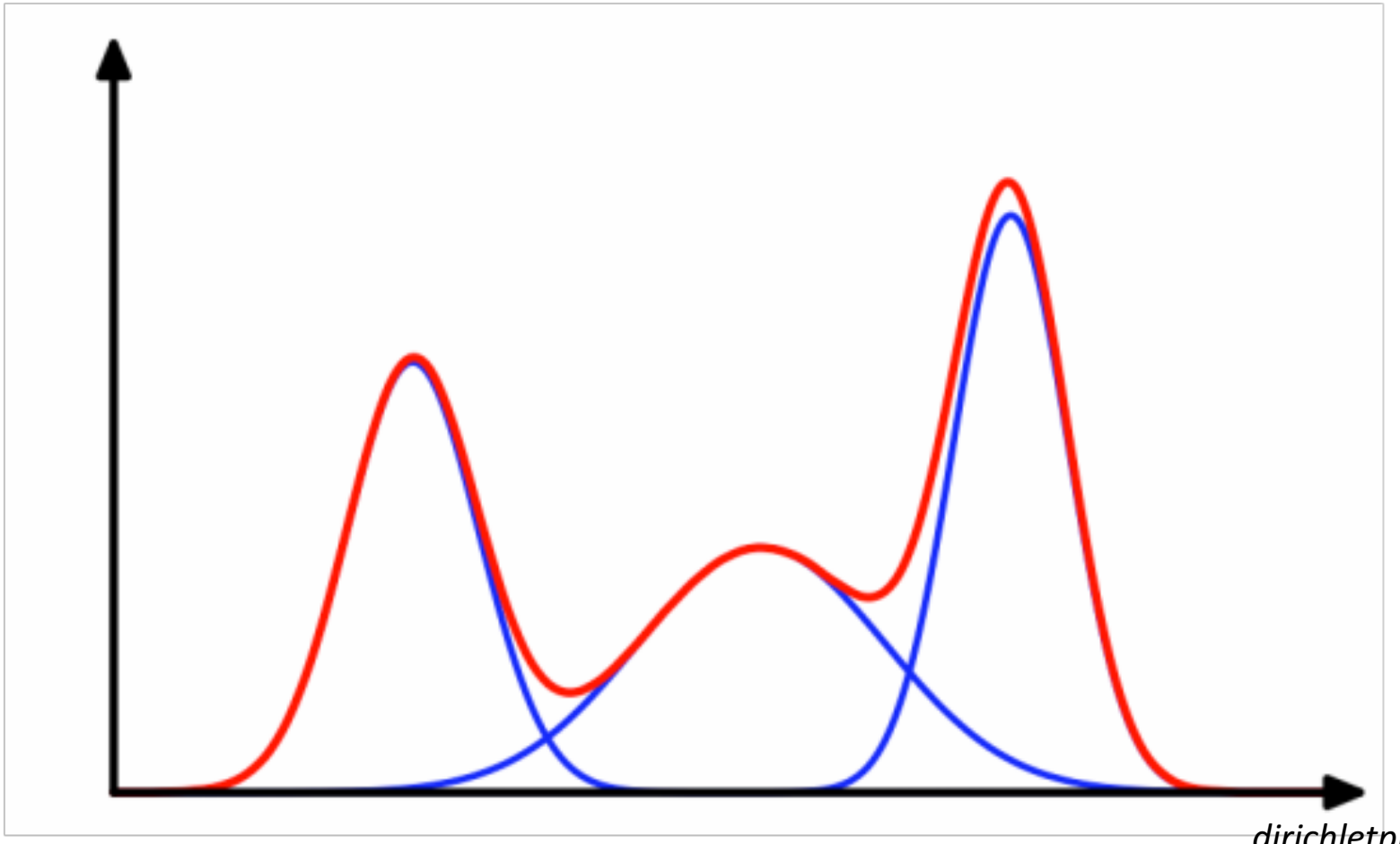
MODULE 2.5

MULTI-DIMENSIONAL OPTIMIZATION

Gaussian mixture model



Credit: dirichletprocess.weebly.com

A general mixture distribution

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

- $\boldsymbol{x}$: observed data
- $\boldsymbol{\pi}$: mixture proportion of each component
- $\boldsymbol{\phi}$: parameters specific to each component
- $\boldsymbol{\eta}$: parameters shared across the components
- f : probability density function of each component
- p : probability density function of mixture distribution

Maximum likelihood estimation

- **Finding maximum likelihood**

$$L = \prod_{i=1}^{n} p(x_i; \boldsymbol{\pi}, \boldsymbol{\phi}, \boldsymbol{\eta})$$

$$l = \log L = \sum_{i=1}^{n} p(x_i; \boldsymbol{\pi}, \boldsymbol{\phi}, \boldsymbol{\eta})$$

$$(\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\phi}}, \hat{\boldsymbol{\eta}}) = \underset{\boldsymbol{\pi},\boldsymbol{\phi},\boldsymbol{\eta}}{\operatorname{argmax}}\, l = \underset{\boldsymbol{\pi},\boldsymbol{\phi},\boldsymbol{\eta}}{\operatorname{argmax}} \sum_{i=1}^{n} p(x_i; \boldsymbol{\pi}, \boldsymbol{\phi}, \boldsymbol{\eta})$$

Single component Gaussian MLE

$$p(\boldsymbol{x}; \mu, \sigma^2) = \prod_{i=1}^{n} f(x_i; \mu, \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$$

- **Given x, what is the MLE parameters of μ, σ^2?**
- **Closed-form solutions exist.**

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \hat{\mu})^2$$

MLE in Gaussian mixture

- **Problem**
  - Need to estimate (up to) 3k-1 parameters
    - Mixing proportions $\pi_1, \pi_2, \cdots, \pi_k \quad (\pi_1 + \pi_2 + \cdots + \pi_k = 1)$
    - Means $\mu_1, \mu_2, \cdots, \mu_k$
    - Variances $\sigma_1^2, \sigma_2^2, \cdots, \sigma_k^2$

- **No analytical solution exists**

- **Numerical optimization required over multiple parameters.**

The Nelder-Mead Method

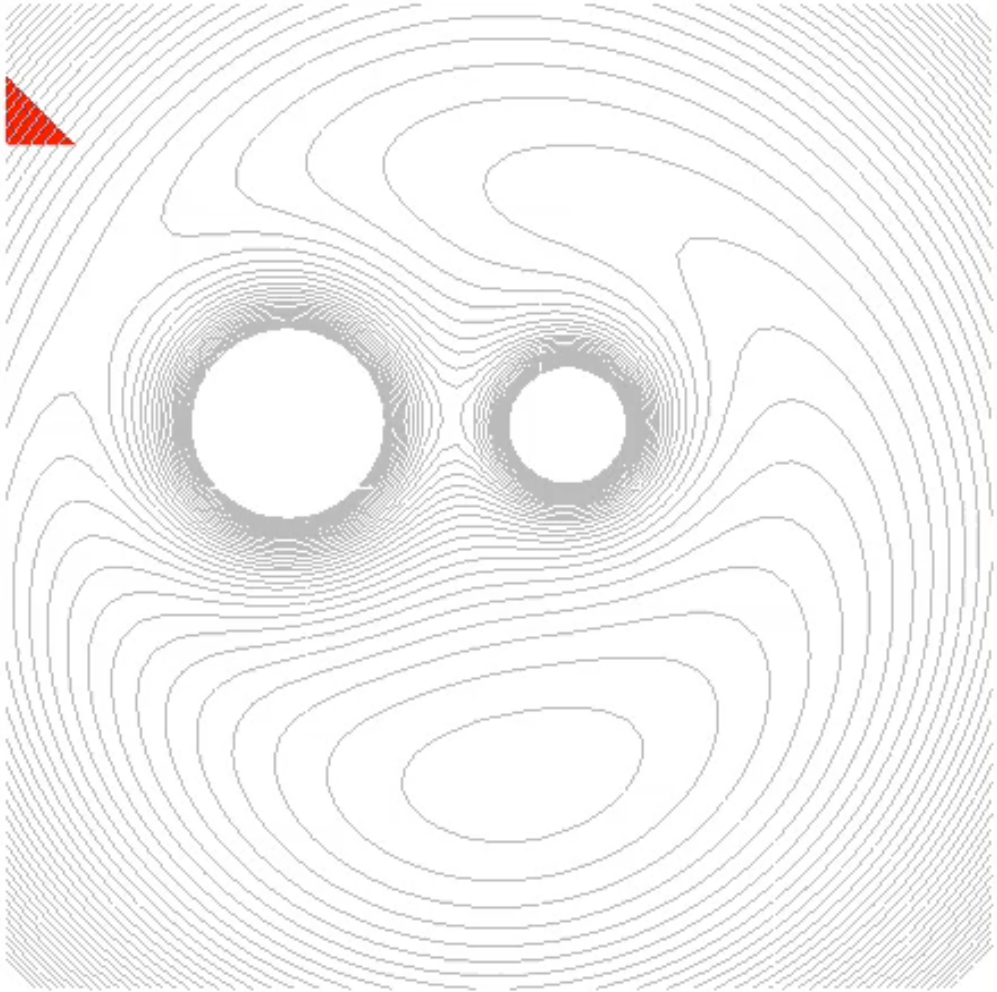
- **Also called simplex method or amoeba method**

- **Calculate likelihoods at simplex vertices**
  - Geometric shape with k+1 corners
  - A triangle when k=2
  - A tetrahedron when k=3

- **Use simplex *crawls* towards minimum.**

- **A commonly used non-linear method without derivative**
  - No guarantee that it will converge to local minimum.

Illustration of Nelder-Mead *(k=2)*



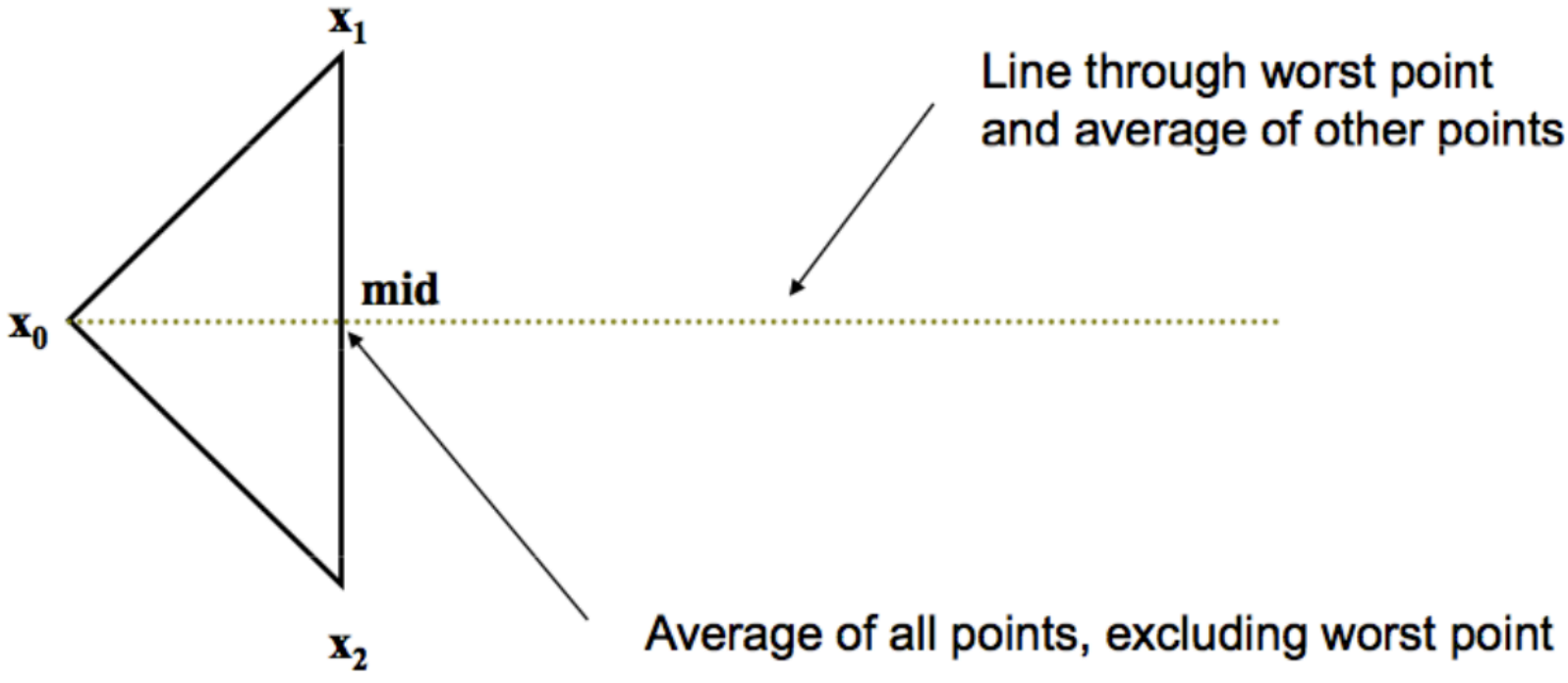
Source :
https://userpages.umbc.edu/~rostamia/2013-01-math625/images/nelder-mead.gif

Nelder-Mead method (in k=2)

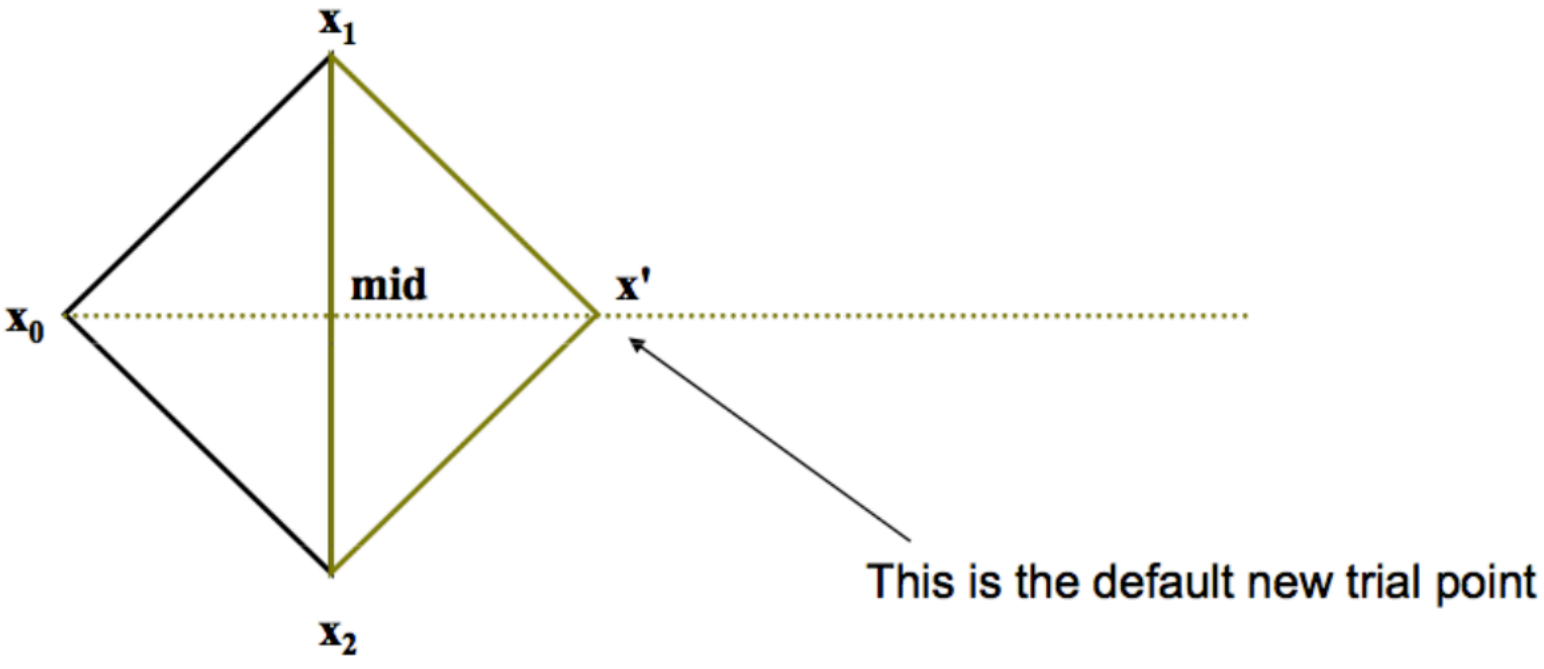
- **Evaluate functions at three vertices**
  - x_0 : The highest (worst) point
  - x_1 : The next highest point
  - x_2 : The lowest (best) point

- **Intuition for choosing a new point**
  - Move away from high point
  - Move towards low point

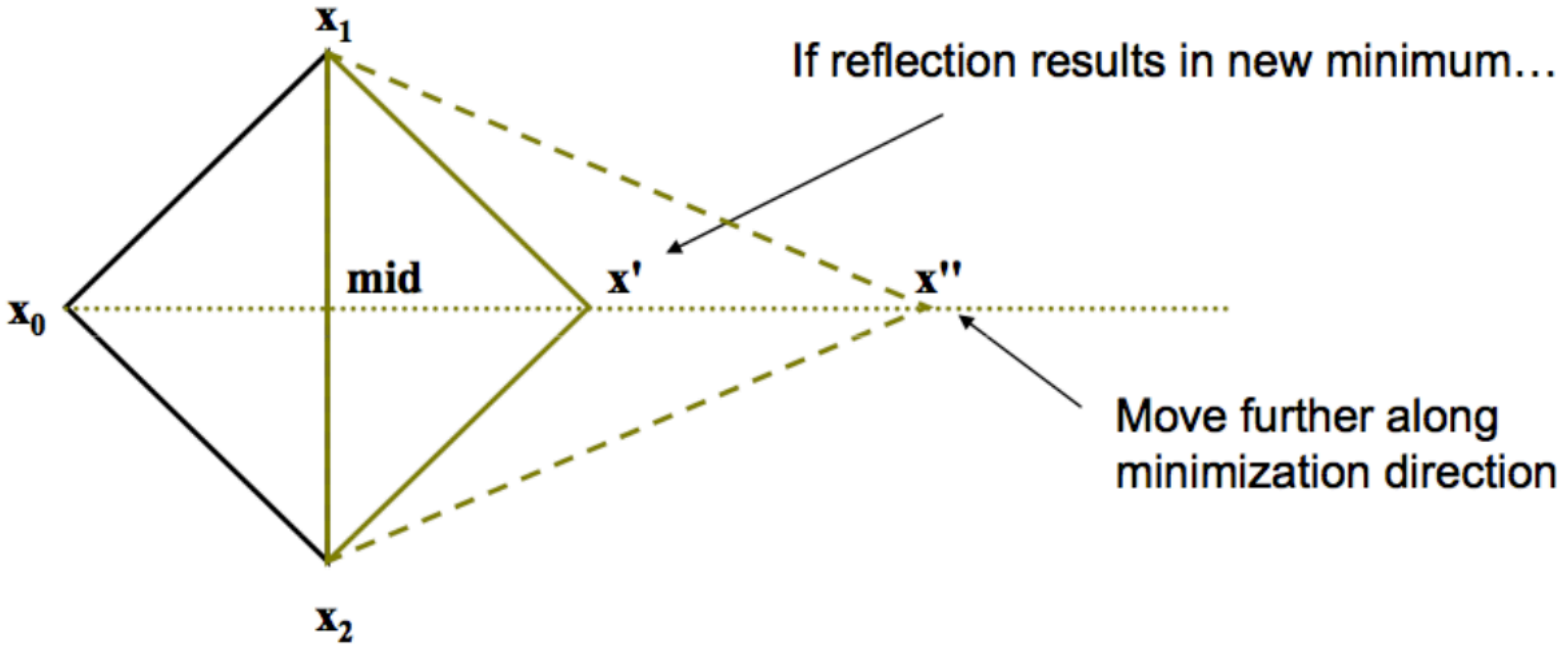
Direction for optimization



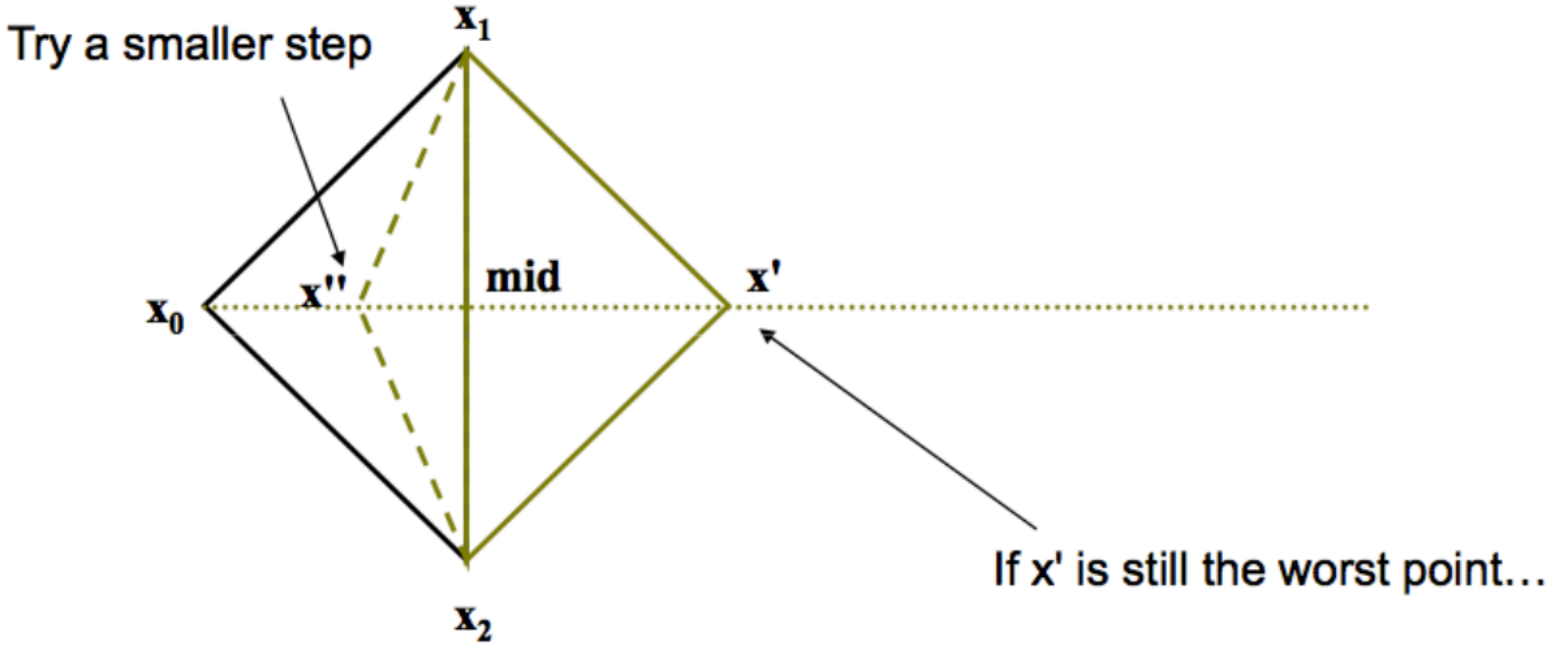
Possible move : Reflection



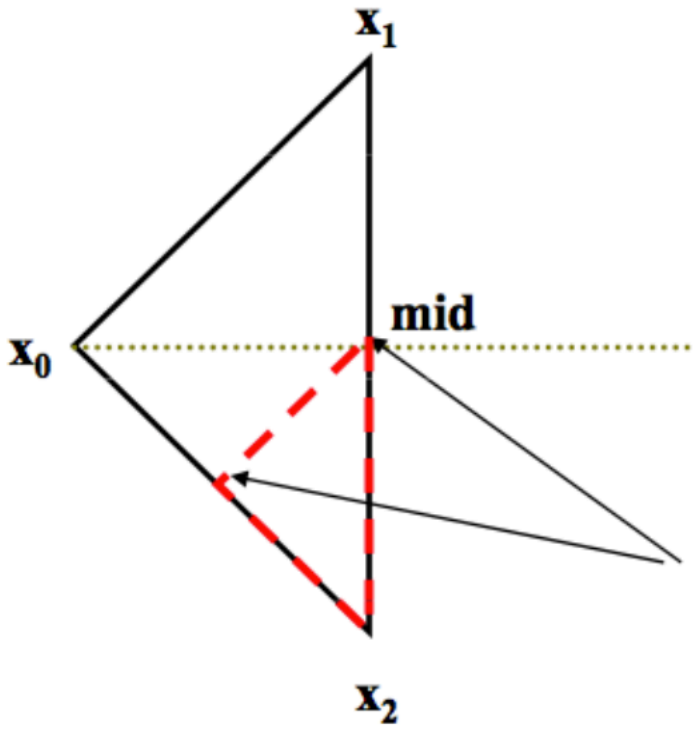
Possible move : Reflection & Expansion



Possible move : 1d-Contraction



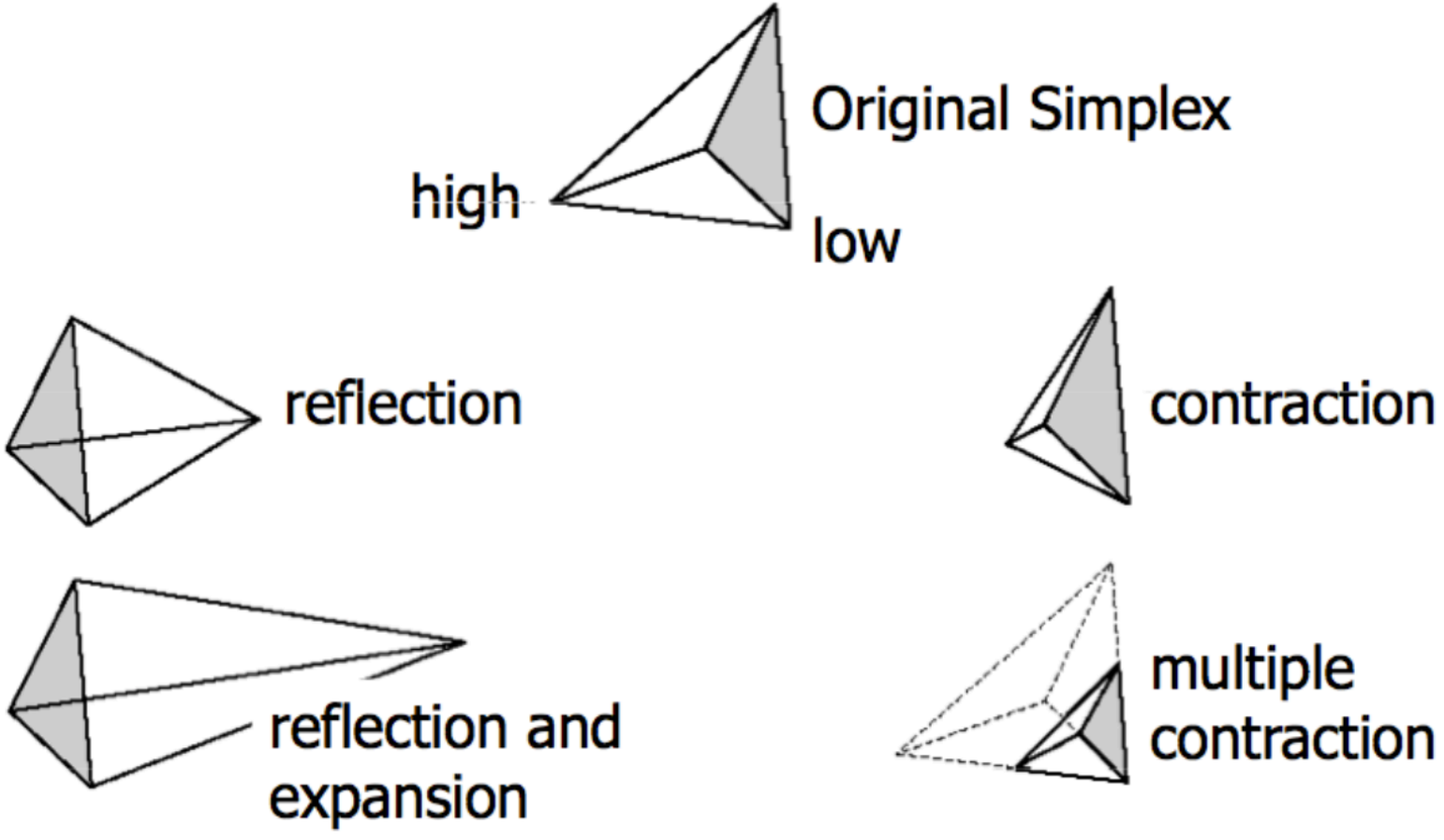
Possible move : Multiple Contraction



"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 of possible moves (k=3)



H

Nelder-Mead algorithm

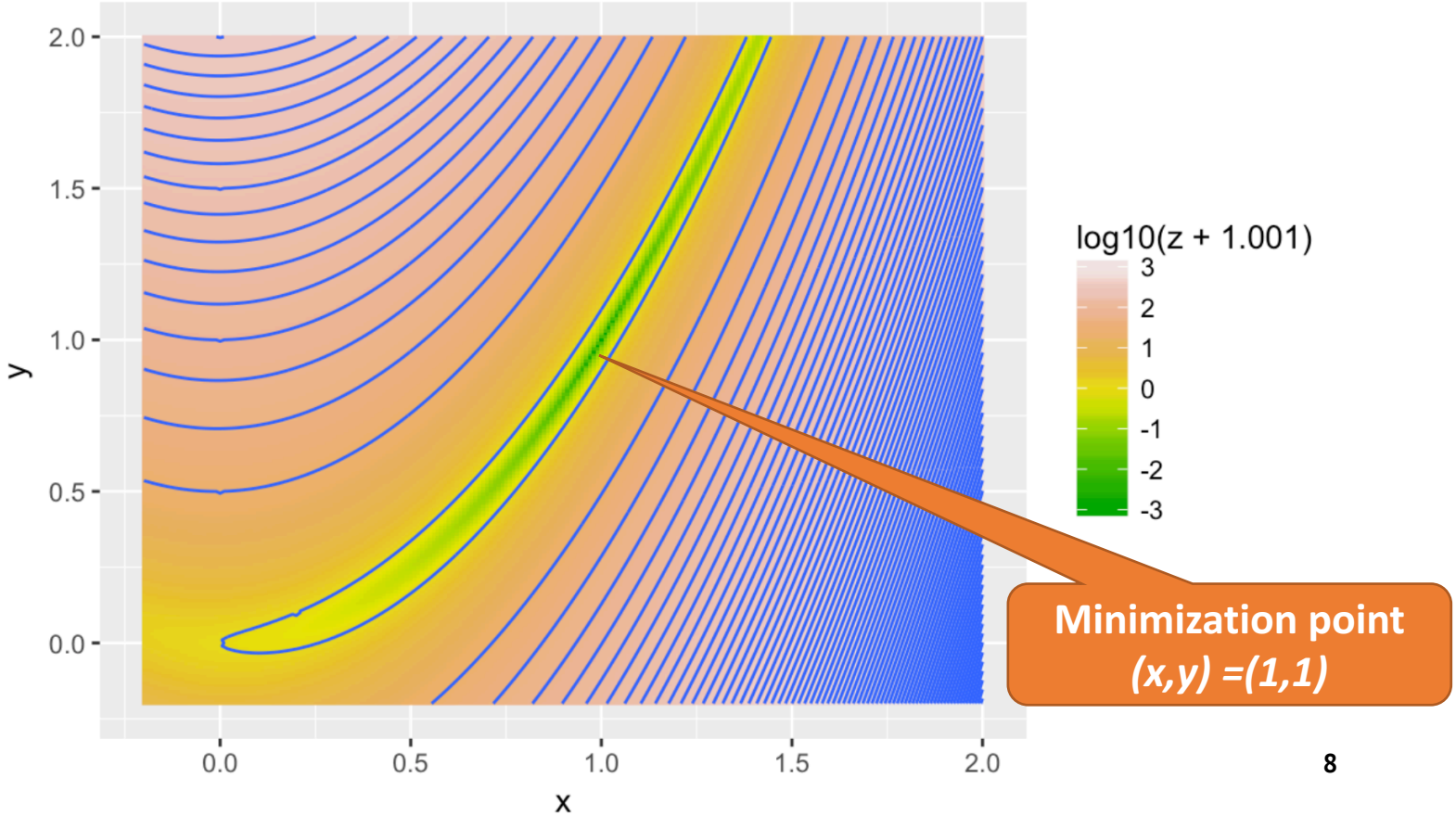
1. **Pick starting points**
2. **Evaluate the values are the starting points**
3. **Determine the direction of optimization**
4. **Reflect on the worse point and update the simplex**
5. **If reflection point beats the current best point,**
   - Expand the reflection point and update the simplex
6. **If reflection point is worst than the 2nd best point,**
   - Perform 1d-contraction and update the simplex
   - If contraction didn't do anything, then perform multiple contraction
7. **Repeat 3-6 until convergence**

Nelder-Mead implementation in R

- In R, there is no need to reinvent the wheel for Nelder-Mead
- The `optim()` function provides a wide range of optimization methods, including Nelder-Mead and Brent.
- The usage is
  `optim(start, func, method="Nelder-Mead")`
- See `help(optim)` for details

Problem

- **An arbitrary 2d function** $f(x,y) = 100(y-x^2)^2+(1-x)^2-1$



Results from `optim()` function

```
## x is size 2 vector, foo is an arbitrary vector
foo <- function(x) {
  return(100*(x[2]-x[1]*x[1])*(x[2]-x[1]*x[1])+(1-x[1])*(1-x[1]))-1
}
```

```
print(optim(c(0,1.8),foo,"Nelder-Mead"))
```

```
$par
[1] 0.9996117 0.9991492

$value
[1] -0.9999993

$counts
function gradient
      55       NA

$convergence
[1] 0
```

```
print(optim(c(0.9,1.1),foo,"Nelder-Mead"))
```

```
$par
[1] 1.000003 1.000021

$value
[1] -1

$counts
function gradient
      75       NA

$convergence
[1] 0
```

utin

Nelder-Mead in Rcpp

```
#include <Rcpp.h>

using namespace std;
using namespace Rcpp;

// this is an example function object
class myFunc {
public:
  double operator() (const NumericVector& x) {
    return(100*(x[1]-x[0]*x[0])*(x[1]-x[0]*x[0])+(1-x[0])*(1-x[0])-1);
  }
};
```

Setting up a class for Nelder-Mead algorithm

```
// template allows class F to replace arbitrary function object
template <class F>
class simplex615 { // contains (dim+1)
  // member variables;
  int dim;           // dimension of data
  NumericMatrix X;  // (ndims+1) simplex points of ndims points
  NumericVector Y;  // (ndims+1) evaluated point
  NumericVector midPoint; // midpoint between ndims best points
  NumericVector thruLine; // line starting from worst point thru midPoint
  int idxLo, idxHi, idxNextHi, iter; // additional variables

  // check whether the values are within tolerance
  bool check_tol(double fmax, double fmin, double ftol) {
    double delta = fabs(fmax - fmin);
    double accuracy = (fabs(fmax) + fabs(fmin)) * ftol;
    return (delta < (accuracy + 1e-15));
  }
```

Initializing the simplex

```
// Constructor, initializing member variables
simplex615(NumericVector& startPoint, double stepSize = 1.0) :
  dim(startPoint.size()),
  X(dim+1, dim),
  Y(dim+1),
  midPoint(dim),
  thruLine(dim)
{
  // initialize all X to startPoint;
  int dim = startPoint.size();
  for(int i=0; i < dim+1; ++i) {
    for(int j=0; j < dim; ++j) {
      X(i,j) = startPoint[j];
      if ( i == j ) X(i,j) += stepSize; // create a simplex
    }
  }
}
```

How the Nelder-Mead algorithm works

```
// main function to run the Nelder-Mead algorithm
void amoeba(F& foo, double tol) {
  iter = 0; // reset counter
  evaluateFunction(foo); // evaluate all points
  while(true) {          // repeat until convergence
    evaluateExtremes();  // set lo/hi/nexthi
    prepareUpdate();     // compute midPoint, thruLine
    // stop if below tolerance
    if ( check_tol(Y[idxHi],Y[idxLo],tol) ) break;
    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) ) // 1d-contraction
        contractSimplex(foo);         // multiple contraction
    }
  }
}
```

Evaluating function values

```
// evaluate all function points at the beginning
void evaluateFunction(F& foo) {
  for(int i=0; i < dim+1; ++i) {
    Y[i] = foo(X(i,_));
    ++iter;
  }
}
```

```
// evaluate extreme values determine lo/hi/nextHi indices
void evaluateExtremes() {
  if ( Y[0] > Y[1] ) {
    idxHi = 0; idxLo = idxNextHi = 1;
  }
  else {
    idxHi = 1; idxLo = idxNextHi = 0;
  }
  // make idxLo to point the lowest point,
  // make idxHi to the highest, nextHi to second highest.
  for(int i=2; i < dim+1; ++i) {
    if ( Y[i] <= Y[idxLo] ) idxLo = i;
    else if ( Y[i] > Y[idxHi] ) {
      idxNextHi = idxHi;
      idxHi = i;
    }
    else if ( Y[i] > Y[idxNextHi] ) idxNextHi = i;
  }
}
```

Calculate the direction of next moves

```
// calculate midPoint and thruLine
void prepareUpdate() {
  std::fill(midPoint.begin(), midPoint.end(), 0);
  for(int i=0; i < dim+1; ++i) {
    if ( i != idxHi ) { // exclude the worst
      for(int j=0; j < dim; ++j)
        midPoint[j] += X(i,j);
    }
  }
  for(int j=0; j < dim; ++j) {
    midPoint[j] /= dim;    // take average
    // direction for optimization
    thruLine[j] = X(idxHi,j) - midPoint[j];
  }
}
```

Function for basic simplex operation

```
// evaluate next point and update lo/hi/nexthi indices
bool updateSimplex(F& foo, double scale) {
  NumericVector nextPoint(dim); // set next point to evaluate
  for(int i=0; i < dim; ++i)
    nextPoint[i] = midPoint[i] + scale * thruLine[i];
  // evaluate the function
  double fNext = foo(nextPoint); ++iter;
  if ( fNext < Y[idxHi] ) {      // exchange with worst point
    for(int i=0; i < dim; ++i)
      X(idxHi,i) = nextPoint[i];
    Y[idxHi] = fNext;
    return true;  // if updated, return true
  }
  else {
    return false;
  }
}
```

Multiple contraction

```
// perform multiple contraction
void contractSimplex(F& foo) {
  for(int i=0; i < dim+1; ++i) {
    if ( i != idxLo ) { // update every point but the best
      for(int j=0; j < dim; ++j) {
        X(i,j) = 0.5*( X(idxLo,j) + X(i,j) );
        Y[i] = foo(X[i]); iter++;
      }
    }
  }
}
```

Returning other key values

```
// return key values
NumericVector xmin() {  return X(idxLo,_); }
double ymin() { return Y[idxLo]; }
int niter() { return iter; }
```

Using Nelder-Mead algorithm

```
// [[Rcpp::export]]
List runNelderMead(NumericVector startPoint, double initStep, double tol) {
  myFunc foo; // make an instance of function object
  simplex615<myFunc> simplex(startPoint, initStep); // set parameters
  simplex.amoeba(foo, tol);   // run simplex method
  return( List::create(Named("ymin")=simplex.ymin(),
                       Named("xmin")=simplex.xmin(),
                       Named("iter")=simplex.niter()
                       ));
}
```

Example run

```
runNelderMead(c(0,1.8), 1.0, 1e-6)
```

```
$ymin
[1] -0.9999983

$xmin
[1] 0.9989365 0.9977975

$iter
[1] 59
```

Varying start point

```
runNelderMead(c(0.9,1.1), 1.0, 1e-6)
```

```
$ymin
[1] -0.9999995

$xmin
[1] 1.000621 1.001216

$iter
[1] 68
```

Varying step size

```
runNelderMead(c(0.9,1.1), 0.15, 1e-6)
```

```
$ymin
[1] -0.9999995

$xmin
[1] 1.000680 1.001368

$iter
[1] 53
```

Applying to a mixture of normal distribution

- **We can still use the `simplex615` class using a different function**

- **Need to define a new function object**
  - Initially take the observed data and number of parameters.
  - Function object calculates the log-likelihood given the parameters
  - Return negative of log-likelihood to make it as minimization problem

- **How should we define the function?**

Connecting equations and implementations

- What we want to calculate

$$p\left(\boldsymbol{x};\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\sigma^2}\right)=\prod_{i=1}^{n}\left[\sum_{j=1}^{k}\pi_j f\left(x_i;\mu,\sigma^2\right)\right]$$

$$l\left(\boldsymbol{x};\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\sigma^2}\right)=\sum_{i=0}^{n}\log\left[\sum_{j=1}^{k}\pi_j f\left(x_i;\mu,\sigma^2\right)\right]$$

- What we need to implement (to use `simplex615`)

```
double operator() (const NumericVector& x)
```

R implementation of the mixture likelihood

```
## x is observed data
## pis, mus, sds are k-dimensional vectors
llk.gmm <- function(x, pis, mus, sds) {
  ## make x to be n x k matrix
  k <- length(pis)
  n <- length(x)
  llks <- rowSums(
            matrix(pis, n, k, byrow=TRUE) *
              dnorm( matrix(x, n, k),
                     matrix(mus, n, k, byrow=TRUE),
                     matrix(sds, n, k, byrow=TRUE) ) )
  return (sum(log(llks)))
}
```

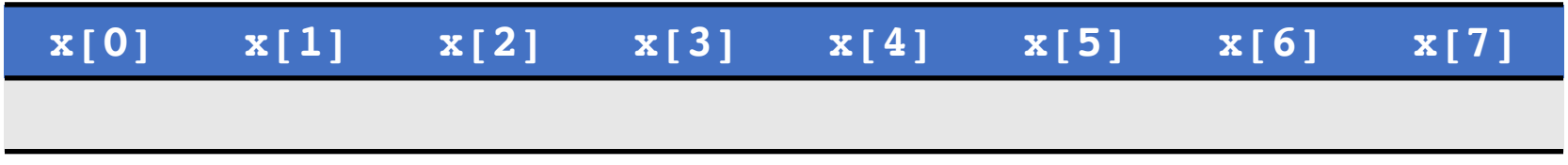
Expand the data into n x k matrix, and calculate the likelihood for each component, and take the weighted sum

Example of parameter assignments (k=3)

$$l\left(\boldsymbol{x};\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\sigma}^2\right)$$

(π_1, π_2, π_3) (μ_1, μ_2, μ_3) $(\sigma_1, \sigma_2, \sigma_3)$

How to assign?



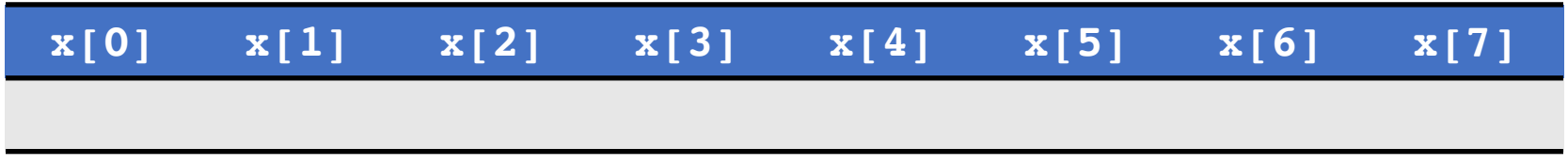
Mapping under certain constraints

$$l\left(\boldsymbol{x};\boldsymbol{\pi},\boldsymbol{\mu},\boldsymbol{\sigma}^2\right)$$

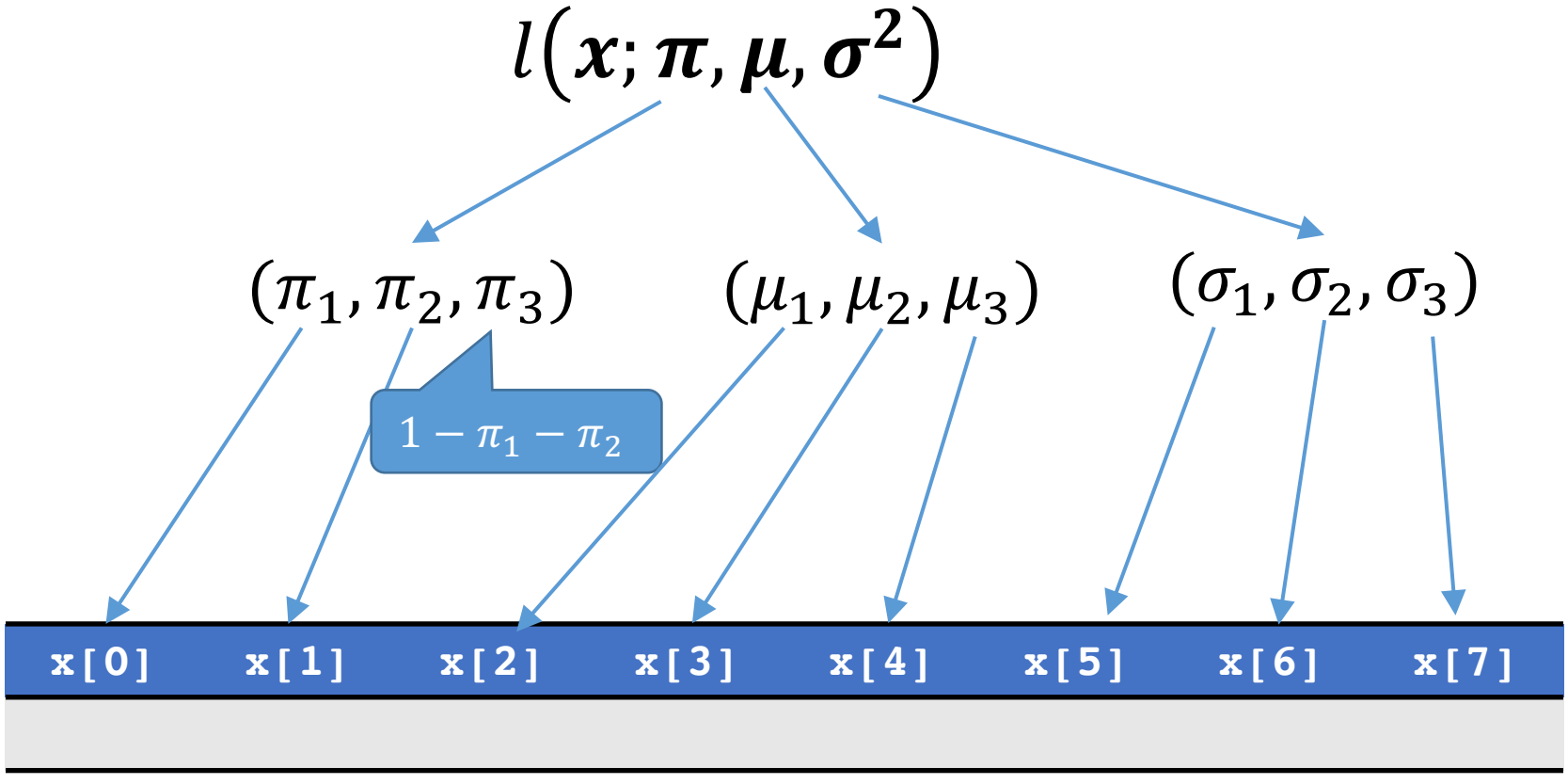
(π_1, π_2, π_3) (μ_1, μ_2, μ_3) $(\sigma_1, \sigma_2, \sigma_3)$

Constraints: $\pi_1 + \pi_2 + \pi_3 = 1$, $0 < \pi_i < 1$; $\sigma_i > 0$

How to assign?



Naive mapping



Using `optim()` function for finding MLE

```
mle.gmm <- function(x, start) {
  k <- (length(start)+1)/3
  r <- optim( start, function(y)  {
    0-llk.gmm(x, ## observed data
              c(y[1:(k-1)],1-sum(y[1:(k-1)])), ## pis
              y[k:(2*k-1)],    ## mus
              y[(2*k):(3*k-1)])  ## sds
  }, control=list(maxit=10000))
  return(r)
}
```

An example run

```
x <- c( rnorm(1000), rnorm(500) + 5 )  ## 1:2 mixture of N(0,1) and N(5,1)
mle.gmm(x, c(0.5, -4, 6, sd(x), sd(x)))
```

```
NaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs prod
ucedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs
 producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs produced
NaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs prod
ucedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs
 producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs produced
NaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs prod
ucedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs
 producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs produced
NaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs prod
ucedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs
 producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs produced
NaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs prod
ucedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs
 producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs produced
NaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs producedNaNs prod
ucedNaNs producedNaNs producedNaNs producedNaNs producedNaNs produced
```

An example run

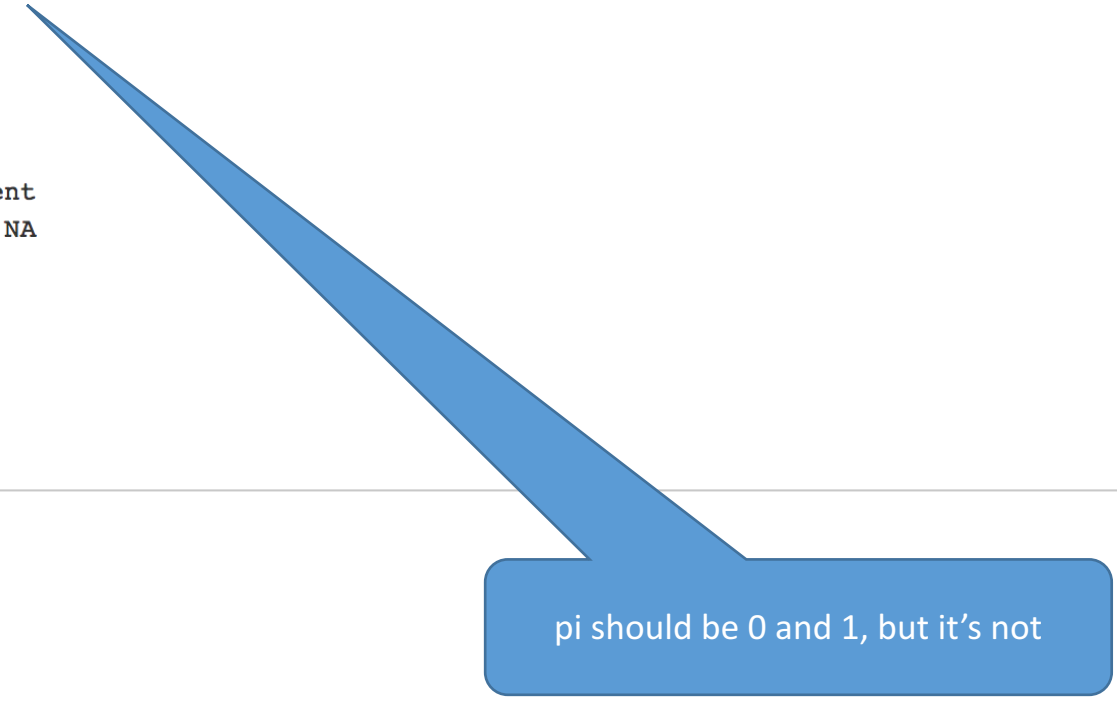
```
$par
[1] -3.869544e+08 -3.094609e+08  1.632685e+00  2.374680e+09  2.588915e+00

$value
[1] -26105.38

$counts
function gradient
    1344       NA

$convergence
[1] 0

$message
NULL
```



Avoiding boundary condition

- Using $(\pi_1, \cdots, \pi_{k-1})$ as k-1 parameters have a risk to reach to a boundary condition.

- We need a 1:1 transformation between $(\pi_1, \cdots, \pi_{k-1})$ and a unconstrained $\mathbb{R}^{k-1}$ space

- Note that $\pi_1 + \cdots + \pi_{k-1} < 1$ is another implicit constraint to enable 1:1 transformation.

Transforming between parameter spaces

- **When k=2, the logistic transformation works**

$$\eta_1 = \log\left(\frac{\pi_1}{1-\pi_1}\right) \qquad \pi_1 = \frac{1}{1+e^{-\eta_1}}$$

- **When k>2 …**

$$\eta_1 = \log\left(\frac{\pi_1}{1-\pi_1}\right) \qquad \eta_2 = \log\left(\frac{\pi_2}{1-\pi_2}\right)$$

$$\pi_1 = \frac{1}{1+e^{-\eta_1}} \qquad \pi_2 = \frac{1}{1+e^{-\eta_2}} \qquad \pi_1 + \pi_2 < 1\,?$$

When k > 2…

- **To ensure 1:1 matching for all cases, define**

$$\tau_1 = \pi_1 \qquad \tau_2 = \frac{\pi_2}{1-\pi_1} \quad \cdots \quad \tau_i = \frac{\pi_i}{1-\sum_{j=1}^{i-1}\pi_j}$$

- **And use the logistic transformation on τ_i**

$$\eta_i = \log\left(\frac{\tau_i}{1-\tau_i}\right) \qquad \tau_i = \frac{1}{1+e^{-\eta_i}}$$

$$\pi_i = \left(1-\sum_{j=1}^{i-1}\pi_j\right)\tau_i$$

Using re-parametrization

```
mle.gmm2 <- function(x, pi0, mu0, sd0) {
  k <- length(pi0)
  start <- c(restricted2free(pi0), mu0, log(sd0))
  r <- optim( start, function(y)  {
    pis <- free2restricted(y[1:(k-1)])
    sds <- exp(y[(2*k):(3*k-1)])
    0-llk.gmm(x, ## observed data
              pis,                    ## new pis
              y[k:(2*k-1)],    ## mus
              sds)                    ## sds
  }, control=list(maxit=10000) )
  return(list(pis=free2restricted(r$par[1:(k-1)]),
              mus=r$par[k:(2*k-1)],sds=exp(r$par[(2*k):(3*k-1)])))
}
```

Initial parameters are now pi, mu, sd scales

Also log-transformed sd parameters

Converting priors to unrestricted space

```
## convert k dimensional pis to k-1 unconstrained dimensions
restricted2free <- function(pis) {
  k <- length(pis)
  ret <- vector(length=k-1)
  res <- 1.0;
  for(i in 1:(k-1)) {
    v <- res * pis[i];
    ret[i] <- log(v/(1-v))
    res <- res * (1-pis[i])
  }
  return(ret);
}
```

Converting back from unrestricted to priors

```
## convert k-1 unconstrained parameters to pi
free2restricted <- function(x) {
  k <- length(x)+1
  pis <- vector(length=k)
  res <- 1.0;
  for(i in 1:(k-1)) {
    v <- 1/(1+exp(0-x[i]))
    pis[i] <- res * v
    res <- res * (1-v)
  }
  pis[k] <- res
  return(pis)
}
```

Running examples

```
x <- c( rnorm(1000), rnorm(500) + 5 )  ## 2:1 mixture of N(0,1) and N(5,1)
mle.gmm2(x, c(0.5, 0.5), c(-4, 6), c(sd(x), sd(x)) )
```

```
$pis
[1] 0.666703 0.333297

$mus
[1] -0.01028766  5.05301843

$sds
[1] 1.012297 1.014218
```

Defining normal mixture likelihood in C++

```
#include <Rcpp.h>
#include <cmath>

using namespace std;
using namespace Rcpp;

class normMixLLK {
public:
  int numComponents;
  NumericVector data;

  normMixLLK(int k, NumericVector& y) {
    numComponents = k;
    data = y;
  }
```

Converting parameter space...

```
// change real values to prior scales
// convert each observations  x1, x2, ..., xn
//            to logit scale, q1, q2, ..., qn
//            and use         q1, (1-q1)*q2, (1-q1)*(1-q2)*q3, ...,
//            to make sure that things adds up to 1 or less
static void assignPriors(const NumericVector& x, NumericVector& priors) {
  double p = 1.;
  int k = priors.size();
  for(int i=0; i < k-1; ++i) {
    double invLogit = 1.0/(1.0 + exp(0-x[i]));
    priors[i] = p*invLogit;
    p *= (1.0 - invLogit);
  }
  priors[k-1] = p;
}
```

Converting priors to unrestricted scale

```
static void priors2params(const NumericVector& priors, NumericVector& x) {
  double p = 1.0;
  int k = priors.size();
  for(int i=0; i < k-1; ++i) {
    double y = priors[i] / p;
    x[i] = log(y/(1.0-y));
    p *= (1-priors[i]);
  }
  x[k-1] = log(p/(1.0-p));
}
```

static function can be called without instance, such as

`normMixLLK::priors2params( ... )`

Likelihood Function

```
double operator() (const NumericVector& x) {
  NumericVector priors(numComponents);
  NumericVector means(numComponents);
  NumericVector sigmas(numComponents);

  assignPriors(x, priors);

  means = x[Range(numComponents-1, 2*numComponents-2)];
  sigmas = x[Range(2*numComponents-1, 3*numComponents-2)];

  int n = data.size();
  double llk = 0;
  for(int i=0; i < n; ++i) {
    double sum = 0;
    for(int j=0; j < numComponents; ++j)
        sum += (priors[j] * R::dnorm(data[i], means[j], exp(sigmas[j]), 0));
    if ( sum == 0 ) sum = 1e-300;
    llk += log(sum);
  }
  return 0-llk;
}
```

Converting 3k-1 dimensional parameters to priors, means, and sds

Actual part that calculates the log-likelihood

Just to avoid the boundary case

Using `simplex615` for optimization

```
// [[Rcpp::export]]
List runGMMSimplex(NumericVector data, int k, NumericVector pi0, NumericVector mu0,
 NumericVector sigma0, double tol) {
  // start with uniform priors
  normMixLLK foo(k, data); // make an instance of function object
  NumericVector startPoint(3*k-1);
  normMixLLK::priors2params(pi0, startPoint);
  startPoint[Range(k-1, 2*k-2)] = mu0;
  for(int i=0; i < k; ++i)
    startPoint[i+2*k-1] = log(sigma0[i]);

  simplex615<normMixLLK> simplex(startPoint); // set parameters
  simplex.amoeba(foo, tol);   // run simplex method
```

The rest of the function..

```
NumericVector pi(k);
NumericVector mu(k);
NumericVector sigma(k);
const NumericVector& xmin = simplex.xmin();
normMixLLK::assignPriors(xmin, pi);
mu = xmin[Range(k-1,2*k-2)];
for(int i=0; i < k; ++i)
  sigma[i] = exp(xmin[2*k-1+i]);

return( List::create(Named("ymin")=simplex.ymin(),
                     Named("pi.min")=pi,
                     Named("mu.min")=mu,
                     Named("sigma.min")=sigma,
                     Named("iter")=simplex.niter()
                     ));
}
```

Running example

```
x <- c( rnorm(1000), rnorm(500) + 5 )
runGMMSimplex(x, 2, c(0.5, 0.5), c(-4, 6), c(1,1), 1e-6)
```

```
$ymin
[1] 3045.914

$pi.min
[1] 0.6655409 0.3344591

$mu.min
[1] 0.01310131 4.99653246

$sigma.min
[1] 0.9971188 0.9729487

$iter
[1] 165
```

Summary : Nelder-Mead algorithm

- A general minimization method without requiring derivatives
- Uses simplex crawls towards the local minimum.
- Very widely used, quite robust, but no convergence guarantee.
- Reparameterization may be necessary for restricted parameter space.
- Works well with Gaussian mixtures