

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`
- `conjgrad.cpp`
- `cumlogit.cpp`
- `cumlogit1.cpp`
- `cumloglog.cpp`
- `cumloglog1.cpp`
- `cumprobit.cpp`
- `cumprobit1.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
- gradlogit.cpp
- gradlogit1.cpp
- gradloglog.cpp
- gradloglog1.cpp
- gradprobit.cpp
- gradprobit1.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
- 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
- unpack.cpp

Distributions of Sums of Independent Multinomial Variables

The functions in this section implement a modified and generalized version of the Lord-Wingsky algorithm (Lord & Wingsky, 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-Wingsky 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 , $0 \leq j < n$ 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-Wingsky 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

```
maxflv maxlinq(const params & mparams, const vec & v,
               maxflv & vary0, function<flv(vec)>f).
```

Here the definition of *maxflv* is

```
struct maxflv{vec locmax; double max; vec grad;};
```

Here *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 *textslmaxlinq.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,  
maxf2v & vary0, function<f2v(vec)>f).
```

The struct *maxf2v* has the definition

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

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(double & x0, double & x1, double & f0, double & f1, double & g0,  
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 double & lower, const 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, *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.

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 , `maxf2vvar.grad` is the gradient of $f.value$ at y , and `maxf2vvar.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, 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, 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, function<f1v(vec)> f).
```


The functions `maxflvvar.cpp`, `maxlinq.cpp`, `maxquad.cpp`, `modit.cpp`, and `rebound.cpp` are used.

maxflvvar.cpp

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

maxflv maxflvvar(const vec & y, const flv & fy).

The returned value *maxflvvar.locmax* is *y*, while *maxflvvar.max* is the value of *f.value* at *y* and *maxflvar.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 and the q_i by p predicting matrix \mathbf{X}_i in a nonempty set \mathcal{X}_i . For some positive real numbers w_i , $0 \leq i < n$, some real functions v_i on \mathcal{Y}_i , and some q_i -dimensional vectors \mathbf{o}_i , the log-likelihood function under study has the form

$$\ell(\boldsymbol{\beta}) = \sum_{i=0}^{n-1} w_i [v_i(\mathbf{Y}_i) + \ell_i(\mathbf{o}_i + \mathbf{X}_i \boldsymbol{\beta}; \mathbf{Y}_i)] \quad (1)$$

for $\boldsymbol{\beta}$ in O . For $0 \leq i < n$, w_i is a sampling weight associated with observation i . 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 $\boldsymbol{\beta}$ in O and \mathbf{X} in \mathcal{X}_i , $\mathbf{o}_i + \mathbf{X}\boldsymbol{\beta}$ 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 $\exp(v_i(\mathbf{y}) + \ell_i(\boldsymbol{\beta}; \mathbf{y}))$ over \mathbf{y} in \mathcal{Y}_i and some random vector \mathbf{Y} equals \mathbf{y} with probability $\exp(v_i(\mathbf{y}) + \ell_i(\boldsymbol{\beta}; \mathbf{y}))$ 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 $\exp(v_i(\mathbf{y}) + \ell_i(\boldsymbol{\beta}; \mathbf{y}))$ over \mathbf{y} in \mathcal{Y}_i is 1 and a continuous random vector \mathbf{Y}_i has density $\exp(v_i(\mathbf{y}) + \ell_i(\boldsymbol{\beta}; \mathbf{y}))$ at \mathbf{y} in \mathcal{Y}_i . 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 $\boldsymbol{\beta}$ is

$$\nabla \ell(\boldsymbol{\beta}) = \sum_{i=0}^{n-1} w_i \mathbf{X}_i^T \nabla \ell_i(\mathbf{o}_i + \mathbf{X}_i \boldsymbol{\beta}; \mathbf{Y}_i), \quad (2)$$

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

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

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

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

(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*. Unless otherwise noted, $v_i(\mathbf{y})$ is equal to 0 for all \mathbf{y} in \mathcal{Y}_i . 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 .

cumlogit.cpp

The function *cumlogit.cpp* computes the function value, gradient, and Hessian matrix associated with a cumulative logit transformation. Let L be the distribution function of a standard logistic random variable, so that $L(y) = 1/[1 + \exp(-y)]$ for real y and the inverse function L^{-1} of L is the logit transformation with $L^{-1}(x) = \log(x/(1-x))$ for real x such that $0 < x < 1$. In this case, $r_i = 1$, $q_i \geq 1$, \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 , $q_i = n_i - 1$, and O_i is the set of all vectors of dimension q_i . For $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \begin{cases} \log(1 - L(\boldsymbol{\beta}(y(0)))), & y(0) = 0, \\ \log(1 - L(\boldsymbol{\beta}(y(0)))) + \sum_{i=0}^{y(0)-1} \log(L(\boldsymbol{\beta}(i))), & 0 < y(0) < q_i, \\ \sum_{i=0}^{y(0)-1} \log(L(\boldsymbol{\beta}(i))), & y(0) = q_i. \end{cases} \quad (5)$$

The function declaration is

f2v cumlogit(ivec & y, vec & beta).

The function *cumlogit.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{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*, r_i , q_i , O_i , and \mathcal{Y}_i are defined as in *cumlogit.cpp*, and Equation 5 holds for $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i . The function declaration is

flv cumlogit1(ivec & y, vec & beta).

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

cumloglog.cpp

The function *cumloglog.cpp* computes the function value, gradient, and Hessian matrix associated with a cumulative log-log transformation. Let G be the standard Gumbel distribution function with $G(y) = \exp(-\exp(-y))$ for real y . The inverse function G^{-1} of G is the log-log transformation with $G^{-1}(x) = -\log(-\log(x))$ for real x such that $0 < x < 1$. Then r_i , q_i , O_i , and \mathcal{Y}_i are defined as in *cumlogit.cpp*, and, for $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \begin{cases} \log(1 - G(\beta(y(0)))), & y(0) = 0, \\ \log[1 - G(\beta(y(0)))] + \sum_{i=0}^{y(0)-1} \log(G(\beta(i))), & 0 < y(0) < q_i, \\ \sum_{i=0}^{y(0)-1} \log(G(\beta(i))), & y(0) = q_i. \end{cases} \quad (6)$$

The function declaration is

f2v cumloglog(ivec & y, vec & beta).

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

cumloglog1.cpp

The function *cumloglog1.cpp* computes the function value and gradient associated with a cumulative log-log transformation. Here r_i , q_i , O_i , and \mathcal{Y}_i are defined as in *cumlogit.cpp*. As in *cumloglog.cpp*, Equation 6 holds for $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i . The function declaration is

flv cumloglog1(ivec & y, vec & beta).

The function *cumloglog1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$.

cumprobit.cpp

The function *cumprobit.cpp* computes the function value, gradient, and Hessian matrix associated with a cumulative probit transformation. Let Φ be the cumulative distribution function of the standard normal distribution with expectation 0 and variance 1, so that the inverse function Φ^{-1} defined on the interval $(0, 1)$ is the probit transformation. In this case, r_i , q_i , O_i , and \mathcal{Y}_i are defined as in *cumlogit.cpp*, and, for $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \begin{cases} \log[1 - \Phi(\beta(y(0)))] & y(0) = 0, \\ \log(1 - \Phi(\beta(y(0)))) + \sum_{i=0}^{y(0)-1} \log(\Phi(\beta(i))), & 0 < y(0) < q_i, \\ \sum_{i=0}^{y(0)-1} \log(\Phi(\beta(i))), & y(0) = q_i. \end{cases} \quad (7)$$

The function declaration is

f2v cumprobit(ivec & y, vec & beta).

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

cumprobit1.cpp

The function *cumprobit1.cpp* computes the function value and gradient associated with a cumulative probit transformation. Definitions of r_i , q_i , \mathcal{Y}_i , and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ are the same as in *cumprobit.cpp*. The function declaration is

f1v cumprobit1(ivec & y, vec & beta).

The function *cumprobit1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$.

gradlogit.cpp

The function *gradlogit.cpp* computes the function value, gradient, and Hessian matrix associated with a graded logit transformation. Define L as in *cumlogit.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 ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \begin{cases} \log(1 - L(\boldsymbol{\beta}(y(0)))) & y(0) = 0, \\ \log(L(\boldsymbol{\beta}(y(0) - 1)) - L(\boldsymbol{\beta}(y(0)))) & 0 < y(0) < q_i, \\ \log[L(\boldsymbol{\beta}(y(0) - 1))] & y(0) = q_i. \end{cases} \quad (8)$$

The function declaration is

f2v gradlogit(ivec & y, vec & beta).

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

gradlogit1.cpp

The function *gradlogit1.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 *gradlogit.cpp*. The function declaration is

f1v gradlogit1(ivec & y, vec & beta).

The function *gradlogit1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$. If $q_i = 1$, then *gradlogit1.cpp* and *cumlogit1.cpp* yield the same result.

gradloglog.cpp

The function `gradlogit.cpp` computes the function value, gradient, and Hessian matrix associated with a graded log-log transformation. Define G as in `cumloglog.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 β in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\beta; \mathbf{y}) = \begin{cases} \log(1 - G(\beta(y(0)))) & y(0) = 0, \\ \log(G(\beta(y(0) - 1)) - G(\beta(y(0)))) & 0 < y(0) < q_i, \\ \log[G(\beta(y(0) - 1))] & y(0) = q_i. \end{cases} \quad (9)$$

The function declaration is

f2v gradloglog(ivec & y, vec & beta).

The function `gradloglog.value` is $\ell_i(\beta; \mathbf{y})$ if y is \mathbf{y} and `beta` is β . If $q_i = 1$, then `gradloglog.cpp` and `cloglog.cpp` yield the same result.

gradloglog1.cpp

The function `gradloglog1.cpp` computes the function value and gradient associated with a graded complementary log-log transformation. Here r_i , q_i , O_i , \mathcal{Y}_i , and $\ell_i(\beta; \mathbf{y})$ are defined as in `gradloglog.cpp`. The function declaration is

f1v gradloglog1(ivec & y, vec & beta).

The function `gradloglog1.value` is $\ell_i(\beta; \mathbf{y})$ if y is \mathbf{y} and `beta` is β . If $q_i = 1$, then `gradloglog1.cpp` and `cloglog1.cpp` yield the same result.

gradprobit.cpp

The function `gradprobit.cpp` computes the function value, gradient, and Hessian matrix associated with a graded probit transformation. Define Φ as in `cumprobit.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 β in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\beta; \mathbf{y}) = \begin{cases} \log(1 - \Phi(\beta(y(0)))) & y(0) = 0, \\ \log(\Phi(\beta(y(0) - 1)) - \Phi(\beta(y(0)))) & 0 < y(0) < q_i, \\ \log[\Phi(\beta(y(0) - 1))] & y(0) = q_i. \end{cases} \quad (10)$$

The function declaration is

f2v gradprobit(ivec & y, vec & beta).

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

gradprobit1.cpp

The function *gradprobit1.cpp* computes the function value and gradient associated with a graded probit transformation. Here r_i , q_i , O_i , \mathcal{Y}_i , and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ are defined as in *gradprobit.cpp*. The function declaration is

f1v gradprobit1(ivec & y, vec & beta).

The function *gradprobit1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if y is \mathbf{y} and *beta* is $\boldsymbol{\beta}$. If $q_i = 1$, then *cumprobit1.cpp* and *gradprobit1.cpp* yield the same result.

gumbel.cpp

The function *gumbel.cpp* computes the function value, gradient, and Hessian matrix associated with the distribution of a random variable with the distribution of $\beta(0) + \beta(1)Z$, where Z has a standard Gumbel distribution with distribution function G defined in *cumloglog.cpp*, $\beta(0)$ is a real constant, and $\beta(1)$ is a positive real constant. The corresponding log-likelihood component is

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \log(\beta(1)) - \beta(0) - \beta(1)y(0) - \exp(-\beta(0) - \beta(1)y(0)), \quad (11)$$

where $r_i = 1$, \mathcal{Y}_i is the set of one-dimensional vectors \mathbf{y} , $q_i = 2$, and O_i is the set of all two-dimensional vectors $\boldsymbol{\beta}$ with $\beta(1)$ positive. The function declaration is

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

The function *gumbel.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ 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 log-likelihood component defined by Equation 11. Here r_i , \mathcal{Y}_i , q_i , and O_i are defined as in *gumbel.cpp*. The function declaration is

f1v gumbel1(vec & y, 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 distribution of $\beta(0) + \beta(1)Z$, where Z has a standard

logistic distribution with distribution function L defined as in `cumlogit.cpp`), $\beta(0)$ is a real constant, and $\beta(1)$ is a positive real constant. In this case, $r_i = 1$, \mathcal{Y}_i , q_i , and O_i are defined as in `gumbel.cpp`. For β in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\beta; \mathbf{y}) = \log(\beta(1)) + \log(L(\beta(0) + \beta(1)y(0))[1.0 - L(\beta(0) + \beta(1)y(0))]). \quad (12)$$

The function declaration is

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

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

logistic1.cpp

The function `logistic1.cpp` computes the function value and gradient associated with the log-likelihood component defined by Equation 12. Define r_i , \mathcal{Y}_i , q_i , O_i , and $\ell_i(\beta; \mathbf{y})$ as in `logistic.cpp`. The function declaration is

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

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

logit.cpp

The function `logit.cpp` computes the function value, gradient, and Hessian matrix associated with a logit transformation. Define L as in `cumlogit.cpp`. In this case, $r_i = 1$, \mathcal{Y}_i is the set of one-dimensional vectors \mathbf{y} such that $y(0)$ is 0 or 1, $q_i = 1$, and O_i is the set of all one-dimensional vectors. For β in O_i and \mathbf{y} in \mathcal{Y}_i , $\ell_i(\beta; \mathbf{y}) = \log(1 - L(\beta(0)))$ if $y(0) = 0$, and $\ell_i(\beta; \mathbf{y}) = \log(L(\beta(0)))$ if $y(0) = 1$. The function declaration is

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

The function *logit.value* is $\ell_i(\beta; \mathbf{y})$ if y is \mathbf{y} and *beta* is β . Use of `cumlogit.cpp` or `gradlogit.cpp` for $q_i = 1$ gives the same result as use of `logit.cpp`.

logit1.cpp

The function `logit1.cpp` computes the function value and gradient associated with a logit transformation. Define r_i , \mathcal{Y}_i , q_i , O_i , and $\ell_i(\beta; \mathbf{y})$ as in `logit.cpp`. The function declaration is

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

The function *logit1.value* is $\ell_i(\beta; \mathbf{y})$ if y is \mathbf{y} and *beta* is β . If $q_i = 1$, use of `cumlogit1.cpp` or `gradlogit1.cpp` yields the same result as use of `logit1.cpp`.

loglog.cpp

The function `loglog.cpp` computes the function value, gradient, and Hessian matrix associated with a log-log transformation. Let G be the standard Gumbel distribution function defined in `cumloglog.cpp`. In this case, $r_i = 1$, \mathcal{Y}_i is the set of one-dimensional vectors \mathbf{y} such that $y(0)$ is 0 or 1, $q_i = 1$, O_i is the set of all one-dimensional vectors. For β in O_i and \mathbf{y} in \mathcal{Y}_i , $\ell_i(\beta; \mathbf{y}) = \log(1 - G(\beta(0)))$ if $y(0) = 0$, and $\ell_i(\beta; \mathbf{y}) = \log(G(\beta(0)))$ if $y(0) = 1$. The function declaration is

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

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

The log-log and complementary log-log transformations are closely related. For $0 < x < 1$, the complementary log-log transformation is $-G^{-1}(1 - x)$.

loglog1.cpp

The function `loglog1.cpp` computes the function value and gradient associated with a log-log transformation. The definitions of r_i , q_i , O_i , \mathcal{Y}_i , and $\ell_i(\beta; \mathbf{y})$ are the same as in `loglog.cpp`. The function declaration is

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

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

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, $q_i = 1$, O_i is the set of all one-dimensional vectors, and $v_i(\mathbf{y}) = -\log([y(0)]!)$ for \mathbf{y} in \mathcal{Y}_i . For β in O_i and \mathbf{y} in \mathcal{Y}_i , $\ell_i(\beta; \mathbf{y}) = y(0)\beta(0) - \exp(\beta(0))$. The function declaration is

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

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

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 $\ell_i(\beta; \mathbf{y})$ as in `logmean.cpp`. The function declaration is

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

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

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. Let $\mathbf{0}_1$ be the one-dimensional vector with the single element 0. For $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \begin{cases} -\log \left(1 + \sum_{k=0}^{q_i-1} \exp(\beta(k)) \right), & \mathbf{y} = \mathbf{0}_1, \\ \beta(y(0) - 1) + \ell_i(\boldsymbol{\beta}; \mathbf{0}_1), & y(0) > 0. \end{cases} \quad (13)$$

The function declaration is

f2v multlogit(ivec & y, 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 *cumlogit.cpp*, *gradlogit.cpp*, or *logit.cpp*.

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(ivec & y, 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 *cumlogit1.cpp*, *gradlogit1.cpp*, or *logit1.cpp*. .

normal.cpp

The function *normal.cpp* computes the function value, gradient, and Hessian matrix associated with the distribution of $\beta(0) + \beta(1)Z$, where Z has a standard normal distribution with distribution function Φ defined as in *cumprobit.cpp*. In this case, $r_i = 1$, \mathcal{Y}_i , q_i , and O_i are defined as in *gumbel.cpp*, while $v_i(\mathbf{y}) = -0.5 \log(2\pi)$. For $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \log(\beta(1)) - (\beta(0) + \beta(1)y(0))^2/2. \quad (14)$$

The function declaration is

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

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

normal1.cpp

The function *normal1.cpp* computes the function value and gradient associated with the log-likelihood component defined by Equation 14. Define r_i , \mathcal{Y}_i , q_i , O_i , v_i , and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ as in *normal.cpp*. The function declaration is

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

The function *normal1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$.

normalv.cpp

The function *normalv.cpp* computes the function value, gradient, and Hessian matrix associated with the distribution of $\mathbf{a} + \mathbf{B}\mathbf{Z}$, where, for some integer $r_i \geq 1$, \mathbf{Z} is an r_i -dimensional multivariate normal random vector with zero mean and with covariance matrix equal to the identity matrix, \mathbf{a} is an r_i -dimensional constant vector with elements a_j for $0 \leq j < r_i$, \mathbf{B} is an r_i by r_i lower-triangular matrix with row j and column k equal to B_{jk} for $0 \leq k \leq j < r_i$, and $B_{jj} > 0$ for $0 \leq j < r_i$. In this case, $r_i = s$, \mathcal{Y}_i consists of all r_i -dimensional real vectors, O_i is the set of q_i -dimensional vectors $\boldsymbol{\beta}$ with elements β_h , $0 \leq h < q_i$ such that $\beta_h > 0$ if $h = r_i + j(j+3)/2$ for $0 \leq j < r_i$, and $v_i(\mathbf{y}) = -0.5r_i \log(2\pi)$ for \mathbf{y} in \mathcal{Y}_i . For $\boldsymbol{\beta}$ in O_i with elements β_h for $0 \leq h < q_i$, let \mathbf{a} be the r_i -dimensional vector with elements $a_j = \beta_j$ for $0 \leq j < r_i$, and let \mathbf{B} be the lower-triangular r_i by r_i matrix with row j and column k equal to β_h for $0 \leq k \leq j < r_i$ and $h = r_i + k + (j(j+1)/2)$. For \mathbf{y} in \mathcal{Y}_i ,

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \sum_{j=0}^{r_i-1} \log(\beta(j)) - (\mathbf{a} + \mathbf{B}\mathbf{y})^2/2. \quad (15)$$

The function declaration is

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

The function *normalv.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$. The function *normalv1.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 defined by Equation 15. Define r_i , \mathcal{Y}_i , q_i , O_i , v_i , and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ as in *normalv.cpp*. The function declaration is

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

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

pack.cpp

The function *pack.cpp* is used in *normalv.cpp* and *normalv1.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 *normalv.cpp* that corresponds to \mathbf{a} and \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(vecmat & u).
```

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

probit.cpp

The function *probit.cpp* computes the function value, gradient, and Hessian matrix associated with a probit transformation. Define Φ as in *cumprobit.cpp*. In this case, $r_i = 1$, \mathcal{Y}_i is the set of one-dimensional vectors \mathbf{y} such that $y(0)$ is 0 or 1, $q_i = 1$, and O_i is the set of all one-dimensional vectors. For $\boldsymbol{\beta}$ in O_i and \mathbf{y} in \mathcal{Y}_i , $\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \log(1 - \Phi(\boldsymbol{\beta}(0)))$ if $y(0) = 0$, and $\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \log(\Phi(\boldsymbol{\beta}(0)))$ if $y(0) = 1$. The function declaration is

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

The function *probit.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$. Use of *cumprobit.cpp* or *gradprobit.cpp* in the case of $q_i = 1$ 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. Define r_i , \mathcal{Y}_i , q_i , O_i , and $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ as in *probit.cpp*. The function declaration is

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

The function *probit1.value* is $\ell_i(\boldsymbol{\beta}; \mathbf{y})$ if *y* is \mathbf{y} and *beta* is $\boldsymbol{\beta}$. If $n_i = 2$, use of *cumprobit1.cpp* or *gradprobit1.cpp* 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 $r_i \leq q_i$. 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, define Z as in `gumbel.cpp` as a real random variable with a standard Gumbel distribution. Consider $\boldsymbol{\beta}$ in O_i . Let U_j , $0 \leq j \leq q_i$, be independent random variables such that U_0 has the same distribution as Z and $U_j - \beta_j$, $1 \leq j \leq q_i$, has the same distribution as Z . 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 , $0 \leq j < r_i$, if U_{y_j} is nonincreasing in j and $U_{y_j} \geq U_k$ 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 $\boldsymbol{\alpha}$ be the vector of dimension $q_i + 1$ such that element $\alpha_0 = 0$ and element $\alpha_j = \beta_{j+1}$ for $1 \leq j \leq q_i$, let K_0 be the set of nonnegative integers no greater than q_i and, for any positive integer $j < r_i$ that may exist, let K_j be formed from K_{j-1} by removing y_{j-1} . Then the log-likelihood component is

$$\ell_i(\boldsymbol{\beta}; \mathbf{y}) = \sum_{j=0}^{r_i-1} \left[\alpha_{y_j} - \log \left(\sum_{h \in K_j} \exp(\alpha_h) \right) \right]. \quad (16)$$

The function declaration is

f2v ranklogit(ivec & y, 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 a multinomial logit transformation. 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(ivec & y, 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 `multlogit.cpp`.

unpack.cpp

The function `unpack.cpp` is used in `normalv.cpp` and `normalv1.cpp` to convert a vector $\boldsymbol{\beta}$ of dimension $r_i(r_i + 3)/2$ to the `vecmat` format described in `pack.cpp`. The

function declaration is

```
vecmat unpack(int & d, vec & beta).
```

Here d is r_i , β is β , unpack.v is \mathbf{a} , and unpack.m is \mathbf{B} .

Computation of Log Likelihood Functions

genresp.cpp

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

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

The struct `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, M for the multinomial logit case, P for the log-mean Poisson case, R for the rank-logit case, and S for the Bernouli case. For discrete cases, `choice.transform` has possible values G for log-log cases, L for logit cases, and P 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 P in the probit case. The variable `choice.transform` is only relevant if `choice.type` is C , D , G , or S .

The struct `resp` has the definition

```
struct resp{ivec iresp, vec dresp}.
```

The function `genresp.cpp` uses `cumlogit.cpp`, `cumloglog.cpp`, `cumprobit.cpp`, `gradlogit.cpp`, `gradloglog.cpp`, `gradprobit.cpp`, `gumbel.cpp`, `logistic.cpp`, `logit.cpp`, `loglog.cpp`, `logmean.cpp`, `multlogit.cpp`, `normal.cpp`, `normalv.cpp`, `pack.cpp`, `probit.cpp`, `ranklogit.cpp`, and `unpack.cpp`.

genresp1.cpp

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

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

The structs *choice* and *resp* are defined as in *genresp.cpp*. The function *genresp1.cpp* uses *cumlogit1.cpp*, *cumloglog1.cpp*, *cumprobit1.cpp*, *gradlogit1.cpp*, *gradloglog1.cpp*, *gradprobit1.cpp*, *gumbel1.cpp*, *logistic1.cpp*, *logit1.cpp*, *loglog1.cpp*, *logmean1.cpp*, *multlogit1.cpp*, *normal1.cpp*, *normalv1.cpp*, *pack.cpp*, *probit1.cpp*, *ranklogit1.cpp*, and *unpack.cpp*.

genrespplik.cpp

The function *genrespplik.cpp* computes the log-likelihood function and its gradient and Hessian matrix when all components involve only discrete variables. The function declaration is

f2v genrespplik(vec & beta).

Use of *genrespplik.cpp* requires several external variables defined in a function that uses *genrespplik.cpp*. This function must permit access from other functions via *extern* specifications in *genrespplik.cpp*. In these definitions, the struct *xsel* is defined by

struct xsel{bool all;ivec list}.

Consider an observation i , $0 \leq i < n$. The array *choices*[] of *model* structs is defined so that *choices*[i] is the *model* struct for observation i . The *resp* array *y*[] is defined so that *y*[i].*iresp* is \mathbf{Y}_i when \mathbf{Y}_i is a discrete response and *y*[i].*dresp* is \mathbf{Y}_i when \mathbf{Y}_i is a continuous response. the *mat* array *x*[i] and the *xsel* array *xselect*[] are defined so that *x*[i] is \mathbf{X}_i if *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 *x*[i]. 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 *vec* array *offset*[] is defined so that *offset*[i] is \mathbf{o}_i .

The function *genrespplik.cpp* uses *genresp.cpp* plus all functions required by *genresp.cpp*.

genrespplik1.cpp

The function *genrespplik1.cpp* computes the log-likelihood function and its gradient when all components involve only discrete variables. The function declaration is

flv genrespplik1(vec & beta).

Use of `genrespplik1.cpp` requires the same external variables that are required by `genrespplik.cpp`.

genrespplik1.cpp

The function `genrespplik1.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 genrespplik(vec & beta).

Use of `genrespplik1.cpp` requires the same external variables that are required by `genrespplik.cpp`. The function uses `genresp1.cpp` plus the functions required by `genresp1.cpp`.

genrespmle.cpp

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

maxf2v genrespmle(const params & mparams, 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 external variables of `genrespplik.cpp` are required. The functions `nrv.cpp` and `genrespplik.cpp` are required, together with all 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 `genrespplik1.cpp`. The function declaration is

maxf1v genrespmle1(const params & mparams, const vec & start).

Here the structs *maxf1v* and *params* are defined as in `maxlinq.cpp`. The vector *start* is the starting vector. The external variables of `genrespplik.cpp` are required. The functions `conjgrad.cpp` and `genrespplik1.cpp` are required, together with all functions they in turn require.

genrespmleg.cpp

The function `genrespmleg.cpp` applies the gradient ascent algorithm in `gradascend.cpp` to the log-likelihood function and gradient of `genrespplik1.cpp`. The function

declaration is

maxf1v genresplmle1(const params & mparams, const vec & start).

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

genresplmlel.cpp

The function *genresplmlel.cpp* applies the Newton-Raphson algorithm in *nrv.cpp* to the log-likelihood function, gradient, and approximate Hessian matrix of *genresplik1.cpp*. The function declaration is

maxf2v genresplmlel(const params & mparams, 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 external variables of *genresplik.cpp* are required. The functions *nrv.cpp* and *genresplik1.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 corresponding 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 corresponding 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

- 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. (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>