search space more rapidly for most well-defined functions

- Minimization problem
  - Golden search reduces the search space by 38%
  - Using a quadratic approximation of the function may achieve better optimization results
Approximation using parabola

![Diagram showing parabolas approximating a curve at points 1, 2, 3, 4, and 5.](image)
Parabolic Approximation

\[ f^*(x) = Ax^2 + Bx + C \]

The value minimizes \( f^*(x) \) is

\[ x_{min} = -\frac{B}{2A} \]

This strategy is called "inverse parabolic interpolation"
Fitting a parabola

- Can be fitted with three points
- Points must not be co-linear
- \( f^*(x_1) = f(x_1), f^*(x_2) = f(x_2), f^*(x_3) = f(x_3). \)

\[
C = f(x_1) - A x_1^2 - B x_1 \\
B = \frac{A (x_2^2 - x_1^2) + f(x_1) - f(x_2)}{x_1 - x_2} \\
A = \frac{f(x_3) - f(x_2)}{(x_3 - x_2)(x_3 - x_1)} - \frac{f(x_1) - f(x_2)}{(x_1 - x_2)(x_3 - x_1)}
\]
Minimum for a Parabola

- General expression for finding minimum of a parabola fitted through three points

\[ x_{\text{min}} = x_2 - \frac{1}{2} \left( \frac{(x_2 - x_1)^2(f(x_2) - f(x_1)) - (x_2 - x_3)^2(f(x_2) - f(x_1))}{(x_2 - x_1)(f(x_2) - f(x_3))} \right) \]
Fitting a Parabola

// Returns the distance between b and the abscissa for the
// fitted minimum using parabolic interpolation

double parabolaStep (double a, double fa, double b, double fb, double c,
double fc) {

    // Quantities for placing minimum of fitted parabola
    double p = (b - a) * (fb - fc);
    double q = (b - c) * (fb - fa);
    double x = (b - c) * q - (b - a) * p;
    double y = 2.0 * (p - q);

    // Check that y is not too close to zero
    if (fabs(y) < ZEPS)
        return goldenStep (a, b, c);
    else
        return x / y;
}
Avoiding degenerate case

- Fitted minimum could overlap with one of original points
- Ensure that each new point is distinct from previously examined points
Avoiding degenerate steps

double adjustStep(double a, double b, double c, double step, double e) {
    double minStep = fabs(e * b) + ZEPS;
    if (fabs(step) < minStep)
        return step > 0 ? minStep : 0-minStep;
    // If the step ends up to close to previous points,
    // return zero to force a golden ratio step ...
    if (fabs(b + step - a) <= e || fabs(b + step - c) <= e)
        return 0.0;
    return step;
}
Generating New Points

- Use parabolic interpolation by default
- Check whether improvement is slow
- If step sizes are not decreasing rapidly enough, switch to golden section
Adaptive calculation of step size

define calculateStep(double a, double fa, double b, double fb, double c, double fc, double lastStep, double e) {
    double step = parabolaStep(a, fa, b, fb, c, fc);
    step = adjustStep(a, b, c, step, e);
    if (fabs(step) > fabs(0.5 * lastStep) || step == 0.0)
        step = goldenStep(a, b, c);
    return step;
}
Overall

The main function simply has to

- Generate new points using building blocks
- Update the triplet bracketing the minimum
- Check for convergence
Root Finding

Minimization

Parabola

Boost

Summary

**Overall Minimization Routine**

```cpp
template<class F>
double adaptiveMinimum(F foo, double a, double b, double c, double e) {
    double fa = foo(a), fb = foo(b), fc = foo(c);
    double step1 = (c - a) * 0.5, step2 = (c - a) * 0.5;
    while (fabs(c - a) > fabs(b * e) + ZEPS) {
        double step = calculateStep(a, fa, b, fb, c, fc, step2, e);
        double x = b + step;
        double fx = foo(x);
        if (fx < fb) {
            if (x > b) { a = b; fa = fb; }
            else { c = b; fc = fb; }
            b = x; fb = fx;
        }
        else {
            if (x < b) { a = x; fa = fx; }
            else { c = x; fc = fx; }
            step2 = step1; step1 = step;
        }
    }
    return b;
}
```

Hyun Min Kang
Biostatistics 615/815 - Lecture 17
November 13th, 2012 44 / 49
Important Characteristics

- Parabolic interpolation often converges faster
  - The preferred algorithm
- Golden search provides worst-case performance guarantee
  - A fall-back for uncooperative functions
- Switch algorithms when convergence is slow
- Avoid testing points that are too close
More advanced strategy: Brent’s algorithm

- Track 6 points (not all distinct)
  - The bracket boundaries \((a, b)\)
  - The current minimum \(x\)
  - The second and third smallest value \((w, v)\)
  - The new points to be examined \(u\)

- Parabolic interpolation
  - Using \((x, w, v)\) to propose new value for \(u\).
  - Additional care is required to ensure \(u\) falls between \(a\) and \(b\).

- Recommended Reading
  - Numerical Recipes in C++: Chapter 10.0 - 10.3
#include <cmath>
#include <iostream>
#include <boost/math/tools/roots.hpp>

#define EPS 1e-6

bool tol(double a, double b) { return (fabs(b-a) <= EPS); }

int main(int argc, char** argv) {
  double lo = 0-M_PI/4;
  double hi = M_PI/2;
  boost::uintmax_t niter;
  std::pair<double,double> rBi = boost::math::tools::bisect(sin, lo, hi, tol, niter);
  std::cout << "bisect: (" << rBi.first << ", " << rBi.second << ") at " << niter << " iterations" << std::endl;
  std::pair<double,double> r748 =
    boost::math::tools::toms748_solve(sin, lo, hi, tol, niter);
  std::cout << "toms748: (" << r748.first << ", " << r748.second << ") at " << niter << " iterations" << std::endl;
  return 0;
}
Other Algorithms for Root Finding

- **TOMS Algorithm 748**
  - Uses a mixture of cubic, quadratic, and linear interpolation to locate the root of $f(x)$.

- **Newton-Raphson algorithm**
  - Uses first derivative of $f(x)$ to better approximate the root

- **Halley’s method**
  - Uses first and second derivatives of $f(x)$ to approximate the root

- **Householder’s method**
  - Uses up to $d$-th derivative of $f(x)$ to approximate the root for faster convergence
Summary

Root Finding Algorithms

- Bisection Method: Simple but likely less efficient
- False Position Method: More efficient for most well-behaved function

Single-dimensional minimization

- Golden Search: 38% reduction of interval per iteration
- Parabola Method: Likely more efficient reduction, but not always guaranteed.
- Brent’s Method: Combination of above two methods. More efficient than both.