Biostatistics 615/815 Lecture 14: Single Dimensional and Multidimensional Optimizations

Hyun Min Kang

November 1st, 2011

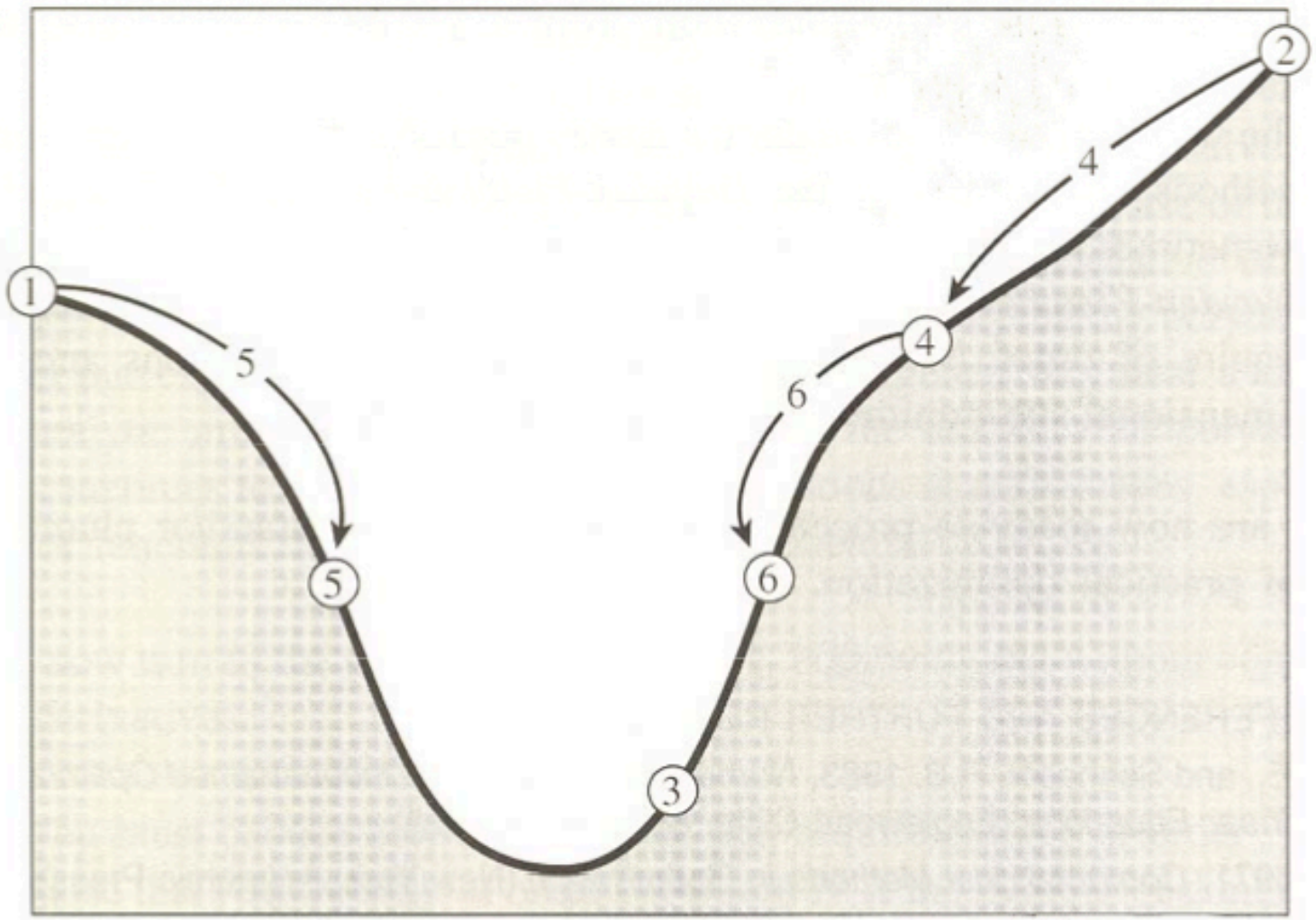
Recap : Root Finding Algorithms

- Implemented two methods for root finding
  - Bisection Method : `binaryZero()`
  - False Position Method : `linearZero()`
- In the bisection method, the bracketing interval is halved at each step
- For well-behaved function, the False Position Method will converage faster, but there is no performance guarantee.

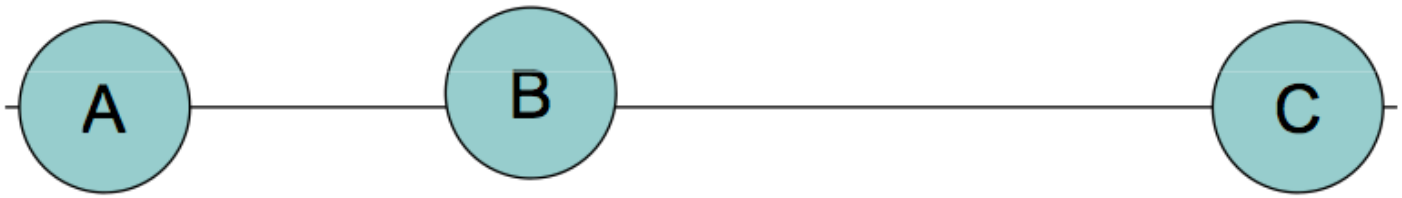
Recap : Root Finding with C++

```
double binaryZero(myFunc foo, double lo, double hi, double e) {
  for (int i=0;; ++i) {
    double d = hi - lo;  // f(lo) < 0, f(hi) > 0, d can be positive or negative
    double point = lo + d * 0.5;   // d is + for increasing func, - for decreasing
    double fpoint = foo(point);    // evaluate the value of the function
    if (fpoint < 0.0) {
      d = lo - point;  lo = point;
    }
    else {
      d = point - hi;  hi = point;
    }
    // e is tolerance level (higher e makes it faster but less accruate)
    if (fabs(d) < e || fpoint == 0.0) {
      std::cout << "Iteration " << i << ", point = " << point
                << ", d = " << d << std::endl;
      return point;
    }
  }
}
```

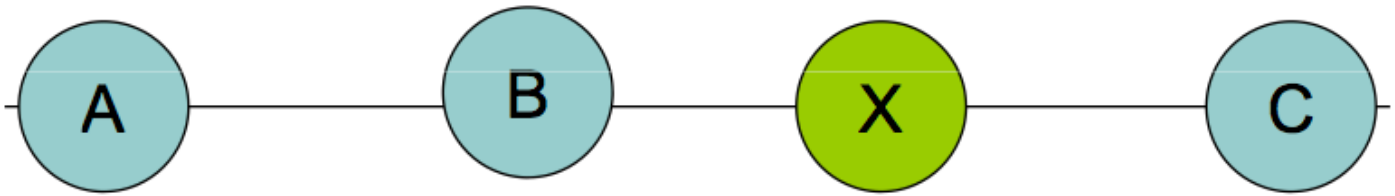
Recap : The Golden Search



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

Minimizing worst case possibility

- Formulae

$$w = \frac{b-a}{c-a}$$
$$z = \frac{x-b}{c-a}$$

  Segments will have length either $1-w$ or $w+z$.

- Optimal case

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

- Solve It

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

Recap : 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;
}
```

Today

A better single-dimensional optimization

- Parabolic interpolation
- Adaptive method
- Utilizing `boost` library

Multi-dimensional optimization

- Simplex algorithm

Using templates with function objects

Last Lecture : Using function object

```
class myFunc {
public: double operator() (double x) const { ... }
};
double binaryZero(myFunc foo, double lo, double hi, double e) {
  ...
}
```

Today : Using function object with templates

```
template<class F>
double binaryZero(F foo, double lo, double hi, double e) {
  ...
}
```

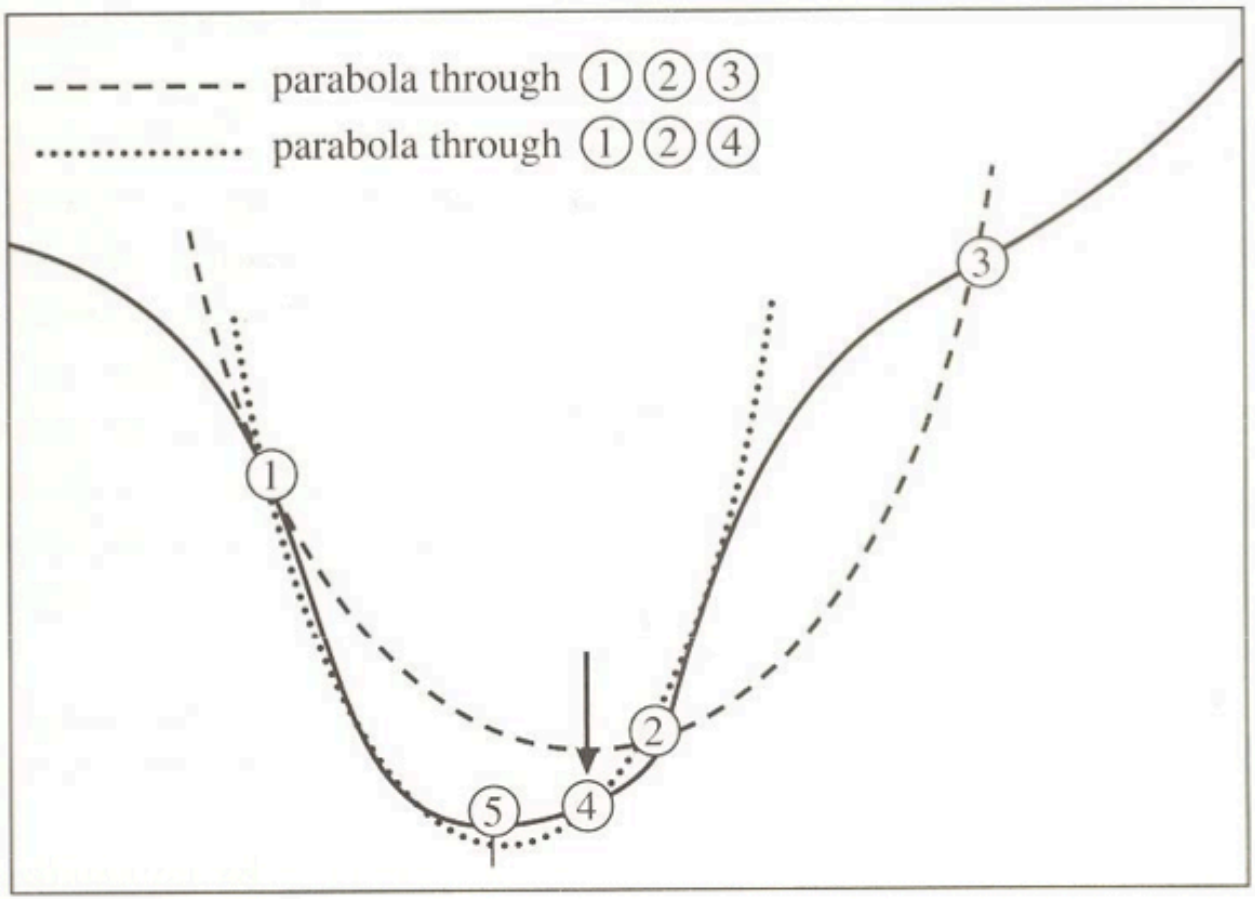
Using templates with function objects

```
class myFunc1 {
  public: double operator() (double x) const { return sin(x); }
};
class myFunc2 {
  public: double operator() (double x) const { return exp(x)-1; }
};
double foo3(double x) { return x*cos(x); }

template<class F>
double binaryZero(F foo, double lo, double hi, double e) {
  ...
}
int main(int argc, char** argv) {
  myFunc1 foo1;
  myFunc2 foo2;
  // equivalent to binaryZero<myFunc1>(foo1, 0-M_PI/4, M_PI/2, 1e-6);
  binaryZero(foo1, 0-M_PI/4, M_PI/6, 1e-6);
  // equivalent to binaryZero<myFunc2>(foo1, 0-M_PI/4, M_PI/2, 1e-6);
  binaryZero(foo2, 0-M_PI/4, M_PI/6, 1e-6);
  binaryZero(foo3, 0-M_PI/4, M_PI/6, 1e-6);
  return 0;
}
```

Better optimization using local approximation

- Root finding example
  - Binary search reduces the search space by constant factor $1/2$
  - Linear approximation may reduce the search space more rapidly for most well-defined functions
- Minimization problem
  - Golden search reduces the search space by 38%
  - Using a quadratic approximation of the function may achieve better optimization results

Approximation using parabola



Parabolic Approximation

$$f^*(x) = Ax^2 + Bx + C$$

The value minimizes $f^*(x)$ is

$$x_{min} = -\frac{B}{2A}$$

This strategy is called "inverse parabolic interpolation"

Fitting a parabola

- Can be fitted with three points
- Points must not be co-linear
- $f^*(x_1) = f(x_1), f^*(x_2) = f(x_2), f^*(x_3) = f(x_3)$.

$$
\begin{aligned}
C &= f(x_1) - Ax_1^2 - Bx_1 \\
B &= \frac{A(x_2^2 - x_1^2) + f(x_1) - f(x_2)}{x_1 - x_2} \\
A &= \frac{f(x_3) - f(x_2)}{(x_3 - x_2)(x_3 - x_1)} - \frac{f(x_1) - f(x_2)}{(x_1 - x_2)(x_3 - x_1)}
\end{aligned}
$$

Minimum for a Parabola

- General expression for finding minimum of a parabola fitted through three points

$$x_{min} = x_2 - \frac{1}{2}\frac{(x_2 - x_1)^2(f(x_2) - f(x_1)) - (x_2 - x_3)^2(f(x_2) - f(x_1))}{(x_2 - x_1)(f(x_2) - f(x_3)) - (x_2 - x_3)(f(x_2) - f(x_3))}$$

Fitting a Parabola

```
// Returns the distance between b and the abscissa for the
// fitted minimum using parabolic interpolation
double parabolaStep (double a, double fa, double b, double fb, double c,
                     double fc) {
  // Quantities for placing minimum of fitted parabola
  double p = (b - a) * (fb - fc);
  double q = (b - c) * (fb - fa);
  double x = (b - c) * q - (b - a) * p;
  double y = 2.0 * (p - q);
  // Check that y is not too close to zero
  if (fabs(y) < ZEPS)
    return goldenStep (a, b, c);
  else
    return x / y;
}
```

Avoiding degenerate case

- Fitted minimum could overlap with one of original points
- Ensure that each new point is distinct from previously examined points

Avoiding degenerate steps

```
double adjustStep(double a, double b, double c, double step, double e) {
  double minStep = fabs(e * b) + ZEPS;
  if (fabs(step) < minStep)
    return step > 0 ? minStep : 0-minStep;
    // If the step ends up to close to previous points,
    // return zero to force a golden ratio step ...
  if (fabs(b + step - a) <= e || fabs(b + step - c) <= e)
    return 0.0;
  return step;
}
```

Generating New Points

- Use parabolic interpolation by default
- Check whether improvement is slow
- If step sizes are not decreasing rapidly enough, switch to golden section

Adaptive calculation of step size

```
double calculateStep(double a, double fa, double b, double fb,
                     double c, double fc, double lastStep, double e) {
  double step = parabolaStep(a, fa, b, fb, c, fc);
  step = adjustStep(a, b, c, step, e);
  if (fabs(step) > fabs(0.5 * lastStep) || step == 0.0)
     step = goldenStep(a, b, c);
   return step;
}
```

Overall

The main function simply has to

- Generate new points using building blocks
- Update the triplet bracketing the minimum
- Check for convergence

Overall Minimization Routine

```
template<class F>
double adaptiveMinimum(F foo, double a, double b, double c, double e) {
  double fa = foo(a), fb = foo(b), fc = foo(c);
  double step1 = (c - a) * 0.5, step2 = (c - a) * 0.5;
  while ( fabs(c - a) > fabs(b * e) + ZEPS) {
     double step = calculateStep (a, fa, b, fb, c, fc, step2, e);
     double x = b + step;
     double fx = foo(x);
     if (fx < fb) {
        if (x > b) { a = b; fa = fb; }
        else { c = b; fc = fb; }
        b = x; fb = fx;
     }
     else {
        if (x < b) { a = x; fa = fx; }
        else { c = x; fc = fx; }
        step2 = step1; step1 = step;
     }
  }
  return b;
}
```

Important Characteristics

- Parabolic interpolation often converges faster
  - The preferred algorithm
- Golden search provides worst-cast performance guarantee
  - A fall-back for uncooperative functions
- Switch algorithms when convergence is slow
- Avoid testing points that are too close

More advanced strategy : Brent's algorithm

- Track 6 points (not all distinct)
  - The bracket boundaries (a, b)
  - The current minimum x
  - The second and third smallest value (w, v)
  - The new points to be examined u
- Parabolic interpolation
  - Using (x, w, v) to propose new value for u.
  - Additional care is required to ensure u falls between a and b.
- Recommended Reading
  - Numerical Recipes in C++ : Chapter 10.0 - 10.3

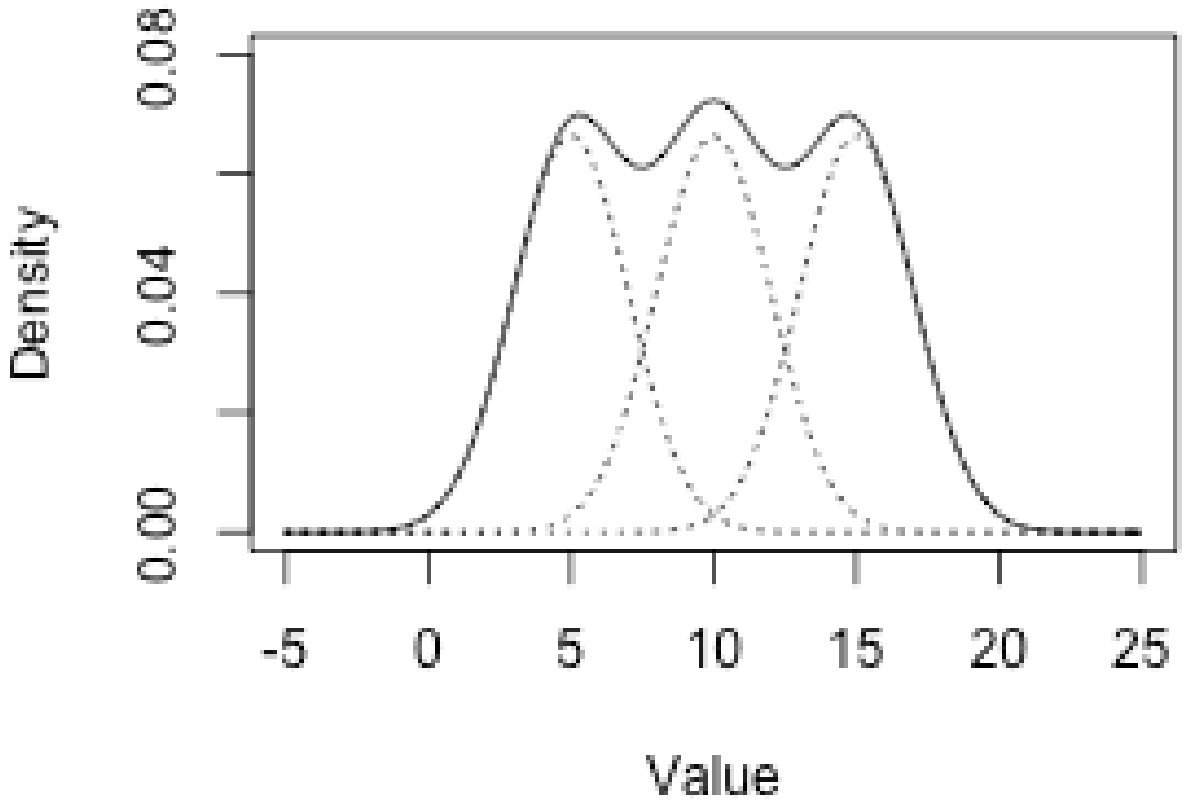
Using boost library for root finding / minimization

```
#include <cmath>
#include <iostream>
#include <boost/math/tools/roots.hpp>

#define EPS 1e-6

bool tol(double a, double b) { return ( fabs(b-a) <= EPS ); }

int main(int argc, char** argv) {
  double lo = 0-M_PI/4;
  double hi = M_PI/2;
  boost::uintmax_t niter;
  std::pair<double,double> rBi = boost::math::tools::bisect(sin, lo, hi, tol, niter);
  std::cout << "bisect : (" << rBi.first << ", " << rBi.second
            << ") at " << niter << " iterations" << std::endl;
  std::pair<double,double> r748 =
             boost::math::tools::toms748_solve(sin, lo, hi, tol, niter);
  std::cout << "toms748 : (" << r748.first << ", " << r748.second
            << ") at " << niter << " iterations" << std::endl;
  return 0;
}
```

Other Algorithms for Root Finding

- TOMS Algorithm 748
  - Uses a mixture of cubic, quadratic, and linear interpolation to locate the root of $f(x)$.
- Newton-Raphson algorithm
  - Uses first derivative of $f(x)$ to better approximate the root
- Halley's method
  - Uses first and second derivatives of $f(x)$ to approximate the root
- Householder's method
  - Uses up to d-th derivative of $f(x)$ to approximate the root for faster convergence

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 numer 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
  - $\mu_1, \mu_2, \cdots, \mu_k$
  - $\sigma_1^2, \sigma_2^2, \cdots, \sigma_k^2$

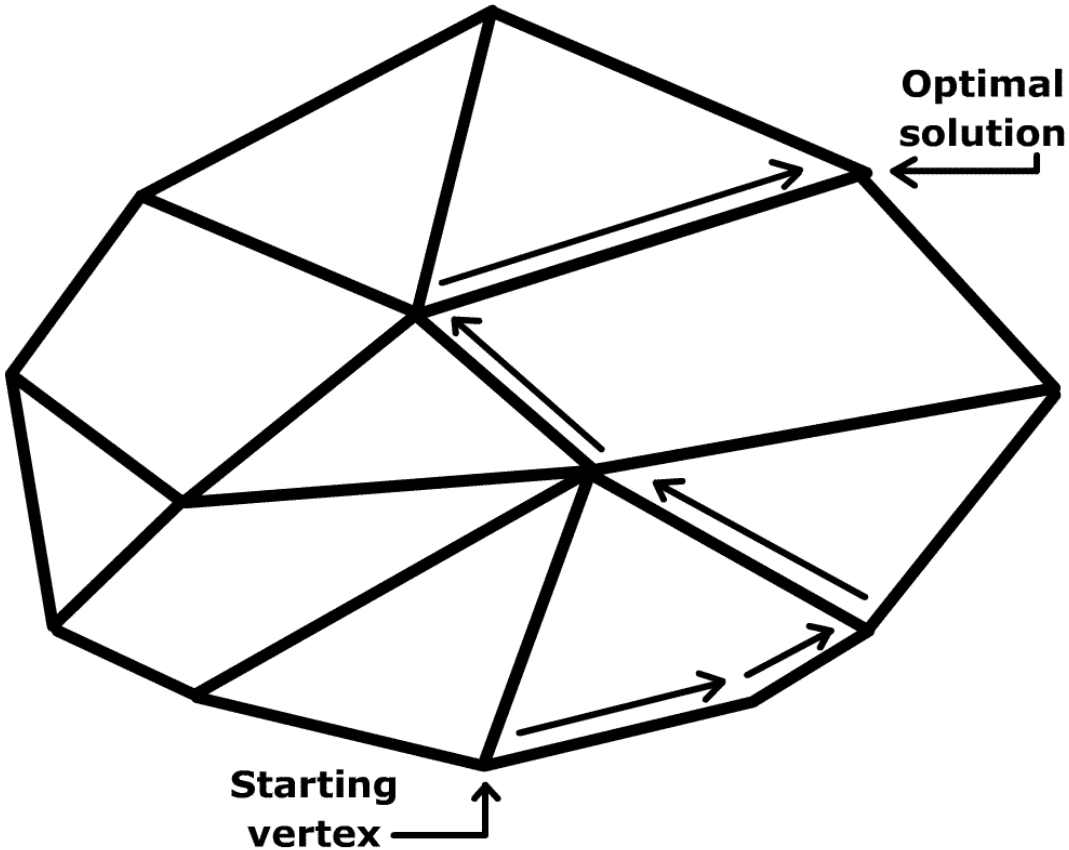
Possible approaches

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

The Simplex Method

- Calculate likelihoods at simplex vetices
  - Geomtric 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

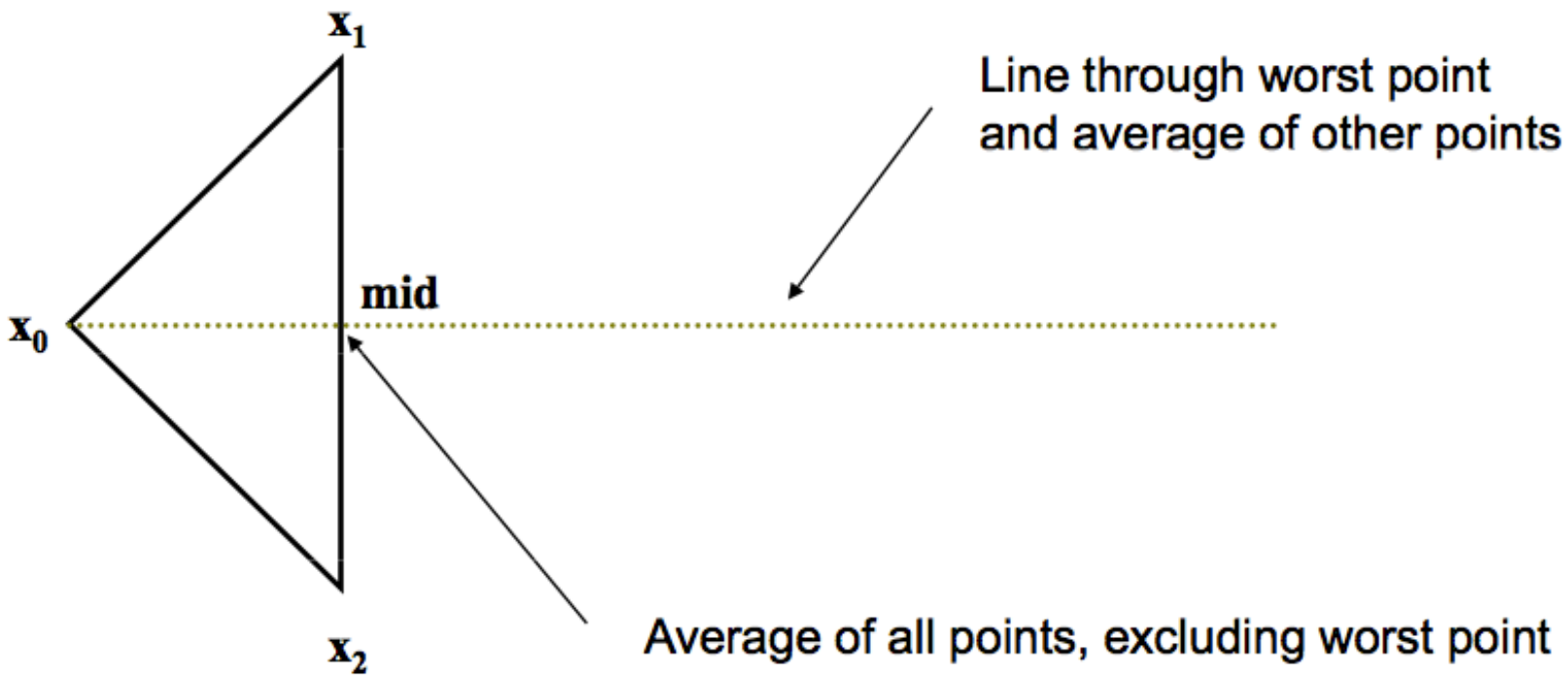
How the Simplex Method Works



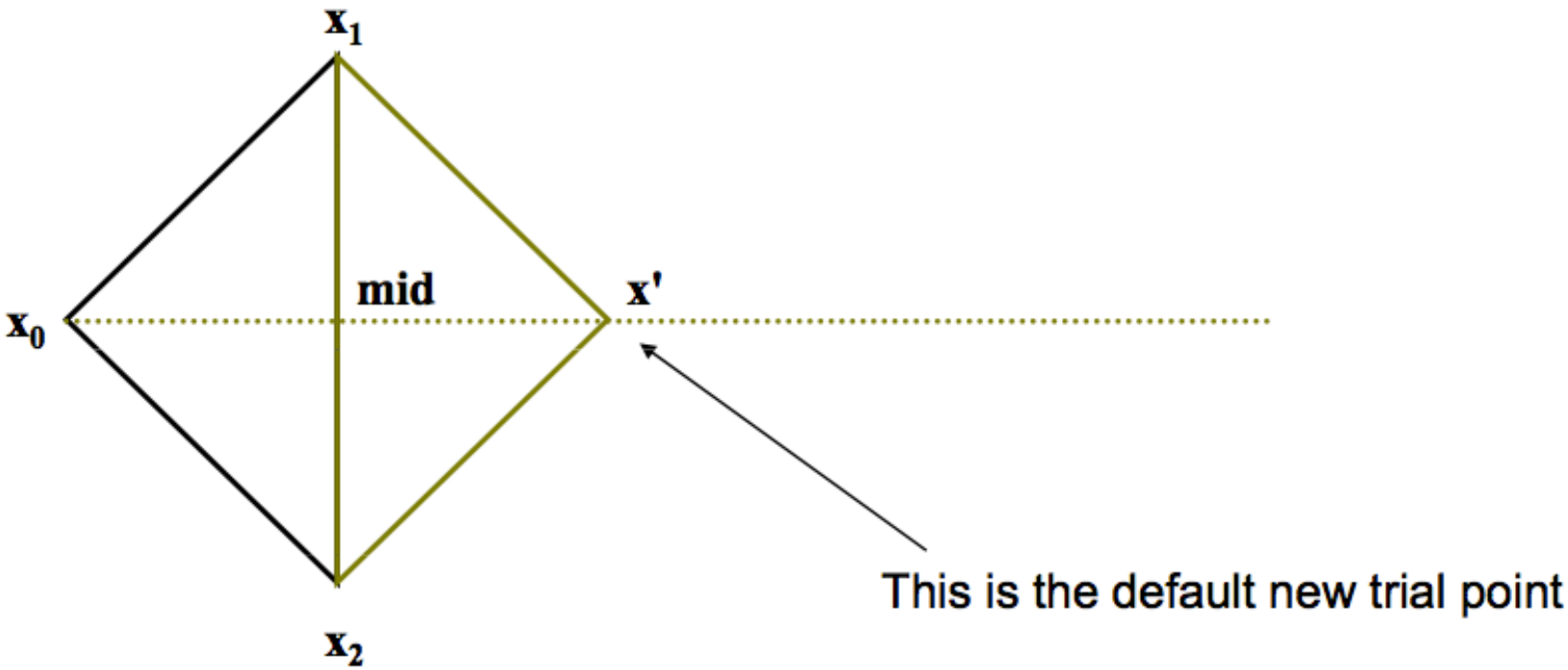
Simplex Method in Two Dimensions

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

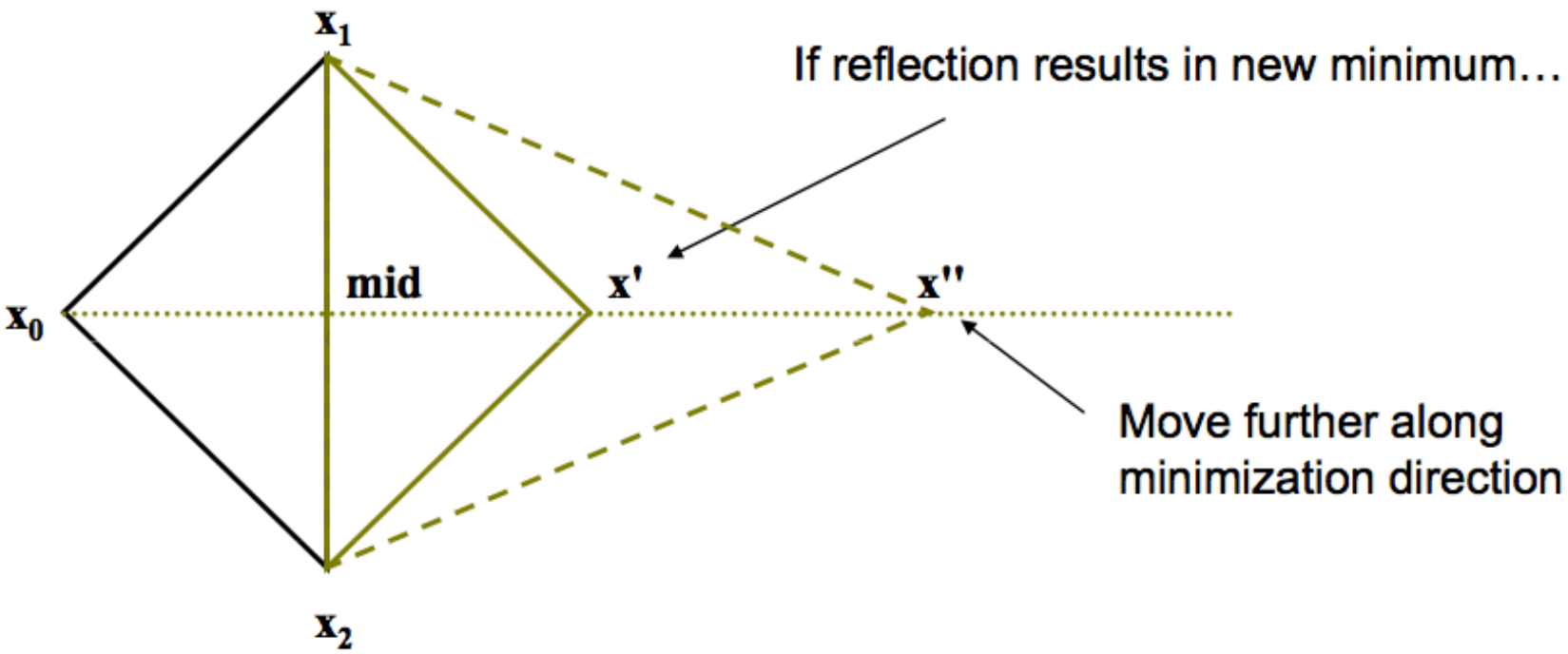
Direction for Optimization



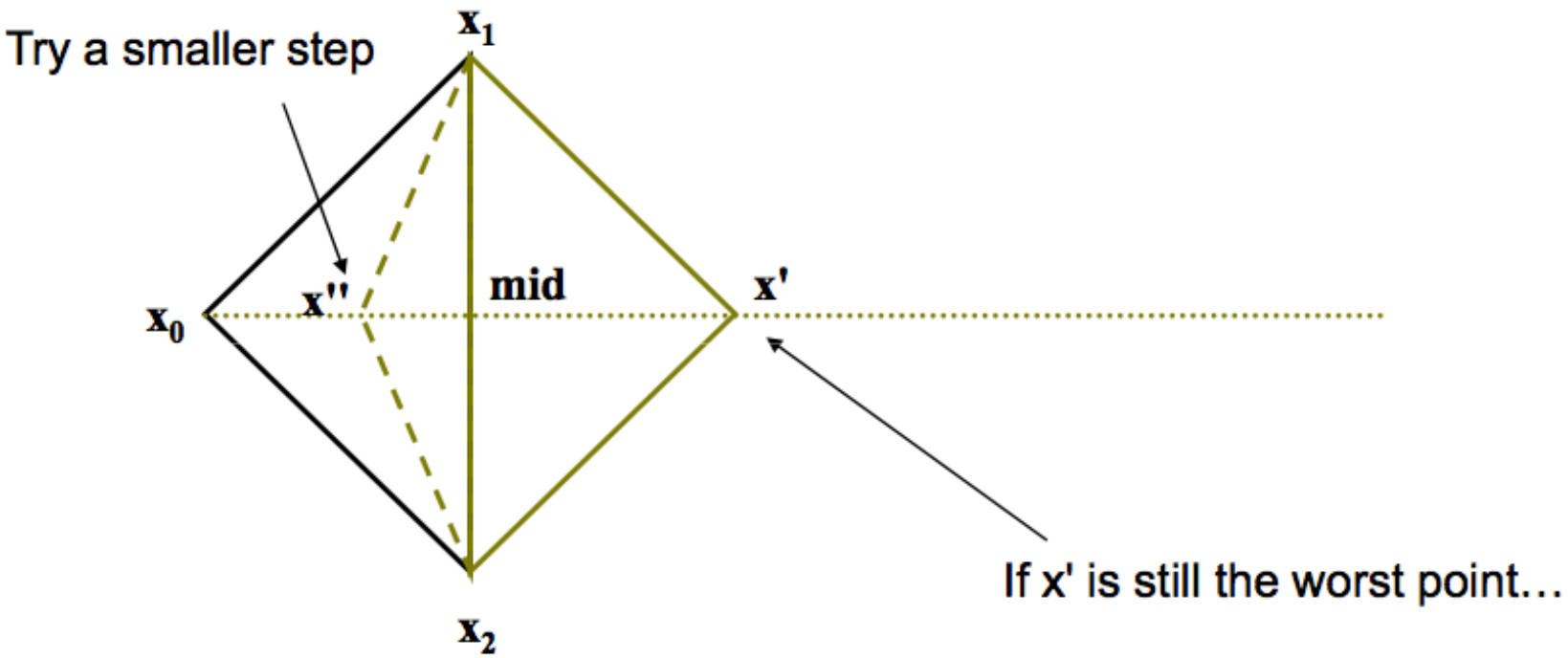
Reflection



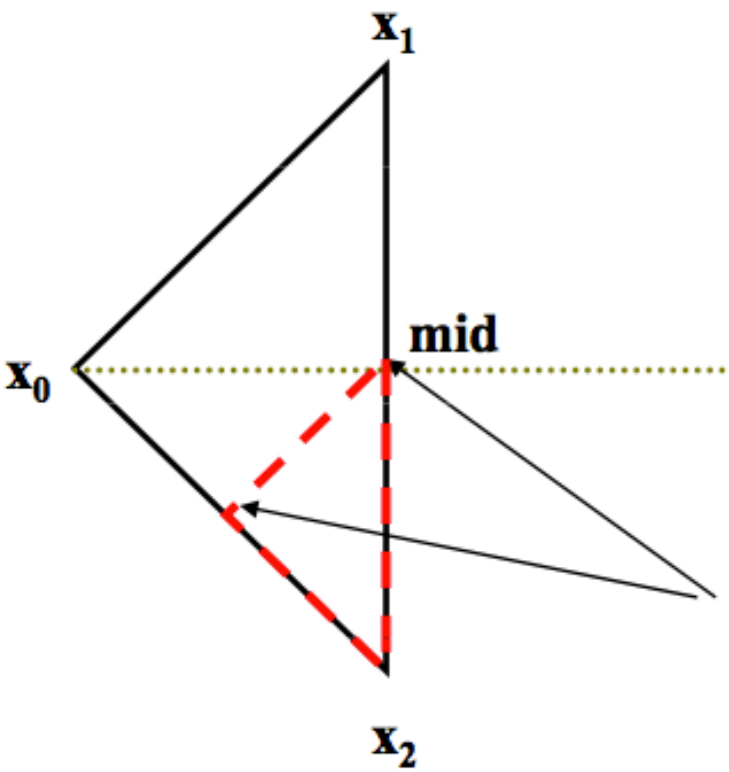
Reflection and Expansion



Contaction (1-dimension)



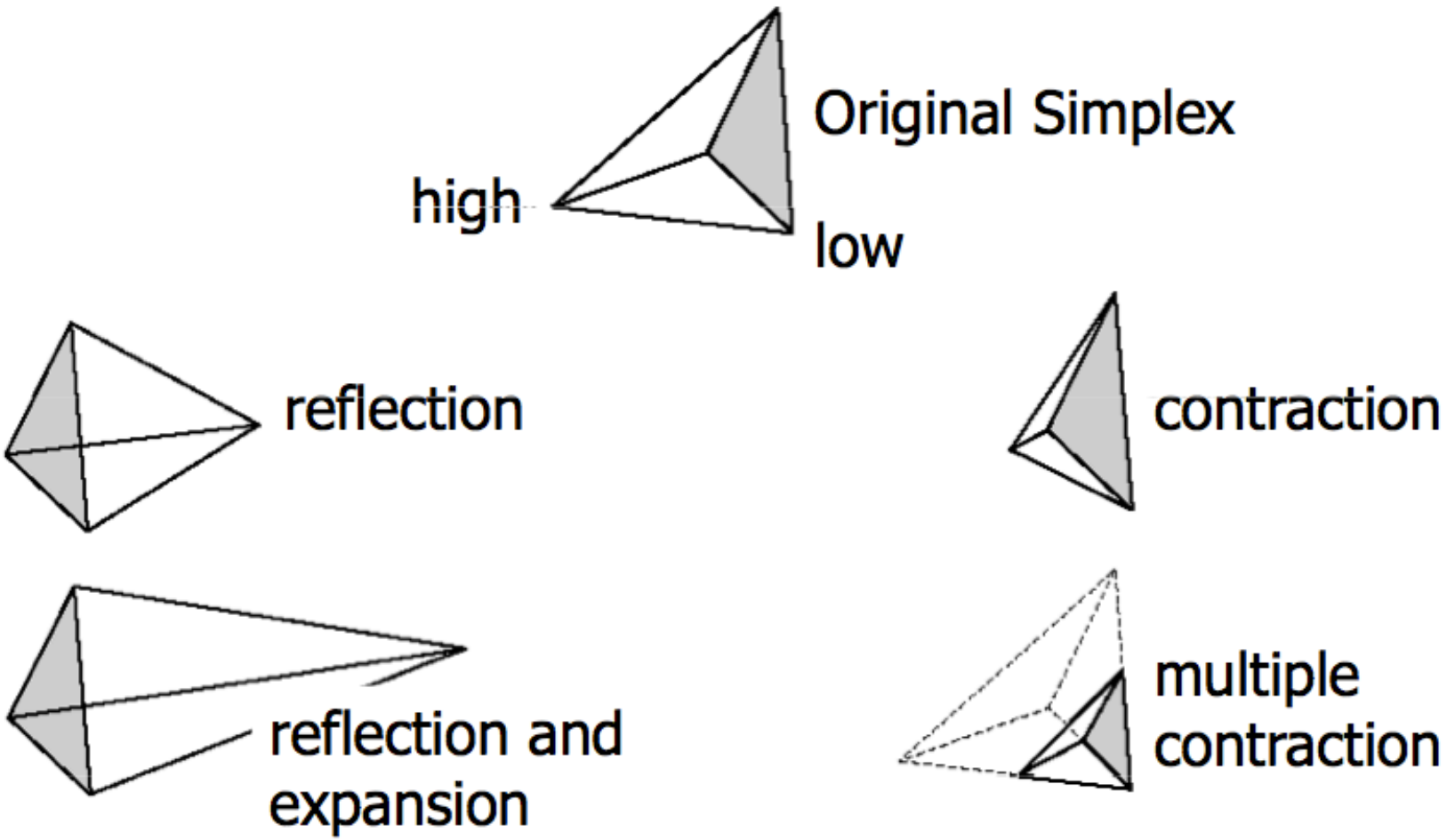
Contaction



"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



Today

- Single-dimensional minimization
  - Minimization using Parabola
  - Adaptive minimization using parabola and golden search
  - A taste of Brent's method
- Multi-dimensional optimization
  - Gaussian mixture example
  - Simplex algorithm

Upcoming lectures

Next Lecture

- Details of simplex algorithm
- Expectation-Maximization algorithm