Multidimensional Optimization: The Simplex Method

Lecture 17
Biostatistics 615/815
Previous Topic: One-Dimensional Optimization

- Bracketing
- Golden Search
- Quadratic Approximation
Bracketing

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

- Locate minimum by gradually trimming bracketing interval

- Bracketing provides additional confidence in result
The Golden Ratio

Bracketing Triplet

New Point

A

B

C

A

B

X

C

0.38196

0.38196
Parabolic Interpolation

For well behaved functions, faster than Golden Search
Today:
Multidimensional Optimization

- Illustrate the method of Nelder and Mead
  - Simplex Method
  - Nicknamed "Amoeba"

- Simple and, in practice, quite robust
  - Counter examples are known

- Discuss other standard methods
C Utility Functions: Allocating Vectors

- Ease allocation of vectors.
- Peppered through today's examples

```c
#include <stdlib.h>

double * alloc_vector(int cols)
{
    return (double *) malloc(sizeof(double) * cols);
}

void free_vector(double * vector, int cols)
{
    free(vector);
}
```
C Utility Functions: Allocating Matrices

double ** alloc_matrix(int rows, int cols)
{
    int i;
    double ** matrix = (double **) malloc(sizeof(double *) * rows);
    
    for (i = 0; i < rows; i++)
        matrix[i] = alloc_vector(cols);
    
    return matrix;
}

void free_matrix(double ** matrix, int rows, int cols)
{
    int i;
    for (i = 0; i < rows; i++)
        free_vector(matrix[i], cols);
    
    free(matrix);
}
The Simplex Method

- Calculate likelihoods at simplex vertices
  - Geometric shape with k+1 corners
  - E.g. a triangle in k = 2 dimensions

- Simplex *crawls*
  - Towards minimum
  - Away from maximum

- Probably the most widely used optimization method
A Simplex in Two Dimensions

- Evaluate function at vertices

- Note:
  - The highest (worst) point
  - The next highest point
  - The lowest (best) point

- Intuition:
  - Move away from high point, towards low point
C Code:
Creating A Simplex

double ** make_simplex(double * point, int dim)
{
    int i, j;
    double ** simplex = alloc_matrix(dim + 1, dim);

    for (int i = 0; i < dim + 1; i++)
        for (int j = 0; j < dim; j++)
            simplex[i][j] = point[j];

    for (int i = 0; i < dim; i++)
        simplex[i][i] += 1.0;

    return simplex;
}
This function is very simple
- This is a good thing!
- Making each function almost trivial makes debugging easy

```c
void evaluate_simplex
    (double ** simplex, int dim,
     double * fx, double (* func)(double *, int))
{
    for (int i = 0; i < dim + 1; i++)
        fx[i] = (*func)(simplex[i], dim);
}
```
void simplex_extremes(double *fx, int dim, int &ihī, int &ilo, int &inhī)
{
    int i;

    if (fx[0] > fx[1])
        { ihī = 0; ilo = inhi = 1; }
    else
        { ihī = 1; ilo = inhi = 0; }

    for (i = 2; i < dim + 1; i++)
        if (fx[i] <= fx[ilo])
            ilo = i;
        else if (fx[i] > fx[ihī])
            { inhi = ihī; ihī = i; }
        else if (fx[i] > fx[inhi])
            inhi = i;
Direction for Optimization

$$x_0$$

$$x_1$$

$$x_2$$

**Average of all points, excluding worst point**

**Line through worst point and average of other points**
void simplex Bearings(double ** simplex, int dim, 
                      double * midpoint, double * line, int ihi)
{
    int i, j;
    for (j = 0; j < dim; j++)
        midpoint[j] = 0.0;

    for (i = 0; i < dim + 1; i++)
        if (i != ihi)
            for (j = 0; j < dim; j++)
                midpoint[j] += simplex[i][j];

    for (j = 0; j < dim; j++)
    {
        midpoint[j] /= dim;
        line[j] = simplex[ihi][j] - midpoint[j];
    }
}
Reflection

This is the default new trial point
Reflection and Expansion

If reflection results in new minimum...

Move further along minimization direction
Contraction (One Dimension)

Try a smaller step

If $x'$ is still the worst point…
int update_simplex(double * point, int dim, double & fmax, double * midpoint, double * line, double scale, double (* func)(double *, int))
{
int i, update = 0; double * next = alloc_vector(dim), fx;

for (i = 0; i < dim; i++)
    next[i] = midpoint[i] + scale * line[i];
fx = (*func)(next, dim);

if (fx < fmax)
{
    for (i = 0; i < dim; i++)
        point[i] = next[i];
    fmax = fx;
    update = 1;
}

free_vector(next, dim);
return update;
}
If a simple contraction doesn't improve things, then try moving all points towards the current minimum.

"passing through the eye of a needle"
void contract_simplex(double ** simplex, int dim, double * fx, int ilo, double (*func)(double *, int))
{
    int i, j;

    for (int i = 0; i < dim + 1; i++)
        if (i != ilo)
        {
            for (int j = 0; j < dim; j++)
                simplex[i][j] = (simplex[ilo][j] + simplex[i][j]) * 0.5;
            fx[i] = (*func)(simplex[i], dim);
        }
}
Summary: The Simplex Method

- Original Simplex
- High
- Low
- Reflection
- Contraction
- Reflection and Expansion
- Multiple Contraction
C Code: Minimization Routine (Part I)

- Declares local variables and allocates memory

```c
double amoeba(double *point, int dim,
               double (*func)(double *, int),
               double tol)
{
    int     ihi, ilo, inhi, j;
    double fmin;
    double * fx = alloc_vector(dim + 1);
    double * midpoint = alloc_vector(dim);
    double * line = alloc_vector(dim);
    double ** simplex = make_simplex(point, dim);

    evaluate_simplex(simplex, dim, fx, func);
```
while (true)
{
    simplex_extremes(fx, dim, ihi, ilo, inhi);
    simplex_bearings(simplex, dim, midpoint, line, ihi);

    if (check_tol(fx[ihi], fx[ilo], tol)) break;

    update_simplex(simplex[ihi], dim, fx[ihi],
                    midpoint, line, -1.0, func);

    if (fx[ihi] < fx[ilo])
        update_simplex(simplex[ihi], dim, fx[ihi],
                        midpoint, line, -2.0, func);
    else if (fx[ihi] >= fx[inhi])
        if (!update_simplex(simplex[ihi], dim, fx[ihi],
                            midpoint, line, 0.5, func))
            contract_simplex(simplex, dim, fx, ilo, func);
}
C Code: Minimization Routine (Part III)

- Store the result and free memory

```c
for (j = 0; j < dim; j++)
    point[j] = simplex[ilo][j];

fmin = fx[ilo];

free_vector(fx, dim);
free_vector(midpoint, dim);
free_vector(line, dim);
free_matrix(simplex, dim + 1, dim);

return fmin;
}
```
#include <math.h>

#define ZEPS 1e-10

int check_tol(double fmax, double fmin, double ftol)
{
    double delta = fabs(fmax - fmin);
    double accuracy = (fabs(fmax) + fabs(fmin)) * ftol;

    return (delta < (accuracy + ZEPS));
}
A general purpose minimization routine
  • Works in multiple dimensions
  • Uses only function evaluations
  • Does not require derivatives

Typical usage:
  • `my_func(double * x, int n) { ... }`
  • `amoeba(point, dim, my_func, 1e-7);`
Example Application
Old Faithful Eruptions (n = 272)
Fitting a Normal Distribution

- Fit two parameters
  - Mean
  - Variance

- Requires ~165 likelihood evaluations
  - Mean = 3.4878
  - Variance = 1.2979

- Maximum log-likelihood = -421.42
Nice fit, eh?

Old Faithful Eruptions

Fitted Distribution
A Mixture of Two Normals

- Fit 5 parameters
  - Proportion in the first component
  - Two means
  - Two variances

- Required about ~700 evaluations
  - First component contributes 0.34841 of mixture
  - Means are 2.0186 and 4.2734
  - Variances are 0.055517 and 0.19102

- Maximum log-likelihood = -276.36
Two Components

Old Faithful Eruptions

Fitted Distribution

Duration (mins)

Frequency

Density

Data points and fitted distribution for Old Faithful eruptions, showing the duration in minutes and the frequency or density of eruptions.
A Mixture of Three Normals

- Fit 8 parameters
  - Proportion in the first two components
  - Three means
  - Three variances
- Required about ~1400 evaluations
  - Did not always converge!
- One of the best solutions ...
  - Components contributing .339, 0.512 and 0.149
  - Component means are 2.002, 4.401 and 3.727
  - Variances are 0.0455, 0.106, 0.2959
  - Maximum log-likelihood = -267.89
Three Components

Old Faithful Eruptions

Fitted Distribution

<table>
<thead>
<tr>
<th>Duration (mins)</th>
<th>Frequency</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0.00</td>
</tr>
</tbody>
</table>

- Duration (mins): 1, 2, 3, 4, 5, 6
- Frequency: 20, 15, 10, 5, 2, 1
- Density: 0.00, 0.00, 0.00, 0.00, 0.00, 0.00
Tricky Minimization Questions

- Fitting variables that are constrained
  - Proportions vary between 0 and 1
  - Variances must be positive

- Selecting the number of components

- Checking convergence
Improvements to amoeba()

- Different scaling along each dimension
  - If parameters have different impact on the likelihood

- Track total function evaluations
  - Avoid getting stuck if function does not cooperate

- Rotate simplex
  - If the current simplex is leading to slow improvement
optim() Function in R

- optim(point, function, method)

  - Point – starting point for minimization
  - Function that accepts point as argument
  - Method can be
    - "Nelder-Mead" for simplex method (default)
    - "BFGS", "CG" and other options use gradient
One parameter at a time

- Simple but inefficient approach

- Consider
  - Parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$
  - Function $f(\theta)$

- Maximize $\theta$ with respect to each $\theta_i$ in turn
  - Cycle through parameters
The Inefficiency...
Steepest Descent

- Consider
  - Parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_k)$
  - Function $f(\theta; x)$

- Score vector

$$S = \frac{d \ln f}{d \theta} = \left( \frac{d \ln f}{d \theta_1}, \ldots, \frac{d \ln f}{d \theta_k} \right)$$

- Find maximum along $\theta + \delta S$
Still inefficient...

Consecutive steps are still perpendicular!
Typically, sophisticated methods will...

Use derivatives
- May be calculated numerically. How?

Select a direction for minimization, using:
- Weighted average of previous directions
- Current gradient
- Avoid right angle turns
Recommended Reading

- Numerical Recipes in C (or C++, or Fortran)
  - Press, Teukolsky, Vetterling, Flannery
  - Chapter 10.4

- Clear description of Simplex Method
  - Other sub-chapters illustrate more sophisticated methods

- Online at
  - [http://www.numerical-recipes.com/](http://www.numerical-recipes.com/)