C++ Functions in Maxliklib Library

Shelby J. Haberman
Haberman Statistics

Abstract

The functions in the maxliklib repository are described. Arguments and their definitions are specified, and dependencies of functions are stated.

Keywords: Maximization procedures, quadrature procedures, maximum likelihood

The maxliklib repository consists of C++ functions helpful in estimation related to maximum likelihood. The functions should be appropriate for C++11. Except if otherwise stated, functions rely on the Armadillo library (Sanderson & Curtin, 2016, 2018) at http://arma.sourceforge.net. The following functions are found in the library:

- adapt.cpp
- adaptv.cpp
- cloglog.cpp
- cloglog1.cpp
- conjgrad.cpp
- cumlogit.cpp
- cumlogit1.cpp
- genfact.cpp
- genprods.cpp

Shelby J. Haberman https://orcid.org/0000-0002-5490-0405
Shelby Haberman is an independent statistical consultant whose website is https://www.habermanstatistics.com. He can also be reached at Barak 3/1, Jerusalem 9350276, Israel or at haberman.statistics@gmail.com.

- gradascent.cpp
- hermcoeff.cpp
- hermpoly.cpp
- hermpw.cpp
- lw.cpp
- lwm.cpp
- maxf1vvar.cpp
- maxf2vvar.cpp
- maxlin2.cpp
- maxlin.cpp
- modit.cpp
- nrv.cpp
- rebound.cpp

Distributions of Sums of Independent Multinomial Variables

The functions in this section implement a modified and generalized version of the Lord-Wingersky algorithm (Lord & Wingersky, 1984; Thissen et al., 1995). The numerical procedures and their rationale are discussed in lw.pdf.

lw.cpp

The function lw.cpp finds the probability mass function of the sum S of mutually independent Bernoulli random variables X_j, $0 \leq j < n$. The function declaration is

vec lw(double & c,vec & p).

The vector p has dimension n and has positive elements that are less than 1. For $0 \leq j < n$, the probability that $X_j = 1$ is element j of p. The variable c is normally a small positive number used as in lw.pdf to remove very small probabilities from consideration in order to speed computation. If c is not positive, then the modified Lord-Wingersky algorithm used by lw.cpp reduces to the conventional algorithm. The probability mass function is provided by *lw*, a vector with $n + 1$ elements. For $0 \leq k \leq n$, element k of *lw* is the probability that $S = k$.

lwm.cpp

The function lwm.cpp finds the probability mass function of the sum S of n mutually independent random variables X_j, $0 \leq j <$ with integer values from 0 to $I_j - 1$ for an integer $I_j > 1$. The function declaration is

vec lwm(double & cc,int & n,vec p[ ]).

The array p of vectors has n members. For $0 \leq j < n$, member j of p is the vector *p[j]* with I_j nonnegative elements. The sum of these elements is 1, and element k, $0 \leq k < I_j$, of *p[j]* is the probability that $X_j = k$. The probability mass function is provided by *lwm*, a vector with $K = 1 + \sum_{j=1}^{n}(I_j - 1)$ elements. Element k of *lwm*, $0 \leq k < K$, is the probability that $S = k$. The variable c is normally a small positive number used as in lw.pdf to remove very small probabilities from consideration in order to speed computation. If c is not positive, then the modified algorithm used by lwm.cpp reduces to the conventional generalization of the Lord-Wingersky algorithm to sums of independent multinomial variables.

Tools for Line Searches

The functions in this section facilitate line searches during function maximization. Throughout discussions in this section and in Functions related to the Newton-Raphson algorithm and Functions Related to Gradient Methods, the theoretical background is provided in convergence.pdf. For some positive integer p and nonempty open convex set O of p-dimensional vectors, a continuously differentiable real function *f.value* on O is to be maximized by an iterative algorithm with a starting value in O. It is assumed that, for some real a, the set A of members of O at which *f.value* is at least a is closed and bounded, and the sets A_0 of members of O at which *f.value* exceeds a is nonempty. The function *f.value* is assumed to be strictly pseudoconcave on A_0. The starting values for algorithms are assumed to be in A_0. The convention is adopted that *f.value* has value *NaN* at any p-dimensional vector not in O.

modit.cpp

The function modit.cpp truncates an iteration to conform to limits on step size and bounds in the case of a real function of one variable with a unique critical point and a limit of $-\infty$ as the absolute value of the function argument approaches ∞. The function declaration is

double modit(const double & eta, const double & alpha0, const double & alpha1,
const double & stepmax, const double & lower, const double & upper).

Here *eta* is a positive multiplier less than 1, *alpha0* is the previous location, *alpha1* is the proposed new location, *stepmax* is the positive limit on step size, *lower* is the lower bound, and *upper* is the upper bound. It is assumed that *alpha0* and *alpha1* are different. The function returns a value *modit* that is normally *alpha1*; however, if *alpha1* exceeds *alpha0*, then *modit* is truncated above so that it does not exceed the minimum of *alpha0+stepmax* and *alpha0+eta(upper-alpha0)*, while if *alpha1* is less than *alpha0*, then *modit* is truncated below so that it is at least the maximum of *alpha0-stepmax* and *alpha0+eta(lower-alpha0)*.

rebound.cpp

The function rebound.cpp updates the lower and upper bounds for maximization of a differentiable real function on the real line with a unique critical point and a limit of $-\infty$ as the absolute value of the function argument approaches ∞. The function declaration is

void rebound(const double & y,const double & der,double & lower,double & upper).

Here y is the current location, *der* is the function derivative at y, *lower* is the lower bound, and *upper* is the upper bound. It is assumed that *der* is not 0. If *der* is positive, *lower* is changed to y. If *der* is negative, *upper* is changed to y.

Functions related to the Newton-Raphson algorithm

In this section, functions are discussed that are related to the Newton-Raphson algorithm. It should be noted that references to function values, gradients, and Hessian matrices do not address computational methods. In fact, the function values, gradients, and Hessian matrices employed may be approximations derived by numerical differentiation or large-sample approximations. In this section, *f.value* is assumed to be twice continuously differentiable.

maxlin2.cpp

The function maxlin2.cpp performs a line seach based on the Newton-Raphson algorithm. The gradient of *f.value* is *f.grad*. The Hessian matrix of *f.value* is *f.hess*. The function declaration is

maxf2v maxlin2(const paramnr & nrparams, const vec & v,
maxf2v & vary0, function<f2v(vec)>f).

The struct *maxf2v* has the definition

struct maxf2v{vec locmax; double max; vec grad; mat hess;};.

The struct *paramnr* has the definition

struct paramnr{int maxit; int maxits; double eta; double gamma1; double gamma2; double kappa; double tol;};.

The struct *f2v* has the definition

struct f2v{double value; vec grad;};.

The gradient of *f.value* is *f.grad*. The Hessian matrix is *f.hess*. Parameters used are defined in *nrparams*. The maximum number of primary iterations used in *nrv.cpp* is *nparams.maxit*. The maximum number of secondary iterations per main iteration used in the line search is *nrparams.maxits*. The maximum fraction of a step toward a boundary is *nrparams.eta*. For secondary iterations, the improvement check is *nrparams.gamma1*<1. The value of *nparams.gamma2* is used in *nrv.cpp* to ensure that the direction of *v* is satisfactory. The largest permitted step length is *nrparams.kappa*>0. The convergence criterion for primary iterations is *nparams.tol*.

Information concerning the starting point of the line search is in *vy0*, while *v* provides the starting direction. The initial location is *vy0.locmax*, the value of *f.value* at *vy0.locmax* is *vy0.max*, the gradient of *f.value* at *vy0.locmax* is *vy0.grad*, and the Hessian matrix of *f.value* at *vy0.locmax* is *vy0.hess*. It is assumed that the inner product of *v* and *vy0.grad* is positive. The returned value includes the approximate location *maxlin2.locmax* of the maximum value of *f.value* at a point on the ray with origin *vy0.locmax* that has direction *v*. In addition, *maxlin2.max* is the value of *f.value* at *maxlin2.locmax*, *maxlin2.grad* is the gradient of *f.value* at *maxlin2.locmax*, and *maxlin2.hess* is the Hessian matrix of *f.value* at *maxlin2.locmax*.

The function *maxlin2.cpp* uses *maxf2vvar.cpp*, *modit.cpp*, and *rebound.cpp*.

maxf2vvar.cpp

The function maxf2vvar.cpp is used to combine information on a location and on a functions value, gradient, and Hessian matrix at the location. The function maxf2vvar.cpp has declaration

maxf2v maxf2vvar(const vec & y,const f2v & fy);.

The structs *f2v* and *maxf2v* are defined as in maxlin2.cpp. The returned value *maxf2vvar.locmax* is *y*, while *maxf2vvar.max* is the value of *f.value* at *y*, *maxf2var.gad* is the gradient of *f.value* at *y*, and *maxf2var.hess* is the Hessian matrix of *f.value* at *y*.

nrv.cpp

The function nrv.cpp applies a modified version of the Newton-Raphson algorithm to maximization of *f.value*. The function nrv.cpp has declaration

maxf2v nrv(const paramnr & nrparams, const vec & start, function<f2v(vec)> f).

The structs *f2v*, *maxf2v*, and *paramnr* are defined as in maxlin2.cpp and maxf2vvar.cpp. In convergence.pdf, γ_1 corresponds to *nparams.gamma1*, γ_2 corresponds to *nrparams.gamma2*, and κ corresponds to *nrparams.kappa*. The starting vector *start* must be in O. Iterations cease once the value of *f.value* increases by less than *nrparams.tol* after a primary iteration.

The function nrv.cpp uses maxf2vvar.cpp, maxlin2.cpp, modit.cpp, and rebound.cpp.

Functions Related to Gradient Methods

In this section, functions are considered based on gradient-based methods.

conjgrad.cpp

The function *conjgrad.cpp* implements a conjugate gradient algorithm for maximization of *f.value*. The function declaration is

maxf1v conjgrad(const paramga & gaparams,
const vec & start, const function<f1v(vec)> f).

The definition of *maxf1v* is

struct maxf1v{vec locmax; double max; vec grad;};.

The definition of *paramga* is

struct paramga{int maxit; int maxits; function<double(vec)> c; double eta;
double gamma1; double gamma2; double kappa; double tol;}.

The definition of *f1v* is

struct f1v{double value; vec grad;};.

The starting vector is *start*. The maximum number of main iterations is *gaparams.maxit*. The maximum number of secondary iterations per main iteration is *gaparams.maxits*. The function *gaparams.c* specifies the step size for numerical differentiation of the gradient *f.grad* of *f.value*. It must be the case that, for any vector

y in the domain of *f.value*, *y+c(y)* is also in that domain. The maximum fraction of a step toward a boundary is *gaparams.eta*. In convergence.pdf, γ_1 corresponds to *gaparams.gamma1*, γ_2 corresponds to *gaparams.gamma2*, and κ corresponds to *gaparams.kappa*. Iterations cease once the value of *f.value* increases by less than *gaparams.tol* after a primary iteration.

The function conjgrad.cpp uses maxf1vvar.cpp, maxlin.cpp, modit.cpp, and rebound.cpp.

gradascent.cpp

The function gradascent.cpp uses a gradient-ascent algorithm for maximization of *f.value*. The function declaration for gradient.cpp is

maxf1v gradascent(const paramga & gaparams,
const vec & start, const function<f1v(vec)> f).

The definitions of *gaparams*, *paramga*, *f1v*, and *f* are the same as in conjgrad.cpp, and maxf1vvar.cpp, maxlin.cpp, modit.cpp, and rebound.cpp are used.

maxf1vvar.cpp

The function maxf1vvar.cpp is used to combine information on a location and on a functions value and gradient at the location. The function maxf1vvar.cpp has declaration

maxf1v maxf1vvar(const vec & y,const f1v & fy).

The structs *f1v* and *maxf1v* are defined as in conjgrad.cpp. The returned value *maxf1vvar.locmax* is *y*, while *maxf1vvar.max* is the value of *f.value* at *y* and *maxf1var.gad* is the gradient of *f.value* at *y*.

maxlin.cpp

The function *maxlin.cpp* uses numerical differentiation to provide an approximate Newton-Raphson procedure for a line search. The function declaration is

maxf1v maxlin(const paramga & gaparams, const vec & v,
maxf1v & vary0, function<f1v(vec)>f).

The structs *maxf1v*, *paramga*, and *f1v* are defined as in conjgrad.cpp. The functions maxf1vvar.cpp, modit.cpp, and rebound.cpp are all used.

Log-likelihood Components

In this section, components of log-likelihood functions are provided. These components involve the contribution to the log likelihood of an individual predicted random variable Y_i in a nonempty set $\mathcal{Y}_i$ and a q_i by p predicting matrix $\mathbf{X}_i$ in a nonempty set $\mathcal{X}_i$, where the nonnegative integer i is less than the positive integer n. The log-likelihood function under study has the form

$$\ell(\gamma) = \sum_{i=0}^{n-1} w_i \ell_i(\mathbf{X}_i \boldsymbol{\gamma}; Y_i) \tag{1}$$

for $\boldsymbol{\gamma}$ in O. For a nonempty open convex set O_i of q_i-dimensional vectors, $\ell_i(\cdot; y)$ is a twice continuously differentiable real function on O_i for all y in $\mathcal{Y}_i$. For any $\boldsymbol{\gamma}$ in O and $\mathbf{X}$ in $\mathcal{X}_i$, $\mathbf{X}\boldsymbol{\gamma}$ is in O_i. If $\mathcal{Y}_i$ is finite or countably infinite and $\boldsymbol{\beta}$ is in O_i, then 1 is the sum of the $\exp(\ell_i(\boldsymbol{\beta}; y))$ over y in $\mathcal{Y}_i$ and some random variable Y equals y with probability $\exp(\ell_i(\boldsymbol{\beta}; y))$ for each y in $\mathcal{Y}_i$. If $\mathcal{Y}_i$ is a real interval that includes a nonempty open set or a convex set with a nonempty interior and $\boldsymbol{\beta}$ is in O_i, then the integral of $\exp(\ell_i(\boldsymbol{\beta}; y))$ over y in $\mathcal{Y}_i$ is 1 and a continuous random variable Y has density $\exp(\ell_i(\boldsymbol{\beta}; y))$ at y in $\mathcal{Y}_i$. The gradient function of $\ell_i(\cdot; y)$ is $\nabla \ell_i(\cdot; y)$ and corresponding Hessian matrix is $\nabla^2 \ell_i(\cdot; y)$. It follows that the gradient of ℓ at $\boldsymbol{\gamma}$ is

$$\nabla \ell(\boldsymbol{\gamma}) = \sum_{i=0}^{n-1} w_i \mathbf{X}_i^T \nabla \ell_i(\mathbf{X}_i \boldsymbol{\gamma}; Y_i), \tag{2}$$

and the Hessian matrix of ℓ at γ is

$$\nabla^2 \ell(\gamma) = \sum_{i=0}^{n-1} w_i \mathbf{X}_i^T \nabla^2 \ell_i(\mathbf{X}_i \gamma; Y_i) \mathbf{X}_i. \tag{3}$$

Many standard cases of $\ell_i(\cdot; y)$ exist. Many are examined in the literature on generalized linear models McCullagh and Nelder (1989). The following C++ functions are employed for common examples. The structs *f1v* and *f2v* are defined as in maxlin2.cpp and maxlin.cpp.

cloglog.cpp

The function cloglog.cpp computes the function value, gradient, and Hessian matrix associated with a complementary log-log transformation. In this case, $\mathcal{Y}_i$ is the set $\{0, 1\}$, $q_i = 1$, O_i is the set of all one-dimensional vectors, $\ell_i(\boldsymbol{\beta}; 0) = -\exp(\beta(0))$, and $\ell_i(\boldsymbol{\beta}; 1) = \log(1 - \exp(-\exp(\beta(0)))$. The function declaration is

f2v cloglog(int y, vec beta).

The function *cloglog.value* is $\ell_i(\boldsymbol{\beta}; y)$ if y is y and *beta* is $\boldsymbol{\beta}$.

cloglog1.cpp

The function cloglog1.cpp computes the function value and gradient associated with a complementary log-log transformation. As in cloglog.cpp, $\mathcal{Y}_i$ is the set $\{0, 1\}$, $q_i = 1$, O_i is the set of all one-dimensional vectors, $\ell_i(\boldsymbol{\beta}; 0) = -\exp(\beta(0))$, and $\ell_i(\boldsymbol{\beta}; 1) = \log(1 - \exp(-\exp(\beta(0)))$. The function declaration is

f1v cloglog1(int y, vec beta).

The function *cloglog1.value* is $\ell_i(\boldsymbol{\beta}; y)$ if y is y and *beta* is $\boldsymbol{\beta}$.

cumlogit.cpp

The function cumlogit.cpp computes the function value, gradient, and Hessian matrix associated with a cumulative logit transformation. In this case, $\mathcal{Y}_i$ is the set of nonnegative integers less than n for an integer $n > 1$, $q_i = n - 1$, O_i is the set of all vectors of dimension $n - 1$,

$$\ell_i(\boldsymbol{\beta}; y) = \begin{cases} -\beta(y) - \sum_{i=0}^{y} \log(1 + \exp(-\beta(i))), & 0 \leq y < n - 1, \\ -\sum_{i=0}^{n-2} \log(1 + \exp(-\beta(i))), & y = n - 1. \end{cases} \tag{4}$$

The function declaration is

f2v cumlogit(int y, vec beta).

The function *cumlogit.value* is $\ell_i(\boldsymbol{\beta}; y)$ if y is y and *beta* is $\boldsymbol{\beta}$.

cumlogit1.cpp

The function cumlogit1.cpp computes the function value and gradient associated with a cumulative logit transformation. As in cumlogit.cpp, $\mathcal{Y}_i$ is the set of nonnegative integers less than the positive integer n, $q_i = n - 1$, O_i is the set of all vectors of dimension $n - 1$, and Equation 4 holds. The function declaration is

f1v cumlogit1(int y, vec beta).

The function *cumlogit1.value* is $\ell_i(\boldsymbol{\beta}; y)$ if y is y and *beta* is $\boldsymbol{\beta}$.

Integration Tools

The functions in this section aid in cases in which integration is required.

adapt.cpp

The function adapt.cpp provides a linear transformation of a set of real quadrature points and adjusts the correponding weights for each point. The linear transformation has the form $L(x) = a + bx$ for x real, where a is a real number and b is a positive real number. The linear transformation is applied to each quadrature point and the weights are multiplied by b. The function declaration is

pw adapt(double & loc, double & scale, pw & pws).

The struct *pw* has the definition

struct pw{vec points; vec weights;};.

The variable *loc* is a and the variable *scale* is b. The original points are provided by *pws.points*, and the original positive weights are given by *pws.weights*. The transformed points are *adapt.points*, and the transformed weights are *adapt.weights*. If *scale* is not positive, then *adapt* is set equal to *pws*. The number of elements in *pws.points*, *pws.weights*, *adapt.points*, and *adapt.weights* is the same.

adaptv.cpp

The function adaptv.cpp provides a linear transformation of a set of D-dimensional quadrature points and adjusts the correponding weights for each point, where D is a positive integer. The linear transformation has the form $L(\mathbf{x}) = \mathbf{a} + \mathbf{B}\mathbf{x}$ for the D-dimensional vector $\mathbf{x}$, where $\mathbf{a}$ is a D-dimensional vector and $\mathbf{B}$ is a D by D lower triangular matrix. The linear transformation is applied to each quadrature point and the weights are multiplied by the determinant of $\mathbf{B}$. The function declaration is

pwv adapt(vec & loc, mat & lt, pwv & pws).

The struct *pwv* has the definition

struct pwv{mat points; vec weights;};.

The variable *loc* is $\mathbf{a}$ and the variable *lt* is $\mathbf{B}$. The original points are provided by *pws.points*, and the original positive weights are in *pws.weights*. The transformed points are in *adaptv.points*, and the transformed weights are in *adaptv.weights*. If any diagonal element of *lt* is not positive, then *adaptv* is set equal to *pws*. The number of elements in *pws.weights* and *adaptv.weights* is the same and is the same as both the number of columns in *adaptv.points* and the number of columns in *pws.points*. The number of rows in *adaptv.points* is equal to the number of rows in *pws.points*.

genfact.cpp

For a vector *sizes* of positive integers, the function genfact.cpp generates all vectors *i* of nonnegative integers with the same number of elements as *sizes* such that each element of *i* is less than the corresponding element of *sizes*. The function declaration is

imat genfact(ivec & sizes).

The columns of *genfact* are the possible vectors *i*. For example, if the elements of *sizes* are 2 and 3, then Column 0 of *genfact* has elements 0 and 0, and Column 1 has elements 1 and 0. In all, *sizes* has 6 columns, and Column 5 has elements 1 and 2.

genprods.cpp

The function genprods.cpp generates a collection of quadrature points and quadrature weights for a multivariate integral from quadrature weights and quadrature points for a univariate integral. The function declaration is

pwv genprods(imat & indices,pw pws [ ]).

The struct *pw* is defined as in adapt.cpp, and the struct *pwv* is defined as in adaptv.cpp. Consider the case of Q quadrature points for a multidimensional integral on the space of D-dimensional vectors, where Q and D are positive integers. Then *genprods.points* has Q columns and *genprods.weights* has Q elements. The matrix *genprods.points* has D rows. The array *pws* has D members. For $0 \leq d < D$, *pws[d].points* and *pws[d].weights* have $m(d) > 1$ members, and the members of *pws[d].weights* are positive. The matrix *indices* specifies the quadrature vectors and quadrature weights to construct from *pws*. If *indices* has p columns, $0 \leq k < p$, and $0 \leq d < D$, then row d and column k of *indices* is nonnegative and less than $m(d)$ and the corresponding row and column of *genprods.points* is *pws[d].points(indices(d,k))* . Element k of *genprods.weights* is the product of *pws[d].weights(indices(d,k))* for $0 \leq d < D$.

hermcoeff.cpp

The function hermcoeff.cpp finds the coefficients of a Hermite polynomial of a given order. The function declaration is

vec hermcoeff(int & n).

The integer variable n is the nonnegative order. The vector *hermcoeff* has n+1 elements. The polynomial is $H_n(x) = \sum_{i=0}^{n} \alpha_i x^{n-i}$ for real x, and element i of

hermcoeff is α_i. For example, if n is 2, then the elements of *hermcoeff* are 1, 0, and -1.

hermpoly.cpp

The function hermpoly.cpp evaluates the Hermite polynomials up to a given order at a specified real value. The function declaration is

vec hermpoly(int &n, double & x).

The order is the nonnegative integer variable n, and the real value is x. The vector *hermpoly* has n+1 elements. For $0 \leq k \leq n$, element k of *hermpoly* is the value of H_k at x.

hermpw.cpp

The function hermpw.cpp uses the algorithm of Golub and Welsch (1969) to find the quadrature points and quadrature weights for Gauss-Hermite quadrature. The function declaration is

pw hermpw(int & n).

The struct *hermpw* has vector elements *hermpw.points* and *hermpw.weights*. The number of quadrature points is n. The ordered quadrature points are in *hermpw.points*. The corresponding weights are in *hermpw.weights*.

References

Golub, G. H., & Welsch, J. H. (1969). Calculation of gauss quadrature rules. *Mathematics of Computation*, *23*, 221–s10. https://doi.org/10.2307/2004418

Lord, F. M., & Wingersky, M. S. (1984). Comparison of IRT true-score and equipercentile observed-score "equatings". *Applied Psychological Measurement*, *8*, 453–461. https://doi.org/10.1177/014662168400800409

McCullagh, P., & Nelder, J. A. (1989). *Generalized linear models* (2nd ed.). Springer US. https://doi.org/10.1007/978-1-4899-3242-6

Sanderson, C., & Curtin, R. (2016). Armadillo: A template-based C++ library for linear algebra. *The Journal of Open Source Software*, *1*, 26. https://doi.org/10.21105/joss.00026

Sanderson, C., & Curtin, R. (2018). A user-friendly hybrid sparse matrix class in C++. *Mathematical software – ICMS 2018* (pp. 422–430). https://doi.org/10.1007/978-3-319-96418-8_50

Thissen, D., Pommerich, M., Billeaud, K., & Williams, V. S. L. (1995). Item response theory for scores on tests including polytomous items with ordered responses. *Applied Psychological Measurement*, *19*, 39–49. https://doi.org/10.1177/014662169501900105