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. They rely on the Armadillo library (Sanderson & Curtin, 2016, 2018) at http://arma.sourceforge.net. Unless otherwise noted, for the library members considered, it is assumed that users have verified that function arguments are valid. The following functions are found in the library.

- adapt.cpp
- adaptv.cpp
- berresp.cpp
- berresp1.cpp
- conjgrad.cpp
- contresp.cpp
- contresp1.cpp
- cumresp.cpp
- cumresp1.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.

- genfact.cpp
- genprods.cpp
- genresp.cpp
- genresp1.cpp
- genresplik.cpp
- genresplik1.cpp
- genresplikl.cpp
- genrespmle.cpp
- genrespmle1.cpp
- genrespmleg.cpp
- genrespmlel.cpp
- gradascent.cpp
- gradresp.cpp
- gradresp1.cpp
- gumbel.cpp
- gumbel1.cpp
- hermcoeff.cpp
- hermpoly.cpp
- hermpw.cpp
- logistic.cpp
- logistic1.cpp
- loglog.cpp
- loglog1.cpp
- logit.cpp
- logit1.cpp

- logmean.cpp
- logmean1.cpp
- lw.cpp
- lwm.cpp
- maxberresp.cpp
- maxberresp1.cpp
- maxf1vvar.cpp
- maxf2vvar.cpp
- maxlinq.cpp
- maxlinq2.cpp
- maxquad.cpp
- multlogit.cpp
- multlogit1.cpp
- modit.cpp
- normal.cpp
- normal1.cpp
- normalv.cpp
- normalv1.cpp
- nrv.cpp
- pack.cpp
- probit.cpp
- probit1.cpp
- ranklogit.cpp
- ranklogit1.cpp
- rebound.cpp

- truncresp.cpp
- truncresp1.cpp
- unpack.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(const double & c, const 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(const double & c, const vector<vec> & p).

Here p 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 and the definitions of η, γ_1, γ_2, and κ are found 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.

maxlinq.cpp

The function *maxlinq.cpp* provides a line search appropriate for algorithms that only use function values and gradients. The function declaration is

maxf1v maxlinq(const params & mparams, const vec & v,
const maxf1v & vary0, const std::function<f1v(vec &)>f).

Here the definition of *maxf1v* is

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

vary0.locmax is the starting vector for the line search, *vary0.max* is the value of *f.value* at the starting vector, and *maxlinq.grad* is the gradient of *f.value* at *vary0.locmax*, while *maxlinq.locmax* is the approximation location of the maximum of *f.value* on the half-line that starts at *vary0.locmax* and has direction *v*, *maxlinq.max* is the approximate maximum of *f.value* on the half-line, and *maxlinq.grad* is the gradient of *f.value* at *maxlinq.locmax*.

The definition of *params* is

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

Here *mparams.maxit* is the number of primary iterations, *mparams.maxits* is the maximum number of uses of maxquad.cpp permitted for each primary iteration, *mparams.eta* is η, *mparams.gamma1* is γ_1, *mparams.gamma2* is γ_2, and *mparams.kappa* is κ. Iterations cease if the function value changes less than *mparams.tol* after a primary iteration.

The definition of *f1v* is

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

where *f.value* is the function value and *f.grad* is the gradient of *f.value*.

The functions maxf1vvar.cpp, maxquad.cpp, modit.cpp, and rebound.cpp are all used.

maxlinq2.cpp

The function maxlinq2.cpp performs the same line search as in maxlinq.cpp; however, Hessian matrices are also computed. The function declaration is

maxf2v maxlinq2(const params & mparams, const vec & v,
const maxf2v & vary0, const std::function<f2v(vec &)>f).

The struct *maxf2v* has the definition

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

while the struct *f2v* has the definition

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

The Hessian matrix of *f.value* is *f.hess*, *vary0.hess* is the Hessian matrix of *f.value* at *vary0.locmax*, and *maxlinq2.hess* is the Hessian matrix of *f.value* at *maxlinq2.locmax*.

The function maxlinq2.cpp uses maxf2vvar.cpp, maxquad.cpp, modit.cpp, and rebound.cpp.

maxquad.cpp

The function maxquad.cpp approximates the maximum of *f.value* along a half-line by use of a quadratic two-point approximation. The function declaration is

double maxquad(const double & x0, const double & x1, const double & f0, const double & f1, const double & g0, const double & stepmax).

Here *x0* and *x1* are the points used, *f0* is the function value at *x0*, *f1* is the function value at *x1*, *g0* is the derivative at *x0*, and *stepmax* is the maximum change from *x0* permitted in the estimated location *maxquad* of the function maximum.

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 bounds & b),

and the struct *bounds* is defined as

struct bounds {double lower; double upper;}.

Here *eta* corresponds to η, *alpha0* is the previous location, *alpha1* is the proposed new location, *stepmax* is the positive limit on step size, *b.lower* is the lower bound, and *b.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(b.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(b.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

bounds rebound(const double & y, const double & der, const bounds & b).

The struct *bounds* is defined as in modit.cpp. Here *y* is the current location, *der* is the function derivative at *y*, *b.lower* is the current lower bound, and *b.upper* is the current upper bound. It is assumed that *der* is not 0. If *der* is positive, *modit.lower* is *y* and *modit.upper* is *b.upper*. If *der* is negative, *modit.upper* is *y* and *modit.lower* is *b.lower*.

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.

maxf2vvar.cpp

The function maxf2vvar.cpp is used to combine information on a location and on a function's 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 maxlinq2.cpp. The returned value *maxf2vvar.locmax* is *y*, while *maxf2vvar.max* is the value of *f.value* at *y*, *maxf2var.grad* 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 params & mparams, const vec & start, const std::function<f2v(vec &)> f).

The structs *f2v*, *maxf2v*, and *params* are defined as in maxlinq.cpp and maxlinq2.cpp. The starting vector *start* must be in O.

The function nrv.cpp uses maxf2vvar.cpp, maxlinq2.cpp, maxquad.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 params & mparams,
const vec & start, const std::function<f1v(vec &)> f).

The starting vector is *start*.

The function conjgrad.cpp uses maxf1vvar.cpp, maxlinq.cpp, maxquad.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 gradascent.cpp is

maxf1v gradascent(const params & mparams,
const vec & start, const std::function<f1v(vec &)> f).

The functions maxf1vvar.cpp, maxlinq.cpp, maxquad.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 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*.

Log-likelihood Components

In this section, components of log-likelihood functions are provided. For a positive integer n and an observation i, $0 \leq i < n$, positive integers r_i and q_i are given. The component of the log likelihood for observation i involves the predicted random vector $\mathbf{Y}_i$ in a nonempty subset $\mathcal{Y}_i$ of r_i-dimensional vectors with elements $Y_i(j)$, $0 \leq j < r_i$, the q_i by p predicting matrix $\mathbf{X}_i$ in a nonempty set $\mathcal{X}_i$, the q_i-dimensional vector $\mathbf{o}_i$, and the positive real weight w_i. If $\boldsymbol{\tau}$ is in O, then let $\boldsymbol{\lambda}_i(\boldsymbol{\tau}) = \mathbf{o}_i + \mathbf{X}_i\boldsymbol{\tau}$ for $0 \leq i < n$, and let the log-likelihood function under study have the form

$$\ell(\boldsymbol{\tau}) = \sum_{i=0}^{n-1} w_i \ell_i(\boldsymbol{\lambda}_i(\boldsymbol{\tau}); \mathbf{Y}_i). \tag{1}$$

Consider observation i for $0 \leq i < n$. For a nonempty open convex set O_i of q_i-dimensional vectors, $\ell_i(\cdot; \mathbf{y})$ is a twice continuously differentiable real function on O_i for all $\mathbf{y}$ in $\mathcal{Y}_i$. For any $\boldsymbol{\tau}$ in O and $\mathbf{X}$ in $\mathcal{X}_i$, $\boldsymbol{\lambda}_i(\boldsymbol{\tau})$ 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 $\exp(\ell_i(\boldsymbol{\beta}; \mathbf{y}))$ over $\mathbf{y}$ in $\mathcal{Y}_i$ and some random vector $\mathbf{Y}$ equals $\mathbf{y}$ with probability $\exp(\ell_i(\boldsymbol{\beta}; \mathbf{y}))$ for each $\mathbf{y}$ in $\mathcal{Y}_i$. If $\mathcal{Y}_i$ is a convex set with a nonempty interior and $\boldsymbol{\beta}$ is in O_i, then the integral of $\exp(\ell_i(\boldsymbol{\tau}; \mathbf{y}))$ over $\mathbf{y}$ in $\mathcal{Y}_i$ is 1 and a continuous random vector $\mathbf{Y}_i$ has density $\exp(\ell_i(\boldsymbol{\beta}; \mathbf{y}))$ at $\mathbf{y}$ in $\mathcal{Y}_i$. In some cases involving censorship, more complex structures arise. The gradient function of $\ell_i(\cdot;\mathbf{y})$ is $\nabla\ell_i(\cdot;\mathbf{y})$ and the corresponding Hessian matrix is $\nabla^2\ell_i(\cdot;\mathbf{y})$. It follows that the gradient of ℓ at $\boldsymbol{\tau}$ in O is

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

and the Hessian matrix of ℓ at $\boldsymbol{\tau}$ is

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

The Hessian matrix $\nabla^2\ell(\boldsymbol{\tau})$ has an approximation

$$\tilde{\nabla}^2\ell(\boldsymbol{\tau}) = -\sum_{i=0}^{n-1} w_i \mathbf{X}_i^T \nabla\ell_i(\boldsymbol{\lambda}_i(\boldsymbol{\tau});\mathbf{Y}_i)[\nabla\ell_i(\boldsymbol{\lambda}_i(\boldsymbol{\tau});\mathbf{Y}_i)]^T\mathbf{X}_i \tag{4}$$

(Haberman, 2013; Louis, 1982) .

Many standard cases of $\ell_i(\cdot;\mathbf{y})$ exist, some of which are examined in the literature on survival analysis (Cox, 1972; Kalbfleisch & Prentice, 2002), generalized linear models (McCullagh & Nelder, 1989), multivariate analysis (Anderson, 2003), and discrete choice (McFadden, 1973). It should be noted that names for models are somewhat variable in different references, especially for graded and cumulative cases. In addition, graded and cumulative cases are defined to be consistent with the Bernoulli cases. The following C++ functions are employed for common examples. The structs *f1v* and *f2v* are defined as in maxlinq.cpp and maxlinq2.cpp. If the argument *beta* is not in O_i, then all values returned equal *NaN*. It is assumed that the user of the function has verified that the input vector y is in $\mathcal{Y}_i$. In the cases under study in this section, unless otherwise stated, the components are strictly concave, so that ℓ is strictly concave whenever $\mathbf{X}_i$, $0 \le i < n$, spans a space of dimension p. Conditions for a unique $\hat{\boldsymbol{\tau}}$ in O such that $\ell(\hat{\boldsymbol{\tau}})$ equals the supremum of ℓ over O are relatively complex (Haberman, 1974, 1977, 1980). It is worth noting that in cases in which $\hat{\boldsymbol{\tau}}$ in O satisfies the conditions that $\nabla\ell(\hat{\boldsymbol{\tau}})$ is the p-dimensional vector $\mathbf{0}_p$ with all elements 0 and $\nabla^2\ell(\hat{\boldsymbol{\tau}})$ is negative definite, then O can be restricted to ensure that ℓ is strictly concave on O and $\hat{\boldsymbol{\tau}}$ is the only member of O such that $\ell(\hat{\boldsymbol{\tau}})$ equals the supremum of ℓ on O and, for $\boldsymbol{\tau}$ in O, $\nabla\ell(\boldsymbol{\tau})$ is only the vector with all elements 0 if $\boldsymbol{\beta}$ equals $\hat{\boldsymbol{\beta}}$.

berresp.cpp

The function berresp.cpp is used to handle standard models for Bernoulli random variables. If this choice applies to observation i, $\mathcal{Y}_i$ is the set of one-dimensional vectors $\mathbf{y}$ with $y(0)$ equal 0 or 1, $r_i = 1$, $q_i = 1$, O_i is the set of all one-dimensional vectors, and F is a three-times continuously differentiable cumulative distribution function with a positive derivative f such that $\log(f)$ has a negative second derivative. For y in $\mathcal{Y}_i$ and $\boldsymbol{\beta}$ in O_i,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = \begin{cases} \log(F(\beta(0)), & y(0)=1, \\ \log(1-F(\beta(0))), & y(0)=0. \end{cases} \tag{5}$$

The function declaration is

f2v berresp(const char & transform, const ivec & y, const vec & beta),

so that the value, gradient, and Hessian matrix of $\ell_i(\cot;\mathbf{y})$ at $\boldsymbol{\beta}$ is found. If *transform* is G, then $F=G$, the standard Gumbel distribution function with value $G(y) = \exp(-\exp(-y))$ for y real. If *transform* is L, then $F=\Psi$, the standard logistic distribution function with value $\Psi(y) = 1/[1+\exp(-y)]$ for y real. If *transform* is N, then $F=\Phi$, the standard normal distribution function with derivative $\phi(y) = \exp(-y^2/2)/(2\pi)^{1/2}$ for real y. The function *berresp.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

The function berresp.cpp requires loglog.cpp, logit.cpp, and probit.cpp.

berresp1.cpp

The function berresp1.cpp corresponds to berresp.cpp; however, only the value and gradient of $\ell_i(\cdot;\mathbf{y})$ at $\boldsymbol{\beta}$ are found. Definitions of $\mathcal{Y}_i$, r_i, q_i, O_i, F, and $\ell_i(\cdot;\mathbf{y})$ are the same as in berresp.cpp. The function declaration is

f1v berresp(const char & transform, const ivec & y, const vec & beta).

The definitions of *transform*, *y*, and *beta* are the same as in berresp.cpp.

The function berresp1.cpp requires loglog1.cpp, logit1.cpp, and probit1.cpp.

contresp.cpp

The function contresp.cpp computes the function value, gradient, and Hessian matrix associated with the distribution of a location and scale model for a continuous random vector. Four cases are considered. In the first three cases, $r_i=1$, $q_i=2$, $\mathcal{Y}_i$ is the set of all one-dimensional vectors, O_i is the set of all two-dimensional vectors $\boldsymbol{\beta}$ with element $\beta(1)>0$, and F and f are defined as in berresp.cpp. For $\mathbf{y}$ in $\mathcal{Y}_i$ and $\boldsymbol{\beta}$ in O_i,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = \log(\beta(1)) + \log(f(\beta(0)+\beta(1)y(0))). \tag{6}$$

These cases correspond to a model that a random variable has a distribution $\beta(0)+\beta(1)Z$, where Z has a distribution function F. Here $\ell_i(\cdot;\mathbf{y})$ is concave, and the function is strictly concave if $y(0)$ is not 0.

The fourth case is somewhat more complex and is only applied to location and scale models for a multivariate normal case. Here r_i is an integer greater than 1, $q_i = r_i(r_i+3)/2$, $\mathcal{Y}_i$ consists of all r_i-dimensional real vectors, and O_i is the set of q_i-dimensional vectors $\boldsymbol{\beta}$ with elements β_h, $0\le h<q_i$ such that $\beta_h>0$ if $h=$

$r_i + j(j+3)/2$ and $0 \le j < r_i$. For such $\boldsymbol{\beta}$, let $\mathbf{a}(\boldsymbol{\beta})$ be the r_i-dimensional vector with elements $a_j(\boldsymbol{\beta}) = \beta_j$ for $0 \le j < r_i$, and let $\mathbf{B}(\boldsymbol{\beta})$ be the lower-triangular r_i by r_i matrx with row j and column k equal to β_h if $0 \le k \le j < r_i$ and $h = r_i + k + (j(j+1)/2$. For an r_i-dimensional vector $\mathbf{z}$ with elements z_j, $0 \le j < q_i$, let $\phi(\mathbf{z}; r_i)$ be the product of the $\phi(z_j)$, $0 \le j < r_i$. For $\mathbf{y}$ in $\mathcal{Y}_i$,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \left[\sum_{j=0}^{r_i-1} \log(\beta(j))\right] + \log(\phi(\mathbf{a}(\boldsymbol{\beta}) + \mathbf{B}(\boldsymbol{\beta})\mathbf{y}; r_i)). \tag{7}$$

This case corresponds to a model that a random vector has a distribution $\mathbf{a}(\boldsymbol{\beta}) + \mathbf{B}(\boldsymbol{\beta})\mathbf{Z}$, where $\mathbf{Z}$ is an r_i-dimensional multivariate normal random vector with zero mean and with covariance matrix equal to the identity matrix. The function $\ell_i(\cdot; \mathbf{y})$ is always concave but is not strictly concave.

For all cases, the function declaration is

f2v contresp(const char & transform, const vec & y, const vec & beta).

The variable *transform* is defined as in berresp.cpp. The one complication is that for *transform* equal N and y with dimension $r_i > 1$, 7 is used. The function *contresp.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

The function contresp.cpp requires gumbel.cpp, logistic.cpp, normal.cpp, normalv.cpp, and unpack.cpp.

contresp1.cpp

The function contresp1.cpp corresponds to contresp.cpp; however, only the value and gradient of $\ell_i(\cdot; \mathbf{y})$ at $\boldsymbol{\beta}$ are found. Definitions of $\mathcal{Y}_i$, r_i, q_i, O_i, F, and $\ell_i(\cdot; \mathbf{y})$ are the same as in contresp.cpp. The function declaration is

f1v contresp(const char & transform, const vec & y, const vec & beta).

The definitions of *transform*, y, and *beta* are the same as in contresp.cpp.

The function contresp1.cpp requires gumbel1.cpp, logistic1.cpp, normal1.cpp, normalv1.cpp, and unpack.cpp.

cumresp.cpp

The function cumresp.cpp computes the function value, gradient, and Hessian matrix associated with a cumulative response transformation. Here $r_i = 1$, $q_i \ge 1$, $\mathcal{Y}_i$ is the set of one-dimensional vectors $\mathbf{y}$ such that $y(0)$ is a nonnegative integer no greater than q_i, $q_i = n_i - 1$, O_i is the set of all vectors of dimension q_i, and F is defined as in berresp.cpp. For $\boldsymbol{\beta}$ in O_i and $\mathbf{y}$ in $\mathcal{Y}_i$,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = \begin{cases} \log(1-F(\beta(y(0))), & y(0)=0, \\ \log(1-F(\beta(y(0)))) + \sum_{i=0}^{y(0)-1}\log(F(\beta(i))), & 0<y(0)<q_i, \\ \sum_{i=0}^{y(0)-1}\log(F(\beta(i))), & y(0)=q_i. \end{cases} \tag{8}$$

The function declaration is

f2v cumresp(const char & transform, const ivec & y, const vec & beta).

Here *transform* is defined as in berresp.cpp. The function *cumresp.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. The function cumresp.cpp requires berresp.cpp, loglog.cpp, logit.cpp, and probit.cpp. If $r_i = 1$, then use of cumresp.cpp is equivalent to use of berresp.cpp. In general, $\ell_i(\cdot;\mathbf{y})$ is concave. Strict concavity holds if $q_i - y(0)$ does not exceed 1.

cumresp1.cpp

The function cumresp1.cpp computes the function value and gradient associated with a cumulative response transformation. Definitions of r_i, q_i, O_i, $\mathcal{Y}_i$, F, and $\ell_i(\boldsymbol{\beta};\mathbf{y})$ are the same as in cumresp.cpp. The function declaration is

f1v cumresp1(const char & transform, const ivec & y, const vec & beta).

Here *transform* is defined as in berresp.cpp. The function *cumtresp1.value* is $\ell_i(\boldsymbol{\beta};y)$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. The function cumresp1.cpp requires berresp1.cpp, loglog1.cpp, logit1.cpp, and probit1.cpp. If $r_i = 1$, then use of cumresp1.cpp is equivalent to use of berresp1.cpp.

gradresp.cpp

The function gradresp.cpp computes the function value, gradient, and Hessian matrix associated with a graded response transformation. Define F as in berresp.cpp. Then $r_i = 1$, $q_i \geq 1$, O_i is the set of all vectors of dimension q_i with strictly decreasing elements, $\mathcal{Y}_i$ is the set of one-dimensional vectors $\mathbf{y}$ with $y(0)$ a nonnegative integer no greater than q_i, and, for $\boldsymbol{\beta}$ in O_i and $\mathbf{y}$ in $\mathcal{Y}_i$,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = \begin{cases} \log(1-F(\beta(y(0))), & y(0)=0, \\ \log(F(\beta(y(0)-1)) - F(\beta(y(0)))), & 0<y(0)<q_i, \\ \log(F(\beta(y(0)-1))), & y(0)=q_i. \end{cases} \tag{9}$$

The function declaration is

f2v gradresp(const char & transform, const ivec & y, const vec & beta).

Here *transform* is defined as in berresp.cpp. The function *gradresp.value* is $\ell_i(\boldsymbol{\beta}; y)$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. If $q_i = 1$, then berresp.cpp, cumresp.cpp and gradresp.cpp yield the same result. The function $\ell_i(\cdot; \mathbf{y})$ is concave. Strict concavity only holds if q_i is 1 or q_i is 2 and $y(0) = 1$.

gradresp1.cpp

The function gradresp1.cpp computes the function value and gradient associated with a graded logit transformation. Here r_i, q_i, O_i, $\mathcal{Y}_i$, and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ are defined as in gradresp.cpp. The function declaration is

f1v gradresp1(const char & transform, const ivec & y, const vec & beta).

The function *gradresp1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. If $q_i = 1$, then berresp1.cpp, cumresp1.cpp and gradresp1.cpp yield the same result. If $q_i = 1$, then gradresp1.cpp and cumresp1.cpp yield the same result.

gumbel.cpp

The function gumbel.cpp provides the computations required in contresp.cpp for $\ell_i(\cdot; \mathbf{y})$ for the Gumbel case of $F = G$. The function declaration is

f2v gumbel(const vec & y, const vec & beta).

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

gumbel1.cpp

The function gumbel1.cpp computes the function value and gradient associated with the Gumbel case $F = G$ in contresp1.cpp. The function declaration is

f1v gumbel1(const vec & y, const vec & beta).

The function *gumbel1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

logistic.cpp

The function logistic.cpp computes the function value, gradient, and Hessian matrix associated with the logistic case $F = \Psi$ in contresp.cpp. The function declaration is

f2v logistic(const vec & y, const vec & beta).

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

logistic1.cpp

The function logistic1.cpp computes the function value and gradient associated with the logistic case $F = \Psi$ in contresp1.cpp. The function declaration is

f1v logistic1(const vec & y, const vec & beta).

The function *logistic1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

logit.cpp

The function logit.cpp computes the function value, gradient, and Hessian matrix associated with the logit case in berresp.cpp with $F = \Psi$. The function declaration is

f2v logit(const ivec & y, const vec & beta).

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

logit1.cpp

The function logit1.cpp computes the function value and gradient associated with the logit case in berresp1.cpp with $F = \Psi$. The function declaration is

f1v logit1(ivec & y, vec & beta).

The function *logit1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

loglog.cpp

The function loglog.cpp computes the function value, gradient, and Hessian matrix associated with the log-log case of berresp.cpp with $F = G$. The function declaration is

f2v loglog(const ivec & y, const vec & beta).

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

loglog1.cpp

The function loglog1.cpp computes the function value and gradient associated with the log-log case of berresp1.cpp with $F = G$. The function declaration is

f1v loglog1(const ivec & y, const vec & beta).

The function *loglog1.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

logmean.cpp

The function logmean.cpp computes the function value, gradient, and Hessian matrix associated with a log-mean transformation for a Poisson random variable. In this case, $r_i = 1$, $\mathcal{Y}_i$ is the set of one-dimensional vectors $\mathbf{y}$ such that $y(0)$ is a nonnegative integer, $q_i = 1$, and O_i is the set of all one-dimensional vectors. For $\boldsymbol{\beta}$ in O_i and $\mathbf{y}$ in $\mathcal{Y}_i$,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = y(0)\beta(0) - \exp(\beta(0)) - \log([y(0)]!). \tag{10}$$

The function declaration is

f2v logmean(const ivec & y, const vec & beta).

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

logmean1.cpp

The function logmean1.cpp computes the function value and gradient associated with a log-mean transformation for a Poisson random variable. Define r_i, $\mathcal{Y}_i$, q_i, O_i, v_i, and $\ell_i(\boldsymbol{\beta};\mathbf{y})$ as in logmean.cpp. The function declaration is

f1v logmean1(const ivec & y, const vec & beta).

The function *logmean1.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$.

maxberresp.cpp

The function maxberresp.cpp finds the log likelihood component, gradient, and Hessian matrix for the maximum of two unobserved Bernoulli random variables. The function F is defined as in berresp.cpp, $\mathcal{Y}_i$ is the set of one-dimensional vectors $\mathbf{y}$ with $y(0)$ equal 0 or 1, $r_i = 1$, $q_i = 2$, and O_i is the set of all two-dimensional vectors. For y in $\mathcal{Y}_i$ and $\boldsymbol{\beta}$ in O_i,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = \begin{cases} \log(F(\beta(0) + F(\beta(1) - F(\beta(0)F(\beta(1)), & y(0) = 1, \\ \log(1 - F(\beta(0))) + \log(1 - F(\beta(1))), & y(0) = 0. \end{cases} \tag{11}$$

It should be noted that

$$F(\beta(0) + F(\beta(1) - F(\beta(0)F(\beta(1)) = 1 - [1 - F(\beta(0))][1 - F(\beta(1)] \tag{12}$$

and

$$\log(1 - F(\beta(0)) + \log(1 - F(\beta(1)) = \log([1 - F(\beta(0)][1 - F(\beta(1)]). \tag{13}$$

The function $\ell_i(\cdot;\mathbf{y})$ is not necessarily concave if $y(0)=1$.

The function declaration is

f2v maxberresp(const char & transform, const ivec & y, const vec & beta).

The variables *transform*, *y*, and *beta* are defined as in berresp.cpp. The functions berresp.cpp, logit.cpp, loglog.cpp, and probit.cpp are required.

maxberresp1.cpp

The function maxberresp1.cpp finds the log likelihood component and gradient for the maximum of two unobserved Bernoulli random variables. Definitions of F, $\mathcal{Y}_i$, r_i, q_i, O_i, and $\ell_i(\cdot;\mathbf{y})$ are the same as in maxberresp.cpp. The function declaration is

f1v maxberresp(const char & transform, const ivec & y, const vec & beta).

The variables *transform*, *y*, and *beta* are defined as in berresp.cpp. The functions berresp1.cpp, logit1.cpp, loglog1.cpp, and probit1.cpp are required.

multlogit.cpp

The function multlogit.cpp computes the function value, gradient, and Hessian matrix associated with a multinomial logit transformation. In this case, $r_i=1$, $\mathcal{Y}_i$ is the set of one-dimensional vectors $\mathbf{y}$ such that $y(0)$ is a nonnegative integer no greater than $q_i \geq 1$, and O_i is the set of all q_i-dimensional vectors. For $\boldsymbol{\beta}$ in O_i and $\mathbf{y}$ in $\mathcal{Y}_i$,

$$\ell_i(\boldsymbol{\beta};\mathbf{y}) = \begin{cases} -\log\left(1+\sum_{k=0}^{q_i-1}\exp(\beta(k))\right), & y(0)=0, \\ \beta(y(0)-1)+\ell_i(\boldsymbol{\beta};\mathbf{0}_1), & y(0)>0. \end{cases} \tag{14}$$

The function declaration is

f2v multlogit(const ivec & y, const vec & beta).

The function *multlogit.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. If $q_i=1$, use of multlogit.cpp gives the same result as use of logit.cpp and as use of berresp.cpp, cumresp.cpp, or gradresp.cpp with *transform* equal L.

multlogit1.cpp

The function multlogit1.cpp computes the function value and gradient associated with a multinomial logit transformation. Define r_i, $\mathcal{Y}_i$, q_i, O_i, and $\ell_i(\boldsymbol{\beta};\mathbf{y})$ as in multlogit.cpp. The function declaration is

f1v multlogit1(const ivec & y, const vec & beta).

The function *multlogit1.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. If $q_i = 1$, use of multlogit1.cpp gives the same result as use of use of logit1.cpp or use of berresp1.cpp, cumresp1.cpp, or gradresp1.cpp with *transform* equal G.

normal.cpp

The function normal.cpp computes the function value, gradient, and Hessian matrix associated with the normal case in contresp.cpp if $\mathcal{Y}_i$ is the space of one-dimensional vectors and $F = \Phi$. The function declaration is

f2v normal(const vec & y, const vec & beta).

The function *normal.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ in contresp.cpp if y is $\mathbf{y}$, *beta* is $\boldsymbol{\beta}$, $\mathbf{y}$ has dimension 1, and $F = \Phi$.

normal1.cpp

The function normal1.cpp computes the function value and gradient associated with the log-likelihood component of contresp.cpp for $F = \Phi$. The function declaration is

f1v normal1(const vec & y, const vec & beta).

The function *normal1.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ in contresp.cpp if y is $\mathbf{y}$, *beta* is $\boldsymbol{\beta}$, $\mathbf{y}$ has dimension 1, and $F = \Phi$.

normalv.cpp

The function normalv.cpp computes the function value, gradient, and Hessian matrix associated with the log-likelihood component of contresp.cpp if $r_i > 1$.The function declaration is

f2v normalv(const vec & y, const vec & beta).

The function *normalv.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ in contresp.cpp if y is $\mathbf{y}$, *beta* is $\boldsymbol{\beta}$, and $\mathbf{y}$ has dimension greater than 1. The function normalv.cpp requires pack.cpp and unpack.cpp.

normalv1.cpp

The function normalv1.cpp computes the function value and gradient associated with the log-likelihood component of contresp.cpp if $r_i > 1$. The function declaration is

f1v normalv1(const vec & y, const vec & beta).

The function *normalv1.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$, *beta* is $\boldsymbol{\beta}$, and $r_i > 1$. The function normalv1.cpp requires pack.cpp and unpack.cpp.

pack.cpp

The function pack.cpp is used in contresp.cpp and contresp1.cpp to take an r_i-dimensional vector $\mathbf{a}$ and an r_i by r_i lower-triangular matrix $\mathbf{B}$ and convert the combination to the $\boldsymbol{\beta}$ in contresp.cpp such that $\mathbf{a}(\boldsymbol{\beta}) = \mathbf{a}$ and $\mathbf{B}(\boldsymbol{\beta}) = \mathbf{B}$. The vector $\mathbf{a}$ and the matrix $\mathbf{B}$ appear in the struct *vecmat* defined by

struct vecmax{vec v; mat m;};.

The function declaration is

vec pack(const vecmat & u).

If *u.v* is $\mathbf{a}$ and *u.m* is $\mathbf{B}$, then *pack* is the corresponding vector $\boldsymbol{\beta}$.

probit.cpp

The function probit.cpp computes the function value, gradient, and Hessian matrix associated with a probit transformation of berresp.cpp with $F = \Phi$. The function declaration is

f2v probit(const ivec & y, const vec & beta).

The function *probit.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$, *beta* is $\boldsymbol{\beta}$, and $F = \Phi$. Use of contresp.cpp, cumresp.cpp, or gradresp.cpp in the case of $q_i = 1$ and *transform* equal N gives the same result as use of probit.cpp.

probit1.cpp

The function probit1.cpp computes the function value and gradient associated with a probit transformation of contresp1.cpp with $F = \Phi$. The function declaration is

f1v probit1(const ivec & y, const vec & beta).

The function *probit1.value* is $\ell_i(\boldsymbol{\beta};\mathbf{y})$ if y is $\mathbf{y}$, *beta* is $\boldsymbol{\beta}$, and $F = \Phi$. If $r_i = 2$, use of berresp1.cpp, cumresp1.cpp, or gradresp1.cpp with *transform* equal N yields the same result as use of probit1.cpp.

ranklogit.cpp

The function ranklogit.cpp computes the function value, gradient, and Hessian matrix associated with a model for discrete choice in which $q_i + 1$ objects are ranked for some positive integer q_i and the r_i most preferred objects are recorded for some positive integer $r_i \le q_i$. The set $\mathcal{Y}_i$ consists of the vectors $\mathbf{y}$ of dimension r_i with distinct nonnegative integer elements that are no greater than q_i, and O_i is the set of all q_i-dimensional vectors. Let $\mathbf{0}_1$ be the one-dimensional vector with the single element 0. To describe the model, consider the standard Gumbel distribution function G defined in contresp.cpp. Consider $\boldsymbol{\beta}$ in O_i. Let U_j, $0 \le j \le q_i$, be independent random variables such that U_0 and $U_j - \beta_j$, $1 \le j \le q_i$, have the common distribution function G. Let $\mathbf{Y}$ be a random vector with values in $\mathcal{Y}_i$ such that $\mathbf{Y}$ is the member $\mathbf{y}$ of $\mathcal{Y}_i$ with elements y_j, $0 \le j < r_i$, if U_{y_j} is nonincreasing in j and $U_{y_j} \ge U_k$ if k is a nonnegative integer no greater than q_i that does not equal y_h for any nonnegative integer element $h < r_i$. For $\boldsymbol{\beta}$ in O_i and $\mathbf{y}$ in $\mathcal{Y}_i$, let $\boldsymbol{\alpha}(\boldsymbol{\beta})$ be the vector of dimension $q_i + 1$ such that element j, $0 \le j \le q_i$, is $\alpha_j(\boldsymbol{\beta}) = 0$ if $j = 0$ and $\alpha_j(\boldsymbol{\beta}) = \beta_{j-1}$ if $j > 0$. For $\mathbf{y}$ in $\mathcal{Y}_i$ and $0 \le j < r_i$, let $K_j(\mathbf{y})$ be the set of nonnegative integers no greater than q_i not equal to y_h for any nonnegative integer $h < j$. Thus $K_0(\mathbf{y})$ is the set of nonnegative integers no greater than q_i. Then the log-likelihood component is

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \sum_{j=0}^{r_i-1} \left[\alpha_{y_j}(\boldsymbol{\beta}) - \log \left(\sum_{h \in K_j(\mathbf{y})} \exp(\alpha_h(\boldsymbol{\beta})) \right) \right]. \tag{15}$$

The function declaration is

f2v ranklogit(const ivec & y, const vec & beta).

The function *ranklogit.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. If $r_i = 1$, use of ranklogit.cpp gives the same result as use of multlogit.cpp.

ranklogit1.cpp

The function ranklogit1.cpp computes the function value and gradient associated with the model for discrete choice of ranklogit.cpp. Define r_i, $\mathcal{Y}_i$, q_i, O_i, and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ as in ranklogit.cpp. The function declaration is

f1v ranklogit1(const ivec & y, const vec & beta).

The function *ranklogit1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is $\mathbf{y}$ and *beta* is $\boldsymbol{\beta}$. If $r_i = 1$, use of ranklogit1.cpp gives the same result as use of multlogit1.cpp.

truncresp.cpp

The function truncresp.cpp computes the function value, gradient, and Hessian matrix associated with a right-censored continuous random variable with the distribution of $\beta(0) + \beta(1)Z$ for some real $\beta(0)$ and positive real $\beta(1)$, where, as in contresp.cpp, Z has distribution function F equal to G, Ψ, or Φ. In this case, $r_i = 2$, $\mathcal{Y}_i$ consists of two dimensional vectors $\mathbf{y}$ such that $y(0)$ is a real number and $y(1)$ is 0 or 1, $q_i = 2$, and O_i is the set of all two-dimensional vectors $\boldsymbol{\beta}$ with element $\beta(1) > 0$. As in berresp.cpp, let f be the derivative of F. For $\boldsymbol{\beta}$ in O_i and $\mathbf{y}$ in $\mathcal{Y}_i$, if $y(1) = 0$, then the observation is not censored and the corresponding log-likelihood component is

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \log(\beta(1)) + \log(f(\beta(0) + \beta(1)y(0))), \tag{16}$$

while in the case of $y(1) = 1$, the the observation is censored at $y(0)$ and the log-likelihood component is

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \log(1 - F(\beta(0) + \beta(1)y(0))). \tag{17}$$

The function declaration is

f2v truncresp(const char & transform, const resp & y, const vec & beta).

The struct *resp* is defined as

struct resp{ivec iresp; vec dresp;}.

Here *y.iresp* has the single element $y(1)$, and *y.dresp* has the single element $y(0)$.

In the function declaration, *beta* is $\boldsymbol{\beta}$, *transform* is defined as in berresp.cpp, and *truncresp.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$. Functions required are berresp.cpp, contresp.cpp, and their respective required functions.

truncresp1.cpp

The function truncresp1.cpp computes the function value and gradient associated with the log-likelihood component defined in truncresp.cpp. The function declaration is

f1v truncresp1(const char & transform, const resp & y, const vec & beta).

Here *transform*, *resp*, *y*, and *beta* are defined as in truncresp.cpp, and *truncresp1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$. Functions required are berresp1.cpp, contresp1.cpp, and their respective required functions.

unpack.cpp

The function unpack.cpp is used in normalv.cpp and normalv1.cpp to convert a vector $\boldsymbol{\beta}$ of dimension $q_i = r_i(r_i + 3)/2$ to the *vecmat* format described in pack.cpp. The function declaration is *vecmat unpack(const int & d, const vec & beta).*

Here *d* is r_i, *beta* is $\boldsymbol{\beta}$, *unpack.v* is $\mathbf{a}(\boldsymbol{\beta})$, and *unpack.m* is $\mathbf{B}(\boldsymbol{\beta})$.

Computation of Log Likelihood Functions

genresp.cpp

The function genresp.cpp provides a general tool for computation of a a component of a log-likelihood function, its gradient, and its Hessian matrix. The function declaration is

f2v genresp(const model & choice, const resp & y, const vec & beta).

The struct *resp* is defined as in truncresp.cpp, while *model* has the definition

struct model{char type; char transform}.

Here *model.type* has value *C* for a cumulative case, *D* for a continuous case, *G* for a graded response, *L* for the multinomial logit case, *M* for the maximum of two independent Bernoulli variables, *P* for the log-mean Poisson case, *R* for the rank-logit case, *S* for the Bernouli case, and *T* for the censored continuous case. For discrete cases, *choice.transform* has possible values *G* for log-log cases, *L* for logit cases, and *N* for probit cases. For continuous cases, *G* is for the Gumbel distribution, *L* is for the logistic case, and *N* is for the normal case. For example, *choice.type* is *C* and *choice.transform* is *G* for the cumulative log-log case, while *choice.type* is *S* and *choice.type* is *N* in the probit case. The variable *choice.transform* is only relevant if *choice.type* is *C*, *D*, *G*, *M*, *S*, or *T*.

The function genresp.cpp uses berresp.cpp, contresp.cpp, cumresp.cpp, gradresp.cpp, logmean.cpp, maxberresp.cpp, multlogit.cpp, ranklogit.cpp, and truncresp.cpp, together with the functions they in turn require.

genresp1.cpp

The function genresp1.cpp provides a general tool for computation of a component of a log-likelihood function and its gradient. The function declaration is

f1v genresp1(const model & choice, const resp & y, const vec & beta).

The function arguments are defined as in genresp.cpp. The function genresp1.cpp uses berresp1.cpp, contresp1.cpp, cumresp1.cpp, gradresp1.cpp, logmean1.cpp, maxberresp1.cpp, multlogit1.cpp, ranklogit1.cpp, and truncresp1.cpp, together with the functions they in turn require.

genresplik.cpp

The function genresplik.cpp computes the log-likelihood function and its gradient and Hessian matrix. The function declaration is

f2v genresplik(const std::vector<dat> & data, const vec & beta).

The struct *dat* is defined by

struct dat{model choice; double weight; resp dep; vec offset; mat indep; xsel xselect;}.

Here *model* is defined as in genresp.cpp, *resp* is defined as in truncresp.cpp, and the struct *xsel* is defined by

struct xsel{bool all; ivec list}.

For $0 \leq i < n$, *data[i]* corresponds to observation i. Thus *data[i].choice* defines the model, *data[i].weight* is the observation weight, w_i, *data[i].resp* defines the dependent vector, $\mathbf{Y}_i$, *data[i].offset* is the offset vector $\mathbf{o}_i$, *data[i].indep* provides the matrix $\mathbf{X}_i$ of independent variables, and *data[i].xselect* are defined so that *x[i]* is $\mathbf{X}_i$ if *data[i].xselect[i].all* is *true*. Otherwise, two cases exist for $0 \leq j < p$. If *xselect[i].list* has K_i elements and j is *xselect[i].list(k)* for a nonnegative integer $k < K_i$, then column j of $\mathbf{X}_i$ is column k of *data[i].indep*. If j is not equal to any element of *x[i].list*, then column j of $\mathbf{X}_i$ is the q_i-dimensional vector with all elements 0.

The function genresplik.cpp uses genresp.cpp plus all C++ functions it in turn requires.

genresplik1.cpp

The function genresplik1.cpp computes the log-likelihood function and its gradient. The function declaration is

f1v genresplik1(const std::vector<dat> & data, const vec & beta).

Definitions of arguments of the function are the same as in genresplik.cpp. The function genresplik1.cpp uses genresp1.cpp plus all C++ functions it in turn requires.

genresplikl.cpp

The function genresplikl.cpp computes the log-likelihood function, its gradient, and its approximate Hessian matrix from Equation 4 when all components involve only discrete variables. The function declaration is *f2v genresplikl(const std::vector<dat> & data, const vec & beta).*

Definitions of arguments of the function are the same as in genresplik.cpp. The function genresplikl.cpp uses genresp1.cpp plus all C++ functions it in turn requires.

genrespmle.cpp

The function genrespmle.cpp applies the Newton-Raphson algorithm in nrv.cpp to the log-likelihood function, gradient, and Hessian matrix of genresplik.cpp. The function declaration is

maxf2v genrespmle(const params & mparams, const std::vector<dat> & data, const vec & start).

Here the structs *maxf2v* and *mparams* are defined as in maxlinq.cpp and maxf2vvar.cpp. The vector *start* is the starting vector. The global variables of genresplik.cpp are required. The functions nrv.cpp and genresplik.cpp are required, together with all C++ functions they in turn require.

genrespmle1.cpp

The function genrespmle1.cpp applies the conjugate gradient algorithm in conjgrad.cpp to the log-likelihood function and gradient of genresplik1.cpp. The function declaration is

maxf1v genrespmle1(const params & mparams, const std::vector<dat> & data, const vec & start).

Here the structs *maxf1v* and *params* are defined as in maxlinq.cpp. The vector *start* is the starting vector. The global variables of genresplik.cpp are required. The functions conjgrad.cpp and genresplik1.cpp are required, together with all functions they in turn require.

genrespmleg.cpp

The function genrespmleg.cpp applies the gradient ascent algorithm in gradascent.cpp to the log-likelihood function and gradient of genresplik1.cpp. The function declaration is

maxf1v genrespmle1(const params & mparams, const std::vector<dat> & data, const vec & start).

Here the structs *maxf1v* and *params* are defined as in maxlinq.cpp. The vector *start* is the starting vector. The global variables used in genresplik.cpp are required. The functions conjgrad.cpp and genresplik1.cpp are required, together with all functions they in turn require.

genrespmlel.cpp

The function genresplmlel.cpp applies the Newton-Raphson algorithm in nrv.cpp to the log-likelihood function, gradient, and approximate Hessian matrix of genresplikl.cpp. The function declaration is

maxf2v genrespmlel(const params & mparams, const std::vector<dat> & data, const vec & start).

Here the structs *maxf2v* and *params* are defined as in maxlinq2.cpp and maxf2vvar.cpp. The vector *start* is the starting vector. The global variables of genresplik.cpp are required. The functions nrv.cpp and genresplikl.cpp are required, together with all functions they in turn require.

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(const double & loc, const double & scale, const 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 adaptv(const vec & loc, const mat & lt, const 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(const 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(const imat & indices, const vector<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 $0 \le d < D$, *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, $0 \le k < p$, and $0 \le d < D$, 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 $0 \le d < D$.

hermcoeff.cpp

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

vec hermcoeff(const 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(const int &n, const double & x).

The order is the nonnegative integer variable *n*, and the real value is *x*. The vector *hermpoly* has *n*+1 elements. For $0 \le k \le n$, 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(const 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

Anderson, T. W. (2003). *An introduction to multivariate statistical analysis* (3rd ed.). Wiley-Interscience.

Cox, D. R. (1972). Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, *34*, 187–202. https://doi.org/10.1111/j.2517-6161.1972.tb00899.x

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

Haberman, S. J. (1974). *The analysis of frequency data*. University of Chicago Press.

Haberman, S. J. (1977). Maximum likelihood estimates in exponential response models. *The Annals of Statistics*, *5*, 815–841. https://doi.org/10.1214/aos/1176343941

Haberman, S. J. (1980). Discussion of *regression models for ordinal data*, by P. McCullagh. *Journal of the Royal Statistical Society, Series B*, *42*, 136–137.

Haberman, S. J. (2013). *A general program for item-response analysis that employs the stabilized Newton-Raphson algorithm* (ETS Research Report RR-13-32). Educational Testing Service. https://doi.org/10.1002/j.2333-8504.2013.tb02339.x

Kalbfleisch, J. D., & Prentice, R. L. (2002). *The statistical analysis of failure time data* (2nd ed.). John Wiley. https://doi.org/10.1002/9781118032985

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

Louis, T. (1982). Finding the observed information matrix when using the *em* algorithm. *Journal of the Royal Statistical Society, Ser. B*, *44*, 226–233. https://doi.org/10.2307/2345828

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

McFadden, D. L. (1973). Conditional logit analysis of qualitative choice behavior. In P. Zarembka (Ed.), *Frontiers in econometrics* (pp. 105–142). Academic Press.

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