2012 FALL BIOSTAT 615/815 Homework #5

Due is Saturday December 1st, 2012 11:59PM by google document (shared to hmkang@umich.edu and atks@umich.edu) containing the source code and answers to the questions. Also email of the compressed tar.gz file containing all the source codes.

Problem 1. Evaluation of Simplex Method (40 pts)

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 Simplex (Nelder-Mead) algorithm for function minimization across the following differeent parameters.

- $k \in \{2, 5, 10, 20\}$
- 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 table (or tab-delimited file) where each line contains the 5 columns in your submission

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 below.

```
#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. Single dimensional optimization for Gaussian Mixture Models (40 pts)

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]

% ./normMixAlphaEstimate ~hmkang/Public/615/data/normmix.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 adapt 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;
}
```

Problem 3. Importance Sampling and Simplex Method (70 pts)

The goal of this problem is to compute the expectation of logit-normal variable.

$$f(\mu, \sigma) = \int_{-\infty}^{\infty} \frac{1}{1+e^{-x}} \mathcal{N}(x; \mu, \sigma) dx$$

We want to implement and compare the following Monte-Carlo algorithms

- Crude Monte-Carlo algorithm using samples from uniform distribution (with very large boundaries)
- Crude Monte-Carlo algorithm using samples from the normal distribution where the latent variable x is sampled from.
- Important sampling algorithm using samples from an arbitrary normal distribution.

The main objective is to find the parameters for importance sampling that gives the smallest relative standard error. From a matrix of pre-sampled random variable R with $r \times n$ dimensions, where each $R_{ij} \sim N(0, 1)$ is i.i.d, we would like to compute the following quantity.

- Adjust n random variables from each row of R to follow $N(\mu', \sigma')$, and use importance sampling to approximate $f(\mu, \sigma)$.
- Repeat the procedure above r times, and calculate the mean and standard deviation of the estimated value, and calculate relative standard deviation (RSD).
- Across different trial parameters, find the optimal parameters using either grid search or using Simplex Algorithm.

(a) Perform a grid search by completing the R code below, and run `grid.logit.normal.integral(100, 1000, -8, 1)`. Copy and paste the output plot and include it in your submission. This will be easier when running in your local machine.

```r
sigmoid <- function(x) {
  ## implement sigmoid (logistic) function
}

unif.crude <- function(r, mu, sigma, lo = -50, hi = 50) {
```

```r
  ## 0. r is a vector
  ## 1. Assuming r follows N(0,1), convert it to u(lo,hi)
  ## 2. Calculate crude Monte-Carlo integral from uniform distribution
}

norm.crude <- function(r, mu, sigma) {
  ## 0. r is a vector
  ## 1. Convert r to follow N(mu,sigma)
  ## 2. Calculate crude Monte-Carlo integral from normal distribution
}

## u ~ unif(0,1) - convert u to follow the desired distribution
importance <- function(r, mu, sigma, adjm, adjs) {
  ## 0. r is a vector
  ## 1. Convert r to follow N(adjm,adjs)
  ## 2. Calculate importance sampling integral using (adjm,adjs)
}

rel.sd.importance <- function(R, mu, sigma, adjm, adjs) {
  ## 0. R is (nrep x nsample) matrix
  ## 1. For each R[i,], compute importance sampling integral
  ## 2. Calculate the mean and SD of estimate interval and return (SD/mean)
}

## r : number of repetition
## n : number of random samples for a single run of Monte-Carlo integral
## mu : mean of latent normal variable
## sigma : sd of latent normal variable
grid.logit.normal.integral <- function(r, n, mu, sigma) {
  ## use the same set of random values for comparison
  R <- matrix(rnorm(r*n),r,n)

  res.unif <- apply(R,1,unif.crude,mu,sigma);  ## uniform-crude
  res.norm <- apply(R,1,norm.crude,mu,sigma);  ## normal-crude

  rel.sd.unif <- sd(res.unif)/mean(res.unif);
  rel.sd.norm <- sd(res.norm)/mean(res.norm);

  adjms <- (10:1)/5*mu;
  adjss <- (1:10)/5*sigma;
  rel.sd.impt <- matrix(NA,10,10);
  for(i in 1:10) {
    for(j in 1:10) {
      rel.sd.impt[i,j] <- rel.sd.importance(R, mu, sigma, adjms[i], adjss[j])
    }
  }
  filled.contour(adjms, adjss, log10(rel.sd.impt),
                 color.palette = terrain.colors,
                 plot.title = title(main=paste("Importance sampling of\nlogit-normal integral (",
                    mu,",",sigma,")\ncrude uniform=",
                    sprintf("%.3f",log10(rel.sd.unif)),
                    ", crude normal=",
                    sprintf("%.3f",log10(rel.sd.norm)),sep=""),
                    xlab = "sampled mean",ylab = "sampled stdev",
                    cex.main=1),
                 plot.axes = { axis(1, seq(2*mu,0,by=1))
                               axis(2, seq(0,2*sigma,by=0.1)) },
                 key.title = title(main="log10\n(sd/mean)",cex.main=0.9),
                 );
}
```

(b) Perform a Simplex search by completing the R code below, including the R implementation of simplex algorithm. Note that R callee function cannot update the variables defined in the caller function, so a special routine was used to update the variables in the caller functions. Run `simplex.logit.normal.integral(100, 1000, -8, 1)`, and copy and paste the output text in your submission.

```
simplex <- function(func, start, tol) {
  simplex.evaluate.extremes <- function() {
    ## update ilo, ihi, ihi2 based on values of Y

    ## and update ilo, ihi, ihi2 in the parent function using lines below
    assign("ilo",ilo,envir=parent.env(environment()));
    assign("ihi",ihi,envir=parent.env(environment()));
    assign("ihi2",ihi2,envir=parent.env(environment()));
  }

  simplex.check.tol <- function(fmax, fmin, tol) {
    ## return true if passed tolerance criteria
  }

  simplex.update <- function(scale) {
    ## perform simplex update

    ## if new point is better than the worst point
    ## 1. Update X and Y in this function as needed
    ## 2. Reflect the update in the parent function using lines below
    ## assign("X",X,envir=parent.env(environment()));
    ## assign("Y",Y,envir=parent.env(environment()));
    ## 3. return TRUE
    ## if new point is not better than the worse point, return FALSE
  }

  simplex.contract <- function() {
    ## perform multiple contraction (update X and Y)
    ## 1. Update X and Y in this function as needed
    ## 2. Reflect the update in the parent function using lines below
    ## assign("X",X,envir=parent.env(environment()));
    ## assign("Y",Y,envir=parent.env(environment()));
  }

  ## main function goes here, should be similar to amoeba() function
  ## X is (dim+1) x (dim) simplexes
  ## Y is (dim+1) vector containing function evaluations
  ## compute midPoint and thruLine within each iteration

  ## print out the final outcome here
  print(paste("X=",paste(X[ilo,],collapse=","),"Y=",Y[ilo],"at iter=",iter,collapse="\t"))
}

## run simplex algorithm for finding optimal parameters for logit normal integration
simplex.logit.normal.integral <- function(r, n, mu, sigma) {
  R <- matrix(rnorm(r*n),r,n)
  iter <- 0

  importance.for.simplex <- function(x) {
    assign("iter",iter+1,parent.env(environment()))
    adjm <- x[1]
    adjs <- abs(x[2]+1e-10)  ## avoid zero
    y <- rel.sd.importance(R, mu, sigma, adjm, adjs)
    print(paste("(",adjm,",",adjs,") = ",y,sep=""))
    return(y)
```

```
  }
  simplex(importance.for.simplex, c(mu, sigma), 1e-6);
}
```

(c) Using the `simple615.h` and `normMix615.h`, implement the Simplex search version of this algorithm in C++. You need to use the Mersenne-Twister algorithm implemented in `boost` algorithm to generate random variables. Below are example runs.

```
% Usage: ./hw-5-1 [r] [n] [mu] [sigma] [seed (optional)]

% time ./hw-5-3c 100 1000 -8 1
Relative SD is minimized at (-7.20852,0.982308) to 0.0255633 with 93 function calls
0:02.04 elapsed, 2.046 u, 0.000 s, cpu 100.0%, 0 swaps, 0 rds, 0 wrts, pgs: 0 avg., 0 max.
```