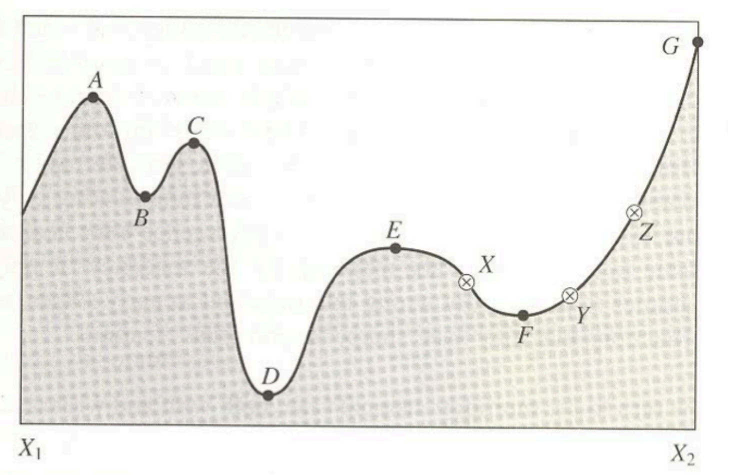
Biostatistics 615/815 Lecture 13: Numerical Optimization

Hyun Min Kang

October 27th, 2011

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 $f(x) = 0$?

A C++ Example : definining a function object

```
#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++

```
// 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 accruate)
    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 pointss
  double fhi = foo(hi);
  for(int i=0;;++i) {
    double d = hi - lo;
    double point = lo + d * flo / (flo - fhi); //
    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 $\pi/2$

```
#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

```
> uniroot( sin, c(0-pi/4,pi/2) )
$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 converage 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 ϵ, 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 $\epsilon^{1/4}$.

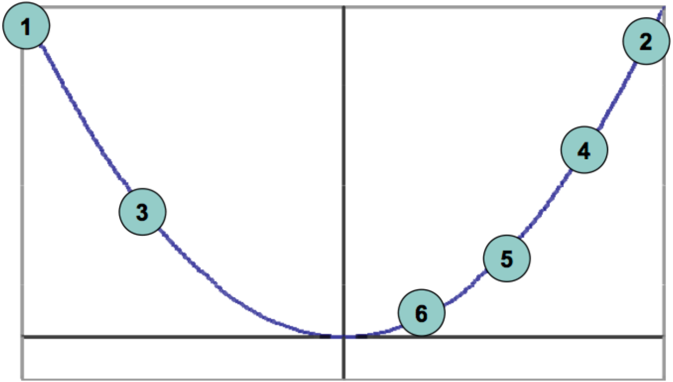
Outline of Minimization Strategy

1. Bracket minimum
2. Successively tighten bracket interval

Detailed 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

Minimization after Bracketing



Part I : Finding a Bracketing Interval

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

Bracketing in C++

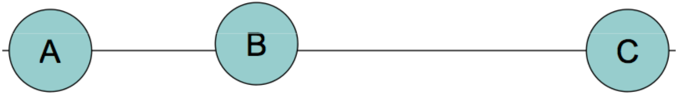
```
#define SCALE 1.618

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);
  }
}
```

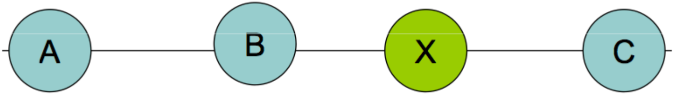
Part II : Finding Minimum After Bracketing

- 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

Minimizing worst case possibility

- Formulae

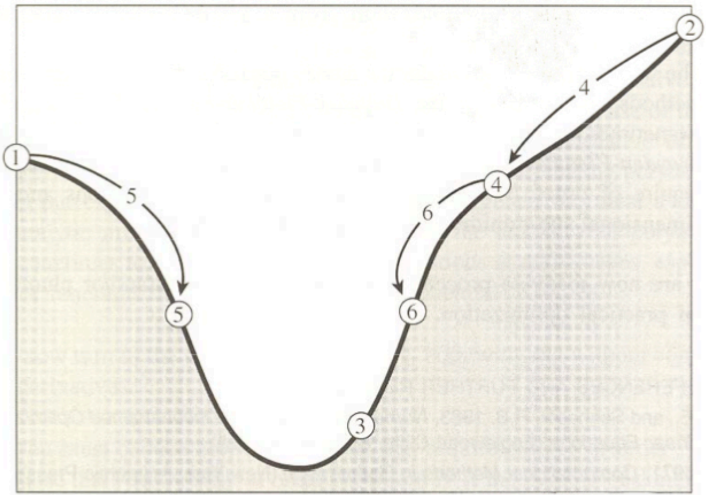
$$w = \frac{b-a}{c-a}$$
$$z = \frac{x-b}{c-a}$$

Segments will have length either $1-w$ or $w+z$.

- Optimal case

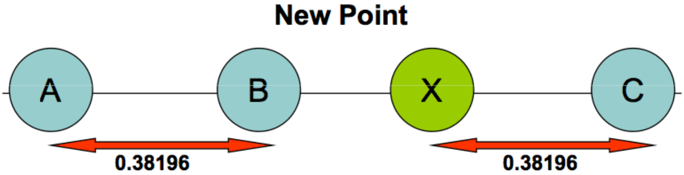
$$1-w = w+z$$
$$\frac{z}{1-w} = w$$
$$w = \frac{3-\sqrt{5}}{2} = 0.38197$$

The Golden Search



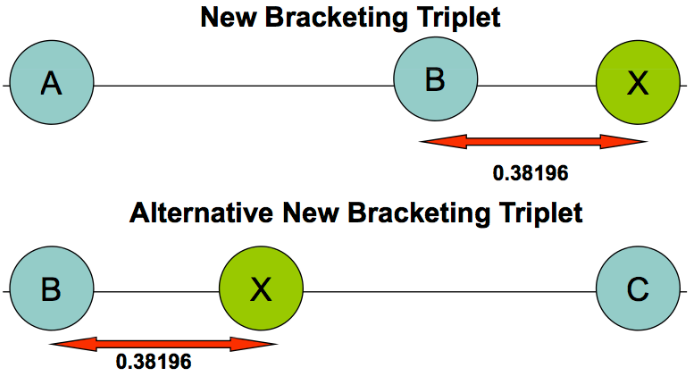
The Golden Ratio

The Golden Ratio



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

The Golden Ratio



Golden Search

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

Golden Step

```
#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

```
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;
}
```

A running example

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

```
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

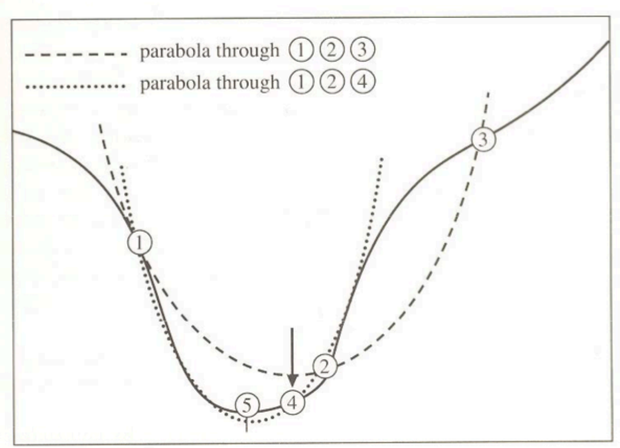
```
> optimize(cos,interval=c(0-pi/4,pi/2),maximum=TRUE)
$maximum
[1] -8.648147e-07

$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



Summary

Today

- 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

Next Lecture

- More Single-dimensional minimization
  - Brent's method
- Multidimensional optimization
  - Simplex method