Introduction to Numerical Optimization

Biostatistics 615/815
Lecture 15
Course is More Than Half Done!

- If you have comments…
- … they are very welcome
  - Lectures
  - Lecture notes
  - Weekly Homework
  - Midterm
  - Content
Last Lecture

- Computer generated “random” numbers

- Linear congruential generators
  - Improvements through shuffling, summing

- Importance of using validated generators
  - Beware of problems with the default \texttt{rand()} function
Today ...

- Root finding
- Minimization for functions of one variable

Ideas:
- Limits on accuracy
- Local approximations
Numerical Optimization

- Consider some function $f(x)$
  - e.g. Likelihood for some model …

- Find the value of $x$ for which $f$ takes a maximum or minimum value

- Maximization and minimization are equivalent
  - Replace $f(x)$ with $-f(x)$
Algorithmic Objectives

- Solve problem...
  - Conserve CPU time
  - Conserve memory

- Most often, the CPU time is dominated by the cost of evaluating \( f(x) \)
  - Minimize the number of evaluations
The Minimization Problem
Specific Objectives

- Finding global minimum
  - *The* lowest possible value of the function
  - Extremely hard problem

- Finding local minimum
  - Smallest value within finite neighborhood
**Typical Quality Checks**

- When solving an optimization problem it is good practice to check the quality of the solution.
- Try different starting values …
- Perturb solution and repeat …
A Quick Detour

- 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 $f(x)=0$?
double zero(double (*func)(double), double lo, double hi, double e)
{
    while (1)
    {
        double d = hi - lo;
        double point = lo + d * 0.5;
        double fpoint = (*func)(point);

        if (fpoint < 0.0)
            { d = lo - point; lo = point; }
        else
            { d = point - hi; hi = point; }

        if (fabs(d) < e || fpoint == 0.0)
            return point;
    }
}
Improvements to Root Finding

- Consider the following approximation:

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

- Select new trial point such that \( f^*(x) \) is zero.
Improved Root Finding in C

double zero (double (*func)(double), double lo, double hi, double e)
{
    double flo = (*func)(lo);
    double fhi = (*func)(hi);
    while (1)
    {
        double d = hi - lo;
        double point = lo + d * flo / (flo - fhi);
        double fpoint = (*func)(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)
            return point;
    }
}
Performance Comparison

- Find the zero for $\sin(x)$
  - In the interval $-\pi/4$ to $\pi/2$
  - Accuracy parameter set to $10^{-5}$

- Bisection method used 17 calls to $\sin(x)$

- Approximation used 5 calls to $\sin(x)$
Program That Uses Root Finding

double zero (double (*func)(double), double lo, double hi, double e);

double function(double x)
{
    return (4*x - 3);
}

int main(int argc, char ** argv)
{
    double min = zero(my_function, -5, +5, 1e-5);

    printf("Minimum for my function is %.3f at %.3f\n", 
            my_function(min), min);
}

Notes on Root Finding

- The 2\textsuperscript{nd} method we implemented is the False Position Method.

- In the bisection method, the bracketing interval is halved at each step.

- For well-behaved functions, the False Position Method will converge faster, but there is no performance guarantee.
Questions on Root Finding

- What care is required in setting precision?

- How to set starting brackets for minimum?
  - If the function was monotonic?
  - If there is a specific target interval?

- What would happen for a function such as
  \[ f(x) = \frac{1}{x - c} \]
Consider some function $f(x)$
- e.g. Likelihood for some model …

Find the value of $x$ for which $f$ takes a maximum or minimum value

Maximization and minimization are equivalent
- Replace $f(x)$ with $-f(x)$
Notes from Root Finding

- Introduces two useful ideas
  - Which can be applied to function minimization

- Bracketing
  - Keep track of interval containing solution

- Accuracy
  - Recognize that solution has limited precision
Note on Accuracy

- When estimating minima and bracketing intervals, floating point accuracy must be considered.

- In general, if the machine precision is $\varepsilon$, the achievable accuracy is no more than $\sqrt{\varepsilon}$. 
The error results from the second term in the 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

- Part I
  - Bracket minimum

- Part II
  - Successively tighten bracketing interval
Detailed Minimization Strategy

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

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

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

- Take successively larger steps beyond $b$ until function starts increasing
Bracketing in C

#define SCALE   1.618

void bracket (double (*f)(double), double* a, double* b, double* c)
{
    double fa = (*f)( *a);
    double fb = (*f)( *b);
    double fc = (*f)( *c = *b + SCALE * (*b - *a) );

    while (fb > fc)
    {
        *a = *b; fa = fb;
        *b = *c; fb = fc;
        *c = *b + SCALE * (*b - *a);
        fc = (*f) (*c);
    }
}
Bracketing in C++

#define SCALE  1.618

void bracket (double (*f)(double), double & a, double & b, double & c)
{
    double fa = (*f)(a);
    double fb = (*f)(b);
    double fc = (*f)(c = b + SCALE * (b - a));

    while (fb > fc)
    {
        a = b; fa = fb;
        b = c; fb = fc;
        c = b + SCALE * (b - a);
        fc = (*f) (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?
Consider ...

What is the best location for a new point X?
Consider ...

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

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

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

Segments will have length

\[ 1-w \text{  or  } w+z \]

We want to minimize worst case possibility so...
Effectively ...

The optimal case is

\[ z = 1 - 2w \]

\[ \frac{z}{1 - w} = w \]

This gives

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

Bracketing Triplet

A

B

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

New Bracketing Triplet

A

B

X

Alternative New Bracketing Triplet

B

X

C

0.38196

0.38196
Golden Search

- Reduces bracketing by ~40% after each function evaluation
- Performance is independent of the function that is being minimized
- Potentially, better schemes are possible
Golden Step

```c
#define GOLD   0.38196
#define ZEPS   1e-10

double golden_step (double a, double b, double c)
{
    double mid = (a + c) * 0.5;

    if (b > mid)
        return GOLD * (a - b);
    else
        return GOLD * (c - b);
}
```
**Golden Search**

```c
double golden_search(double (*func)(double),
                     double a, double b, double c, double e)
{
    double fb = (*func)(b);

    while (fabs(c - a) > fabs(b * e) + ZEPS)
    {
        double x = b + golden_step(a, b, c);
        double fx = (*func)(x);

        if (fx < fb)
            {
                if (x > b) a = b; else c = b;
                b = x; fb = fx;
            }
        else
            if (x < b) a = x; else c = x;
    }

    return b;
}
```
Further Improvements

- As with root finding, performance can improve substantially when a local approximation is used …

- However, a linear approximation won't do in this case!
Approximating The Function

- Parabola through \( 1 \), \( 2 \), \( 3 \)
- Dotted parabola through \( 1 \), \( 2 \), \( 4 \)
Recommended Reading

- Numerical Recipes in C (or C++)
  - Press, Teukolsky, Vetterling, Flannery
  - Chapters 10.0 – 10.2

- Excellent resource for scientific computing

- Online at
  - http://www.numerical-recipes.com/
  - http://www.library.cornell.edu/nr/