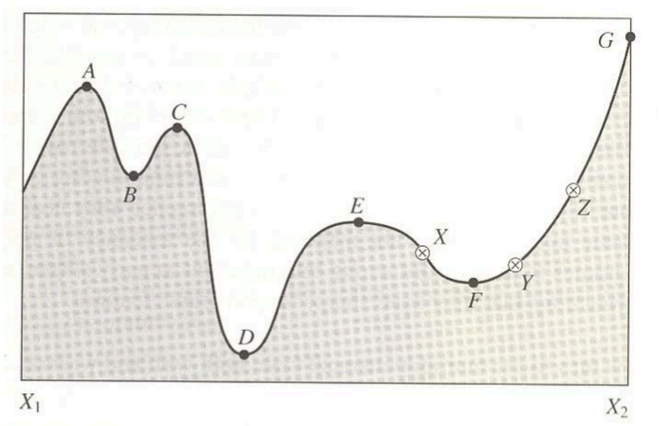
Biostatistics 615 - Statistical Computing

Lecture 16
Optimizations

Jian Kang

Nov 19, 2015

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

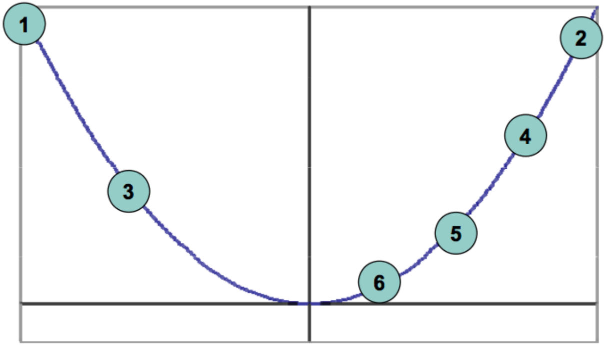
Maximization Problem

- 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 $-f(x)$

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

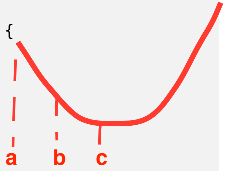


Step 1: Finding a Bracketing Interval

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

Bracketing in C++

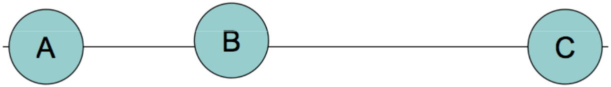
```
#define SCALE 1.618

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) ); if b>a then c >b
  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.
}
```



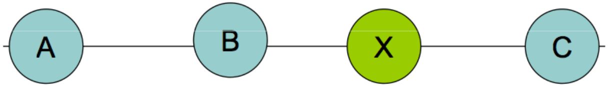
Step 2 : Finding Minimum After Bracketing

- 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

$$\begin{aligned} w &= \frac{b-a}{c-a} \\ z &= \frac{x-b}{c-a} \end{aligned}$$

Segments will have length either $1 - w$ or $w + z$.
[B,C] **[A,X]**

- Optimal case

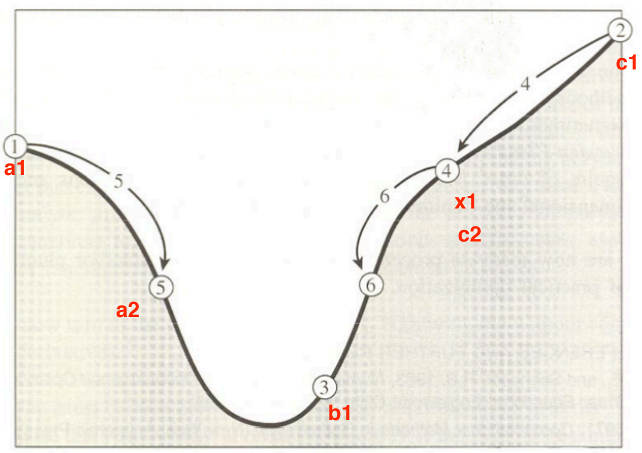
$$\begin{cases} 1 - w = w + z \\ \frac{z}{1-w} = w \end{cases}$$

[B,C] = [A,X]
[B,X]/[B,C] = [A,B]/[A,C]

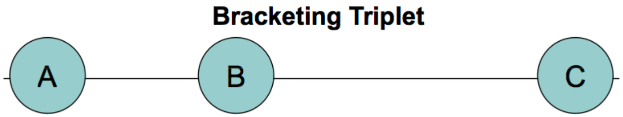
- Solve It

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

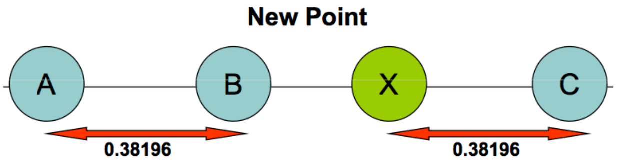
The Golden Search



The Golden Ratio

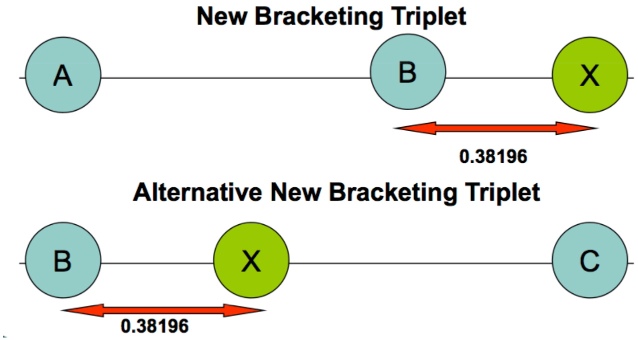


The Golden Ratio



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

```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

```
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
```

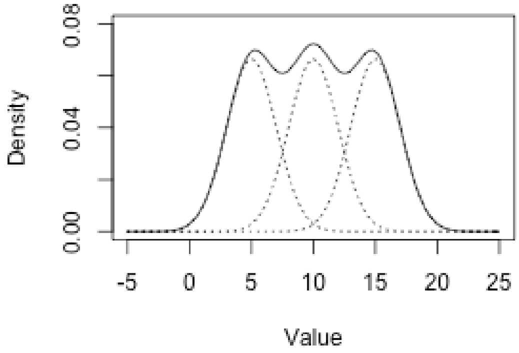
R example of minimization

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

$objective
[1] 1
```

Other algorithms

- Parabola Method: Using a quadratic approximation of the function may achieve better optimization results; Likely more efficient reduction, but not always guaranteed.
- Brent's Method: Combination of above two methods. More efficient than both.
- The Newton-Raphson Method: Quadratic convergence. Requires derivatives.

Multidimensional Optimization : A mixture distribution



A general mixture distribution

$$p(x; \pi, \phi, \eta) = \sum_{i=1}^{k} \pi_i f(x; \phi_i, \eta)$$

x	observed data
π	mixture proportion of each component
f	the probability density function
ϕ	parameters specific to each component
η	parameters shared among components
k	number of mixture components

Problem : Maximum Likelihood Estimation

Finding Maximum-likelihood

Find parameters that maximizes the likelihood of the entire sample

$$L = \prod_i p(x_i|\pi, \phi, \eta)$$

Calculating in log-space

Or equivalently, consider log-likelihood to avoid underflow

$$l = \sum_i \log p(x_i|\pi, \phi, \eta)$$

Gaussian MLE in single-dimensional space

$$p(x; \mu, \sigma^2) = \mathcal{N}(x; \mu, \sigma^2)$$

Given x, what is the MLE parameters of μ and σ^2?

- Analytical solution does exist
- $\hat{\mu} = \sum_{i=1}^{n} x_i / n$
- $\hat{\sigma}^2 = \sum_{i=1}^{n} (x_i - \hat{\mu})^2 / n$

MLE in Gaussian mixture

Parameter estimation in Gaussian mixture

- No analytical solution
- Numerical optimization required
- Multi-dimensional optimization problem
  - $\pi_1, \pi_2, \ldots, \pi_k$
  - $\mu_1, \mu_2, \ldots, \mu_k$
  - $\sigma_1^2, \sigma_2^2, \ldots, \sigma_k^2$

Possible methods for multi-dimensional optimization

- Nelder-Mead Method
- Gradient Descent
- Newton's Method
- Expectation Maximization
- Simulated Annealing
- Markov-Chain Monte Carlo

The Nelder-Mead Method

- a.k.a. downhill simplex method or amoeba method
- Calculate likelihoods at simplex vertexes
  - Geometric shape with $k+1$ corners
  - A triangle in $k=2$ dimensions
  - A tetrahedron in $k=3$ dimensions
- Simplex *crawls*
  - Towards minimum
  - Away from maximum
- A commonly used nonlinear optimization method, without using derivatives.
- May converge to non-stationary points.

(http://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method)

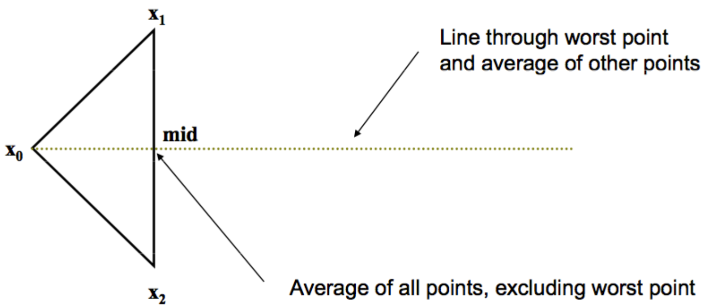
How the Nelder-Mead Method Works

(http://userpages.umbc.edu/~rostamia/2013-01-math625/images/nelder-mead.gif)

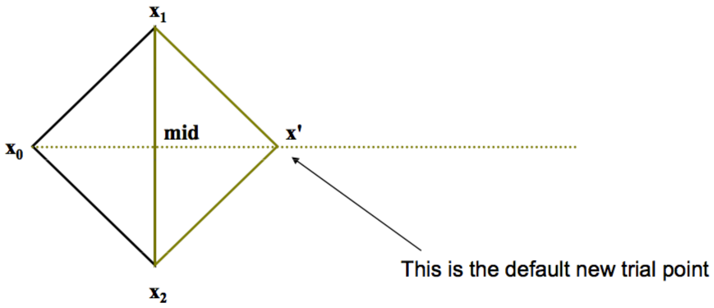
Nelder-Mead Method in Two Dimensions

- Evaluate functions at three vertexes
  - The highest (worst) point
  - The next highest point
  - The lowest (best) point
- Intuition
  - Move away from high point, towards low point

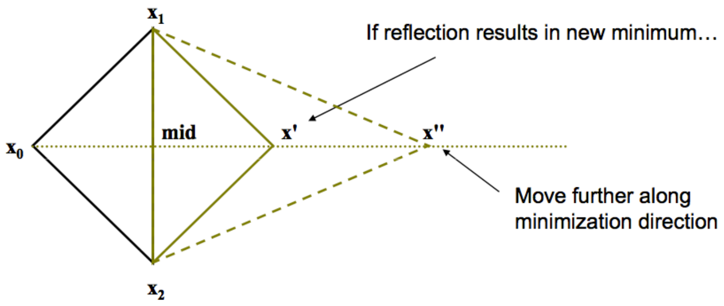
Direction for Optimization


$\mathbf{x}_1$
Line through worst point
and average of other points
mid
$\mathbf{x}_0$
$\mathbf{x}_2$
Average of all points, excluding worst point

Reflection

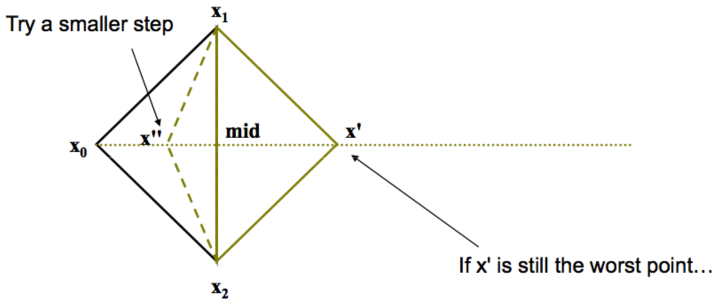


Reflection and Expansion

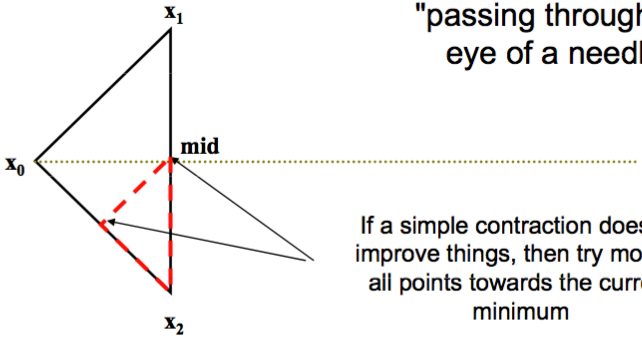


x_1
If reflection results in new minimum…
mid
x_0
x'
x''
Move further along minimization direction
x_2

Contraction (1-dimension)



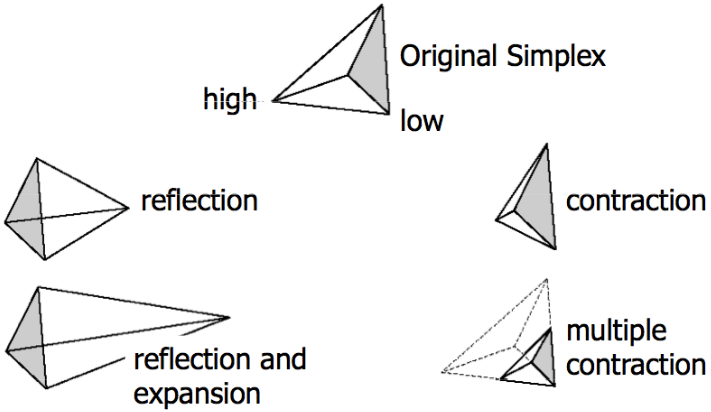
Multiple Contraction



"passing through the eye of a needle"

If a simple contraction doesn't improve things, then try moving all points towards the current minimum

Summary : The Nelder-Mead Method (in 3-D)



R function for optimization

General purpose optimizations

```
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

Example,

```
> fn = function(x)
+ return(100*(x[2]-x[1]^2)^2 + (1-x[1])^2)
> results = optim(par=c(-1.2,1),fn=fn,method="Nelder-Mead")
> results$par
[1] 1.000260 1.000506
> results$value
[1] 8.825241e-08
```

Implementing the Nelder-Mead Method in C++

```
template <class F> // F is a function object
class simplex615 { // contains (dim+1) points of size (dim)
 protected:
  std::vector<std::vector<double> > X; // (dim+1)*dim matrix
  std::vector<double> Y;               // (dim+1) vector
  std::vector<double> midPoint;        // variables for update
  std::vector<double> thruLine;        // variables for update
  int dim, idxLo, idxHi, idxNextHi;    // dimension, min, max, 2ndmax values
  void evaluateFunction(F& foo); // evaluate function value at each point
  void evaluateExtremes();             // determine the min, max, 2ndmax
  void prepareUpdate();                // calculate midPoint, thruLine
  bool updateSimplex(F& foo, double scale);  // for reflection/expansion..
  void contractSimplex(F& foo);  // for multiple contraction
  static int check_tol(double fmax, double fmin, double ftol); // check tolerance
 public:
  simplex615(double* p, int d);          // constructor with initial points
  void amoeba(F& foo, double tol); // main function for optimization
  std::vector<double>& xmin();           // optimal x value
  double ymin();                         // optimal y value
};
```

Implementation overview

- Data representation
  - Each `X[i]` is point of the simplex
  - `Y[i]` corresponds to $f(X[i])$
  - `midPoint` is the average of all points (except for the worst point)
  - `thruLine` is vector from the worst point to the `midPoint`

Implementation overview

- Data representation
  - Each `X[i]` is point of the simplex
  - `Y[i]` corresponds to $f(X[i])$
  - `midPoint` is the average of all points (except for the worst point)
  - `thruLine` is vector from the worst point to the `midPoint`
- Reflection, Expansion and Contraction
  After calculating `midPoint` and `thruLine`

Reflection	Call `updateSimplex(foo, -1.0)`
Expansion	Call `updateSimplex(foo, -2.0)`
Contraction	Call `updateSimplex(foo, 0.5)`

Initializing a Simplex

```cpp
// constructor of simplex615 class : initial point is given
template <class F>
simplex615<F>::simplex615(double* p, int d) : dim(d) { // set dimension
  // Determine the space required
  X.resize(dim+1);      // X is vector-of-vector, like 2-D array
  Y.resize(dim+1);      // Y is function value at each simplex point
  midPoint.resize(dim);
  thruLine.resize(dim);
  for(int i=0; i < dim+1; ++i) {
    X[i].resize(dim);  // allocate the size of content in the 2-D array
  }
  // Initially, make every point in the simplex identical
  for(int i=0; i < dim+1; ++i)
    for(int j=0; j < dim; ++j)
      X[i][j] = p[j];  // set each simple point to the starting point
  // then increase each dimension by one unit except for the last point
  for(int i=0; i < dim; ++i)
    X[i][i] += 1.;     // this will generate a simplex
}
```

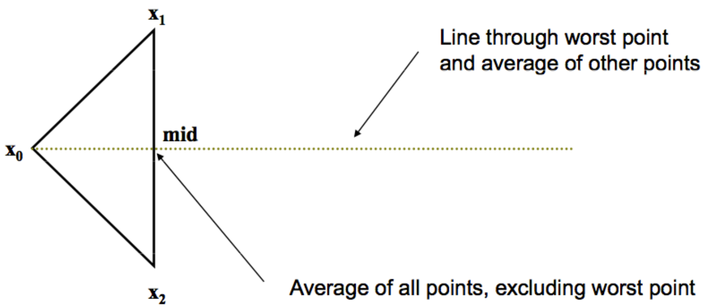
Evaluating function values at each simplex point

```
// simple function for evaluating the function value at each simple point
// after calling this function Y[i] = foo(X[i]) should hold
template <class F>
void simplex615<F>::evaluateFunction(F& foo) {
  for(int i=0; i < dim+1; ++i) {
    Y[i] = foo(X[i]);  // foo is a function object, which will be visited later
  }
}
```

Determine the best, worst, and the second-worst points

```
template <class F>
void simplex615<F>::evaluateExtremes() {
  if ( Y[0] > Y[1] ) {  // compare the first two points
    idxHi = 0; idxLo = idxNextHi = 1;
  }
  else {
    idxHi = 1; idxLo = idxNextHi = 0;
  }
  // for each of the next points
  for(int i=2; i < dim+1; ++i) {
    if ( Y[i] <= Y[idxLo] )           // update the best point if lower
      idxLo = i;
    else if ( Y[i] > Y[idxHi] ) {     // update the worst point if higher
      idxNextHi = idxHi; idxHi = i;
    }
    else if ( Y[i] > Y[idxNextHi] )  // update also if it is the 2nd-worst point
      idxNextHi = i;
  }
}
```

Direction for Optimization


$\mathbf{x}_1$
Line through worst point
and average of other points
mid
$\mathbf{x}_0$
Average of all points, excluding worst point
$\mathbf{x}_2$

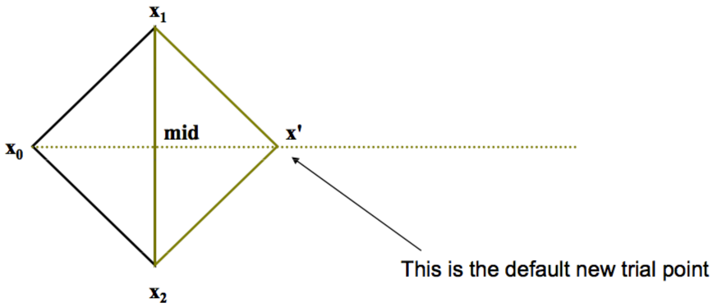
Determining the direction for optimization

```
template <class F>
void simplex615<F>::prepareUpdate() {
  for(int j=0; j < dim; ++j) {
    midPoint[j] = 0;         // average of all points but the worst point
  }
  for(int i=0; i < dim+1; ++i) {
    if ( i != idxHi ) {      // exclude the worst point
      for(int j=0; j < dim; ++j) {
        midPoint[j] += X[i][j];
      }
    }
  }
  for(int j=0; j < dim; ++j) {
    midPoint[j] /= dim;      // take average
    thruLine[j] = X[idxHi][j] - midPoint[j]; // direction for optimization
  }
}
```

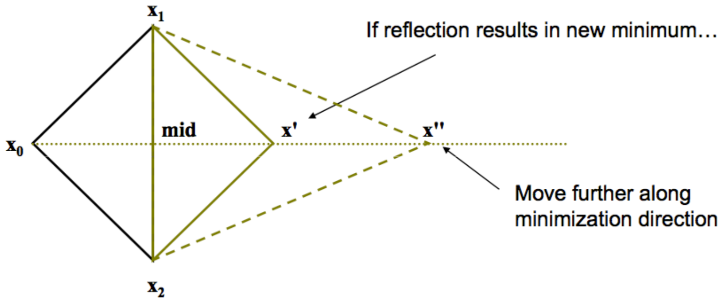
Updating simplex along the line

```
// scale determines which point to evaluate along the line
// scale = 1 : worse point, scale = 0 : midPoint
template <class F>
bool simplex615<F>::updateSimplex(F& foo, double scale) {
  std::vector<double> nextPoint;   // next point to evaluate
  nextPoint.resize(dim);
  for(int i=0; i < dim; ++i) {
    nextPoint[i] = midPoint[i] + scale * thruLine[i];
  }
  double fNext = foo(nextPoint);
  if ( fNext < Y[idxHi] ) {        // update only maximum values (if possible)
    for(int i=0; i < dim; ++i) {   //   because the order can be changed with
      X[idxHi][i] = nextPoint[i];  //   evaluateExtremes() later
    }
    Y[idxHi] = fNext;
    return true;
  }
  else {
    return false;                  // never mind if worse than the worst
  }
}
```

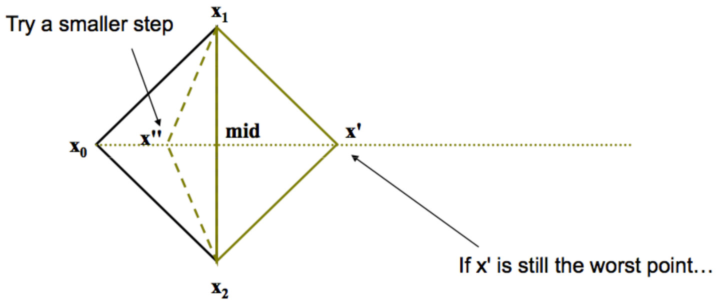
Reflection


$\mathbf{x_1}$
$\mathbf{x_0}$
mid
$\mathbf{x'}$
$\mathbf{x_2}$
This is the default new trial point

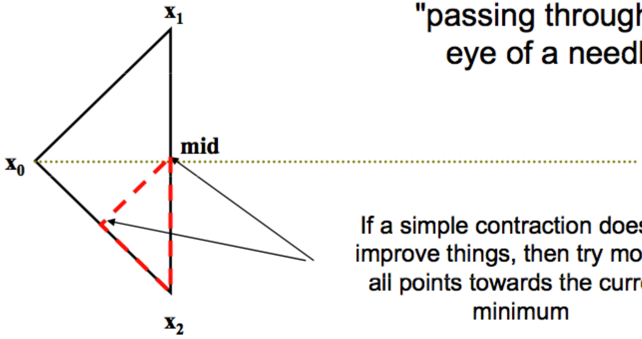
Reflection and Expansion



Contraction (1-dimension)



Multiple Contraction



"passing through the eye of a needle"

If a simple contraction doesn't improve things, then try moving all points towards the current minimum

Updating simplex along the line

```
// if none of the tried points make things better
//   reduce the search space towards the minimum point
template <class F>
void simplex615<F>::contractSimplex(F& foo) {
  for(int i=0; i < dim+1; ++i) {
    if ( i != idxLo ) {    // except for the minimum point
      for(int j=0; j < dim; ++j) {
        X[i][j] = 0.5*( X[idxLo][j] + X[i][j] ); // move the point towards minimum
      }
      Y[i] = foo(X[i]);                          // re-evaluate the function
    }
  }
}
```

Putting things together

```
template <class F>
void simplex615<F>::amoeba(F& foo, double tol) {
  evaluateFunction(foo);  // evaluate the function at the initial points
  while(true) {
    evaluateExtremes();   // determine three important points
    prepareUpdate();      // determine direction for optimization

    if ( check_tol(Y[idxHi],Y[idxLo],tol) ) break; // check convergence
    updateSimplex(foo, -1.0);      // reflection
    if ( Y[idxHi] < Y[idxLo] ) {
      updateSimplex(foo, -2.0);   // expansion
    }
    else if ( Y[idxHi] >= Y[idxNextHi] ) {
      if ( !updateSimplex(foo, 0.5) ) {  // 1-d contraction
         contractSimplex(foo);   // multiple contractions
      }
    }
  }
}
```

`amoeba()` function

- A general purpose minimization routine
  - Works in multiple dimensions
  - Uses only function evaluations
  - Does not require derivatives

Checking convergence

```
// Note that the function is declared as "static" function as
//
// static int check_tol(double fmax, double fmin, double ftol);
//
//   because it does not use any member variables
template <class F>
int simplex615<F>::check_tol(double fmax, double fmin, double ftol) {
  // calculate the difference
  double delta = fabs(fmax - fmin);
  // calculate the relative tolerance
  double accuracy = (fabs(fmax) + fabs(fmin)) * ftol;
  // check if difference is within tolerance
  return (delta < (accuracy + ZEPS));
}
```

Using the Nelder-Mead Method Implementation

```
#include <vector>
#include <cmath>
#include <iostream>
#include "simplex615.h"
#define ZEPS 1e-10

int main(int main, char** argv) {
  double point[2] = {-1.2, 1};  // initial point to start

  arbitraryFunc foo;          // WILL BE DISCUSSED LATER
  simplex615<arbitraryFunc> simplex(point, 2); // create a simplex
  simplex.amoeba(foo, 1e-7);    // optimize for a function
  // print outputs
  std::cout << "Minimum = " << simplex.ymin() << ", at ("
            << simplex.xmin()[0] << ", " << simplex.xmin()[1]
            << ")" << std::endl;
  return 0;
}
```

Defining arbitraryFunc

```
// function object used as an argument
class arbitraryFunc {
 public:
  double operator() (std::vector<double>& x) {
    // f(x0,x1) = 100*(x1-x0^2)^2 + (1-x0)^2
    return 100*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1-x[0])*(1-x[0]);
  }
};
```

A working example

```
Minimum = 1.35567e-11, at (0.999999, 0.999997)
```

Normal Density

Normal density function

$$f(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

Implementation

```
class NormMix615 {
public:
  static double dnorm(double x, double mu, double sigma) {
    return 1.0 / (sigma * sqrt(M_PI * 2.0)) *
           exp (-0.5 * (x - mu) * (x-mu) / sigma / sigma);
  }
  ...
};
```

Gaussian mixture distribution

Density function

$$p(x|k, \pi, \mu, \sigma) = \sum_{i=1}^{k} \pi_i f(x|\mu_i, \sigma_i)$$

Implementation (within NormMix615)

```
static double dmix(double x, std::vector<double>& pis,
                   std::vector<double>& means, std::vector<double>& sigmas) {
  double density = 0;
  for(int i=0; i < (int)pis.size(); ++i)
    density += pis[i] * dnorm(x,means[i],sigmas[i]);
  return density;
```

Likelihood of multiple observations

Calculating in log-space

$$L = \prod_i p(x_i|\pi, \pi, \mu, \sigma)$$

$$l = \sum_i \log p(x_i|\pi, \mu, \sigma)$$

Implementation (within `NormMix615`)

```cpp
static double mixLLK(std::vector<double>& xs, std::vector<double>& pis,
                     std::vector<double>& means, std::vector<double>& sigmas) {
  int i=0;
  double llk = 0.0;
  for(int i=0; i < xs.size(); ++i)
    llk += log(dmix(xs[i], pis, means, sigmas));
  return llk;
}
```

Gaussian Mixture Function Object

```
class NormMix615 {
public:  // these are internal function
  static double dnorm(double x, double mu, double sigma);
  static double dmix(...);
  static double mixLLK(...);
};
class LLKNormMixFunc {
public:     // below are public functions
  LLKNormMixFunc(int k, std::vector<double>& y) :
      numComponents(k), data(y), numFunctionCalls(0) {}
  // core function - called when foo() is used
  // x is the combined list of MLE parameters (pis, means, sigmas)
  double operator() (std::vector<double>& x);
  void assignPriors(std::vector<double>& x, std::vector<double>& priors);
  std::vector<double> data;
  int numComponents;
  int numFunctionCalls;
};
```

Avoiding boundary conditions

Problem

- The simplex algorithm do not know that $0 \leq \pi_i \leq 1$, and $\sum_{i=1}^{n} \pi_i = 1$
- During the iteration of simplex algorithm, it is possible that π_i goes out of bound

Possible solutions

- Modify simplex algorithm to avoid boundary conditions
- Transform the parameter space to infinite ranges

Transforming the parameter space

Constraints

- $0 \leq \pi_i \leq 1$
- $\sum_{i=1}^{n} \pi_i = 1$

Mapping between the space

- Given $x \in \mathbb{R}^{n-1}$, for $i = 1, \cdots, n-1$
- $\pi_i = \frac{1}{1+e^{-x_i}}(1 - \sum_{j=1}^{i-1} \pi_j)$
- $\pi_n = 1 - \sum_{i=1}^{n-1} \pi_i$.

Q: how about σ?

Implementing likelihood of data

```
double LLKNormMixFunc::operator() (std::vector<double>& x) { // x has (3*k-1) dims
  std::vector<double> priors;
  std::vector<double> means;
  std::vector<double> sigmas;
  // transform (k-1) real numbers to priors
  assignPriors(x, priors);
  for(int i=0; i < numComponents; ++i) {
    means.push_back(x[numComponents-1+i]);
    sigmas.push_back(x[2*numComponents-1+i]);
  }
  return 0-NormMix615::mixLLK(data, priors, means, sigmas);
}
```

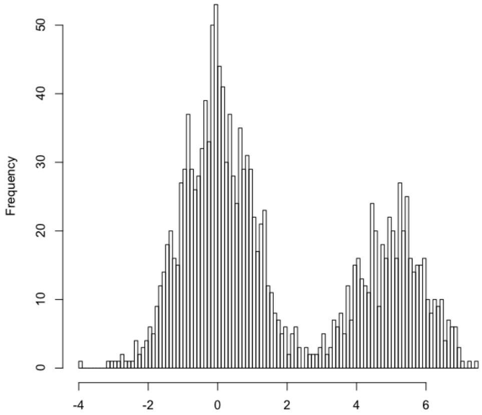
Transforming between bounded and unbounded space of priors

```
void LLKNormMixFunc::assignPriors(std::vector<double>& x, std::vector<double>& priors) 
  priors.clear();
  // convert priors (from [k-1]-d real scale to [k]-d simplex scale)
  double p = 1.;
  for(int i=0; i < numComponents-1; ++i) {
    double logit = 1./(1.+exp(0-x[i]));
    priors.push_back(p*logit);
    p = p*(1.-logit);
  }
  priors.push_back(p);
}
```

Nelder-Mead Method for Gaussian Mixture

```cpp
#include <iostream>
#include <fstream>
#include "simplex615.h"
#include "normMix615.h"
#include "llkNormMixFunc.h"
#define ZEPS 1e-10
int main(int main, char** argv) {
  double point[5] = {0, -1, 1, 1, 1};   // 50:50 mixture of N(-1,1) and N(1,1)
  simplex615<LLKNormMixFunc> simplex(point, 5);
  std::vector<double> data;                  // input data
  std::ifstream file(argv[1]);               // open file
  double tok;                                // temporary variable
  while(file >> tok) data.push_back(tok);   // read data from file
  LLKNormMixFunc foo(2, data);               // 2-dimensional mixture model
  simplex.amoeba(foo, 1e-7);                 // run the Nelder-Mead Method
  std::cout << "Minimum = " << simplex.ymin() << ", at pi = "
            << (1./(1.+exp(0-simplex.xmin()[0]))) << "," << "between N("
            << simplex.xmin()[1] << "," << simplex.xmin()[3] << ") and N("
            << simplex.xmin()[2] << "," << simplex.xmin()[4] << ")" << std::endl;
  return 0;
}
```

A working example



A working example

Simulation of data

```
> x <- rnorm(1000)
> y <- rnorm(500)+5
> write.table(matrix(c(x,y),1500,1),'mix.dat',row.names=F,col.names=F)
```

A Running Example

```
Minimum = 3043.46, at pi = 0.667271,
between N(-0.0304604,1.00326) and N(5.01226,0.956009)
```