2011 FALL BIOSTAT 615/815 Homework #5

- Due is Tuesday November 29th, 08:40AM (before the class starts).
- You need to both (1) hand-in your hard copy of the source code (using smaller font is fine), and (2) submit your source codes (ready to compile and run) in a zip or tar.gz compressed format by email.

Problem 1. Evaluation of Simplex Method

Consider the vector norm function:

$$f(\mathbf{x}) = |\mathbf{x}| = \sqrt{\sum_{i=1}^{k} x_i^2} \tag{1}$$

The function has minimum 0 when $x = 0$.

We will use this function to examine the efficacy of the Nelder-Mead algorithm for function minimization across the following differeent parameters using simplex algorithm.

- $k \in \{2, 5, 10, 20\}$
- Starting point of each dimension is independently sampled from $N(\mu, \sigma^2)$ where $\mu \in \{0, 100\}$, $\sigma \in \{1, 10, 100\}$.

For each of the possible 24 ($4 \times 2 \times 3$) configurations, repeat running the Nelder-Mead algorithm at least 5 times for each configuration to find the minimum of the vector norm function above (use 10^{-7} as the relative accuracy threshold). Then submit a tab-delimited file where each line contains the 5 columns

1. k
2. μ
3. σ
4. The norm evaluated at the minimum point identified by the Nelder-Mead routine.
5. The total number of function evaluations carried out.

When generating starting point, use current timestamp to randomize your outcomes. You may (but don't have to) start from the source code given in the lecture. The source code is also available in the class web page

```
#ifndef __SIMPLEX_615_H
#define __SIMPLEX_615_H

#include <vector>
#include <cmath>
#include <iostream>

#define ZEPS 1e-10

// Simplex contains (dim+1)*dim points
template <class F>
class simplex615 {
 protected:
  std::vector<std::vector<double> > X;
  std::vector<double> Y;
  std::vector<double> midPoint;
  std::vector<double> thruLine;

  int dim, idxLo, idxHi, idxNextHi;

  void evaluateFunction(F& foo);
```

```
  void evaluateExtremes();
  void prepareUpdate();
  bool updateSimplex(F& foo, double scale);
  void contractSimplex(F& foo);
  static int check_tol(double fmax, double fmin, double ftol);

 public:
  simplex615(double* p, int d);
  void amoeba(F& foo, double tol);
  std::vector<double>& xmin();
  double ymin();
};

template <class F>
simplex615<F>::simplex615(double* p, int d) : dim(d) {
  X.resize(dim+1);
  Y.resize(dim+1);
  midPoint.resize(dim);
  thruLine.resize(dim);
  for(int i=0; i < dim+1; ++i) {
    X[i].resize(dim);
  }

  // set every point the same
  for(int i=0; i < dim+1; ++i) {
    for(int j=0; j < dim; ++j) {
      X[i][j] = p[j];
    }
  }

  // then increase each dimension by one except for the last point
  for(int i=0; i < dim; ++i) {
    X[i][i] += 1.;
  }
}

template <class F>
void simplex615<F>::evaluateFunction(F& foo) {
  for(int i=0; i < dim+1; ++i) {
    Y[i] = foo(X[i]);
  }
}

template <class F>
void simplex615<F>::evaluateExtremes() {
  if ( Y[0] > Y[1] ) {
    idxHi = 0;
    idxLo = idxNextHi = 1;
  }
  else {
    idxHi = 1;
    idxLo = idxNextHi = 0;
  }

  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;
    }
  }
}

template <class F>
void simplex615<F>::prepareUpdate() {
  for(int j=0; j < dim; ++j) {
    midPoint[j] = 0;
  }
  for(int i=0; i < dim+1; ++i) {
    if ( i != idxHi ) {
      for(int j=0; j < dim; ++j) {
        midPoint[j] += X[i][j];
      }
    }
  }
  for(int j=0; j < dim; ++j) {
    midPoint[j] /= dim;
    thruLine[j] = X[idxHi][j] - midPoint[j];
  }
}

template <class F>
bool simplex615<F>::updateSimplex(F& foo, double scale) {
  std::vector<double> nextPoint;
  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] ) { // exchange with maximum
    for(int i=0; i < dim; ++i) {
      X[idxHi][i] = nextPoint[i];
    }
    Y[idxHi] = fNext;
    return true;
  }
  else {
    return false;
  }
}

template <class F>
void simplex615<F>::contractSimplex(F& foo) {
  for(int i=0; i < dim+1; ++i) {
    if ( i != idxLo ) {
      for(int j=0; j < dim; ++j) {
        X[i][j] = 0.5*( X[idxLo][j] + X[i][j] );
        Y[i] = foo(X[i]);
      }
    }
  }
}

template <class F>
void simplex615<F>::amoeba(F& foo, double tol) {
```

```
  evaluateFunction(foo);
  while(true) {
    evaluateExtremes();
    prepareUpdate();

    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) ) {
        contractSimplex(foo);
      }
    }
  }
}

template <class F>
std::vector<double>& simplex615<F>::xmin() {
  return X[idxLo];
}

template <class F>
double simplex615<F>::ymin() {
  return Y[idxLo];
}

template <class F>
int simplex615<F>::check_tol(double fmax, double fmin, double ftol) {
  double delta = fabs(fmax - fmin); double accuracy = (fabs(fmax) + fabs(fmin)) * ftol;
  return (delta < (accuracy + ZEPS));
}

#endif // __SIMPLEX_615_H
```

Problem 2. Traveling Salesman Problem

Implement the Traveling Salesman Problem program described in the class. Add an additional routine which performs random 300,000 permutation among the possible paths and takes the minimum distance among them.

Using the 10 and 11 point examples described in the class, compare the three methods : (1) exhausive search (2) 300,000 random paths, and (3) simulated annealing in terms of running time and the solutions. Run the program at least 5 independent times to compare the outcomes with different seeds.

Between (2) and (3), which one is find solutions closer to optimal solution? Explain why.

Problem 3. Single dimensional optimization for Gaussian Mixture Models

Consider the 2-component Gaussian Mixture Models

$$f(x) = \alpha\mathcal{N}(\mu_1, \sigma_1^2) + (1-\alpha)\mathcal{N}(\mu_2, \sigma_2^2)$$

Unlike the Gaussian Mixture model described in the class, here you need to estimated only α, given observed data and other parameters $\mu_1, \mu_2, \sigma_1, \sigma_2$. You will need to implement (1) Golden Search (1e-6 for relative accuracy threshold), (2) Brent algorithm implemented in the boost library (20 bits for precision parameter), and (3) E-M algorithm (1e-6 for relative accuracy threshold), which should be different from what is described in the class in order to accomplish this. The example outcome of the algorithm look like below (outcome depends on input data).

```
$ ./normMixAlphaEstimate
Usage : ./normMixAlphaEstimate [infile] [mean1] [sd1] [mean2][sd2]

$ ./normMixSimul 10000 0.8 0 1 5 1 > input.txt

$./normMixAlphaEstimate input.txt 0 1 5 1
Golden Search  : alpha = 0.794469, LLK = -19139, niter = 31
Brent Algorithm: alpha = 0.794471, LLK = -19139, niter = 11
E-M Algorithm  : alpha = 0.794466, LLK = -19139, niter = 3
```

You may use the routines described in the class, and for the usage of Brent algorithm implemented in the boost library, you may refer to the example code below.

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

class myFunc {
public:
  double operator()(double x) {
    return 0-cos(x);
  }
};

int main(int argc, char** argv) {
  myFunc minusCos;

  boost::uintmax_t niter;
  std::pair<double,double> r =
    boost::math::tools::brent_find_minima(minusCos, 0-M_PI/4, M_PI/2, 20, niter);
  std::cout << "Brent : x=" << r.first << ", f=" << r.second << ", niter=" << niter << std::endl;

  return 0;
}
```