

# Biostatistics 615/815 Lecture 19: Multidimensional Optimizations

Hyun Min Kang

March 29th, 2011

# Announcements

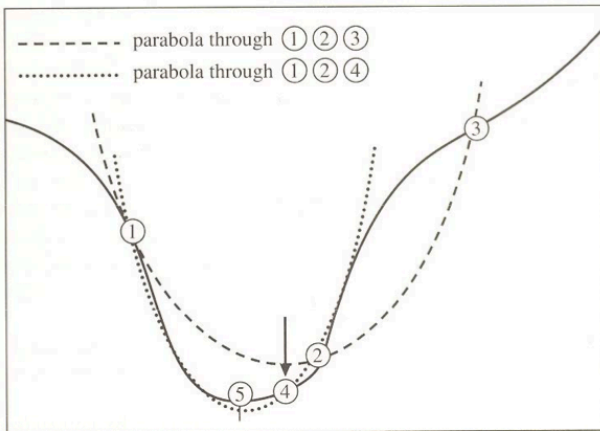
## Homework

- Homework #5 due today
- Extension to thursday is allowed

## Today's lecture

- The Simplex Method Details
- MLE estimation of mixture of normals

# Recap : Single-dimensional minimization using parabola



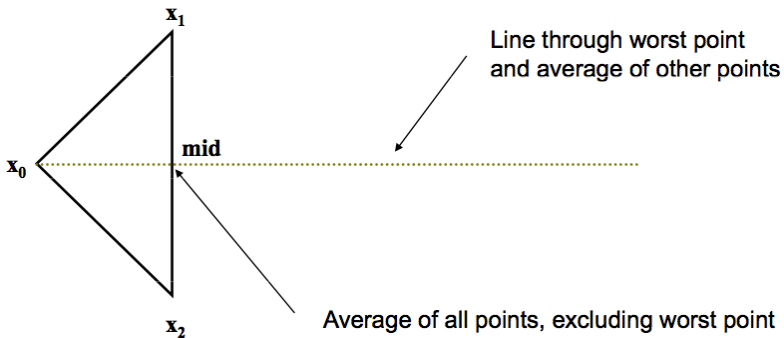
# Recap : Adaptive Minimization

- 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

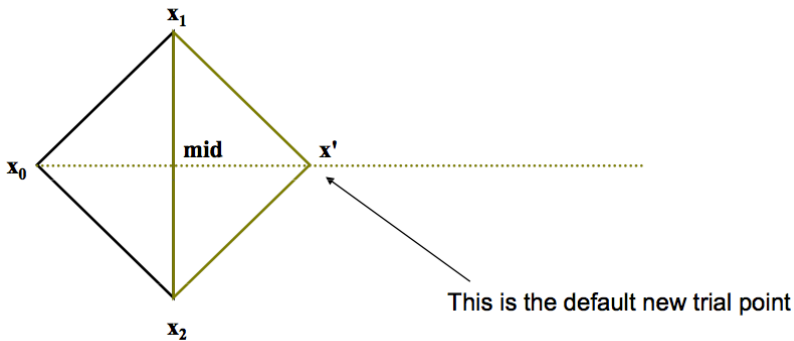
# The Simplex Method

- Calculate likelihoods at simplex vertices
  - Geometric shape with  $k + 1$  corners
  - A triangle in  $k = 2$  dimensions
- Simplex *crawls*
  - Towards minimum
  - Away from maximum
- Probably the most widely used optimization method

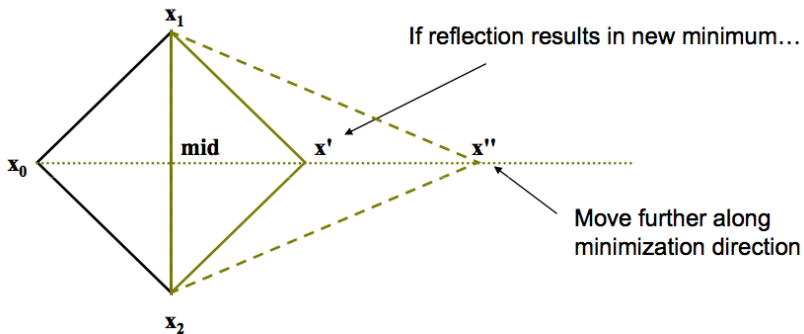
# Direction for Optimization



# Reflection

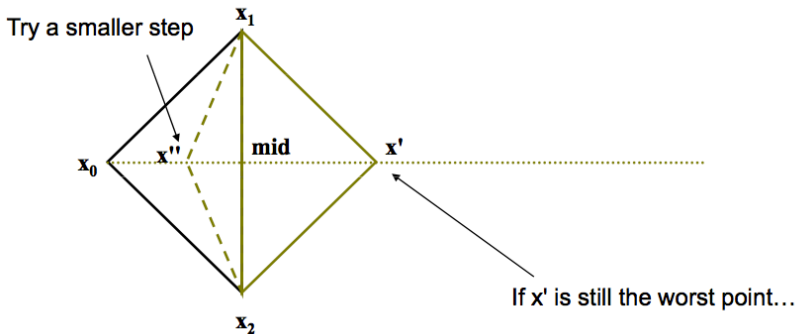


# Reflection and Expansion

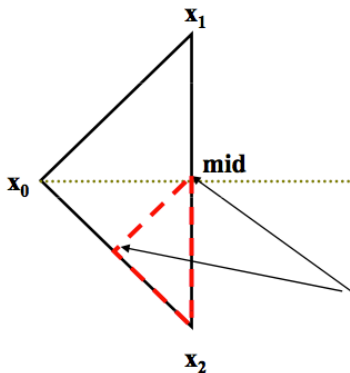




# Contraction (1-dimension)



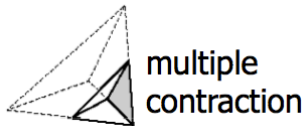
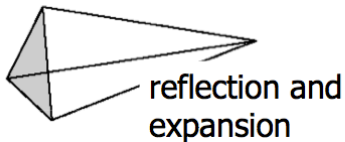
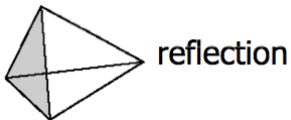
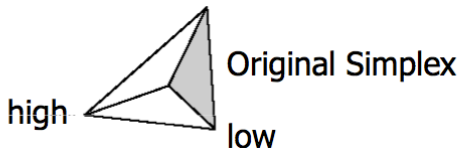
# Contraction



"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 Simplex Method



# Implementing the Simplex Method

```
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(optFunc& foo); // evaluate function value at each point
    void evaluateExtremes(); // determine the min, max, 2ndmax
    void prepareUpdate(); // calculate midPoint, thruLine
    bool updateSimplex(optFunc& foo, double scale); // for reflection/expansion..
    void contractSimplex(optFunc& 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(optFunc& 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 worse 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 worse 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

```

// constructor of simplex615 class : initial point is given
simplex615::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

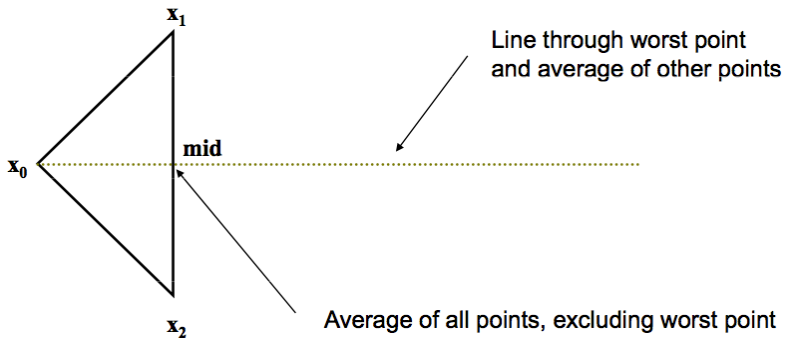
```
// simple function for evaluating the function value at each simple point
// after calling this function Y[i] = foo(X[i]) should hold
void simplex615::evaluateFunction(optFunc& foo) {
    for(int i=0; i < dim+1; ++i) {
        Y[i] = foo(X[i]); // foo is a function object, which will be visited later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



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

```
void simplex615::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



# Determining the direction for optimization

```
void simplex615::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
    }
}
```

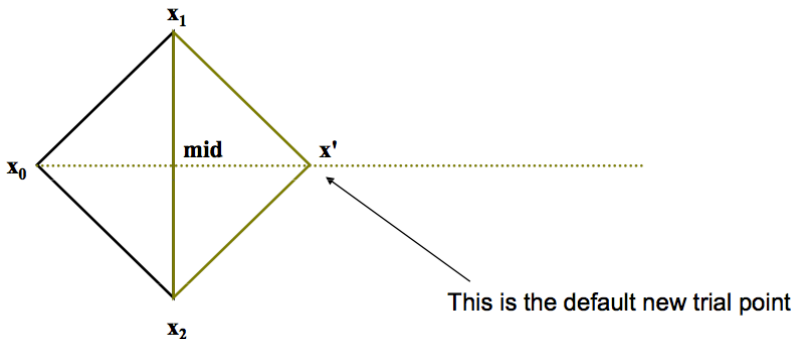
# Updating simplex along the line

```

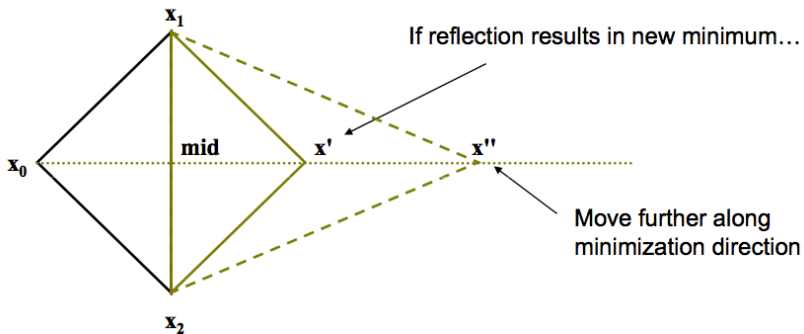
// scale determines which point to evaluate along the line
// scale = 1 : worse point, scale = 0 : midPoint
bool simplex615::updateSimplex(optFunc& foo, double scale) {
    std::vector<double> nextPoint;    // next point to evaluate
    nextPoint.resize(dim);
    for(int i=0; i < dim; ++i) {
        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
    }
}

```

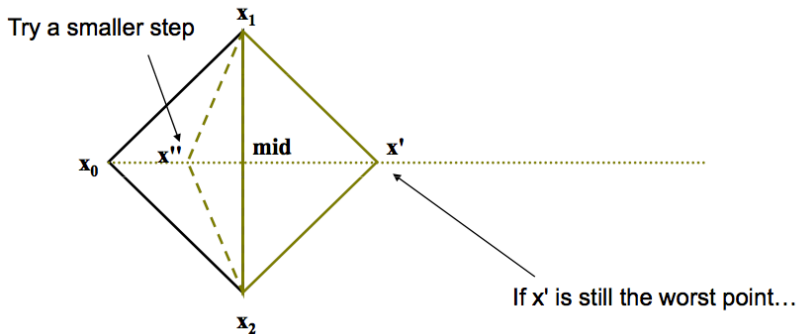
# Reflection



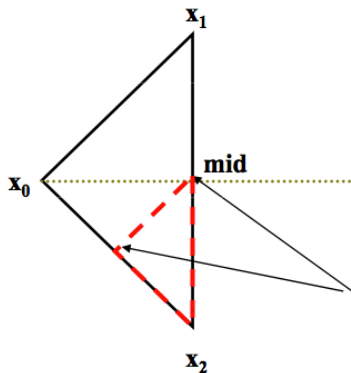
# Reflection and Expansion



# 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
void simplex615::contractSimplex(optFunc& 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

```
void simplex615::amoeba(optFunc& 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
int simplex615::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 Simplex 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

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

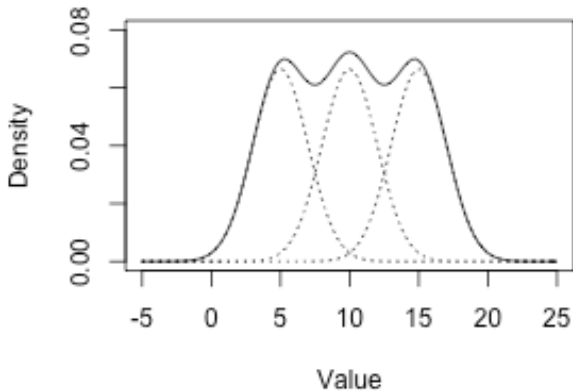
# Defining a function using inheritance

```
// this is an abstract base class, which CAN NOT be used as class instance
class optFunc {
public:
    // 'virtual' means inherited method can be used when
    // optFunc class is used via pointer or reference
    virtual double operator() (std::vector<double>& x) = 0; // function disabled
};
// Define a function inherits the function
// when foo() is called at the simplex, this function is actually called
class arbitraryOptFunc : public optFunc {
public:
    virtual double operator() (std::vector<double>& x) {
        //  $100*(x_1-x_0^2)^2 + (1-x_0)^2$ 
        return 100*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1-x[0])*(1-x[0]);
    }
};
```

# A working example

Minimim =  $1.35567e-11$ , at  $(0.999999, 0.999997)$

# Recap : A mixture distribution





# Recap : MLE in Gaussian mixture

## Parameter estimation in Gaussian mixture

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

## Possible approaches

- Simplex Method
- Expectation Maximization
- Markov-Chain Monte Carlo

# 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 mixLLKFunc : public optFunc {
protected:
    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

```
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

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

## Implementation

```
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 mixLLKFunc : public optFunc {
protected: // these are internal function
    static double dnorm(double x, double mu, double sigma);
    static double dmix(...);
    static double mixLLK(...);
public: // below are public functions
    mixLLKFunc(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)
    virtual double operator() (std::vector<double>& x);
    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  $\pi_i$  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  $i = 1, \dots, n-1$
- $\pi_i = \frac{1}{1+e^{-x_i}} (1 - \sum_{j=1}^{i-1} \pi_j)$
- $\pi_n = 1 - \sum_{i=1}^{n-1} \pi_i.$

# Implementing likelihood of data

```

virtual double 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
    double p = 1.;
    for(int i=0; i < numComponents-1; ++i) {
        double logit = 1./(1.+exp(θ-x[i]));
        priors.push_back(p*logit);
        p = p*(1.-logit);
    }
    priors.push_back(p);
    for(int i=0; i < numComponents; ++i) {
        means.push_back(x[numComponents-1+i]);
        sigmas.push_back(x[2*numComponents-1+i]);
    }
    return θ-mixLLK(data, priors, means, sigmas);
}

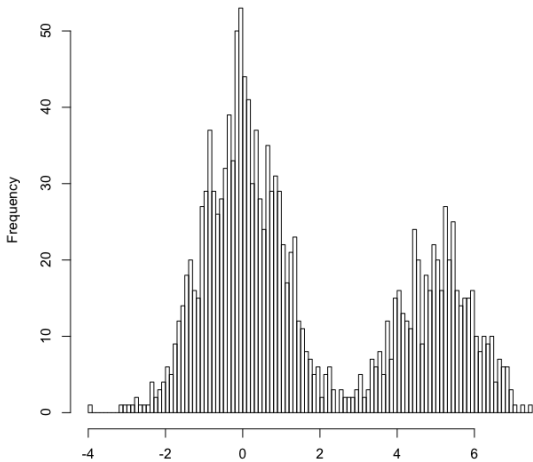
```



# Simplex Method for Gaussian Mixture

```
#include <iostream>
#include <fstream>
#include "simplex615.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 simplex(point, 5);
    std::vector<double> data; // input data
    std::ifstream file(argv[1]); // open file
    double tok; // temporaru variable
    while(file >> tok) data.push_back(tok); // read data from file
    mixLLKFunc foo(2, data); // 2-dimensional mixture model
    simplex.amoeba(foo, 1e-7); // run the Simplex Method
    std::cout << "Minimim = " << simplex.ymin() << ", at pi = "
        << (1./(1.+exp(θ-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

```
Minimim = 3043.46, at pi = 0.667271,
between N(-0.0304604,1.00326) and N(5.01226,0.956009)
(305 function evaluations in total)
```

# Summary

## Today

- Implementation of the Simplex Method
- Application to mixture of normal distributions

## Recommended Readings

- Numerical recipes 10.5 - clear description of simplex method
- Subsequent sections contains more sophisticated multivariate normal distribution

## Next Lecture

- The Expectation-Maximization Algorithm