

# 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++14. They rely on the Armadillo library (Sander-son & Curtin, 2016, 2018) at <http://arma.sourceforge.net>, the StatsLib library at <https://www.kthohr.com/statslib.html>, and the Boost library at <https://www.boost.org>. Unless otherwise noted, for the library members considered, it is assumed that users have verified that function arguments are valid. Namespaces assumed where relevant are *std*, *arma*, and *boost::math*. The following functions are found in the library.

- adaptpwr.cpp
- addsel.cpp
- berresp.cpp
- conjgrad.cpp
- conjgradn.cpp
- contresp.cpp
- cumresp.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 [haberman.statistics@gmail.com](mailto:haberman.statistics@gmail.com).

- eaps.cpp
- genfact.cpp
- genprods.cpp
- genresp.cpp
- genresplik.cpp
- genrespmle.cpp
- gradascent.cpp
- gradascentn.cpp
- gradresp.cpp
- gumbel.cpp
- hermcoeff.cpp
- hermpoly.cpp
- hermpw.cpp
- irtc.cpp
- irtcomps.cpp
- irtm.cpp
- irtmle.cpp
- irtms.cpp
- ivecsel.cpp
- linsel.cpp
- loggamma.cpp
- logistic.cpp
- logit.cpp
- logitbeta.cpp
- logitdirichlet.cpp

- loglog.cpp
- logmean.cpp
- lw.cpp
- lwm.cpp
- maxberresp.cpp
- maxf2vvar.cpp
- maxlinq2.cpp
- maxquad.cpp
- multlogit.cpp
- modit.cpp
- ngh.cpp
- normal.cpp
- normalv.cpp
- normwt.cpp
- nrv.cpp
- nrvn.cpp
- pack.cpp
- posterior.cpp
- posteriors.cpp
- probit.cpp
- qnormpw.cpp
- ??
- ranklogit.cpp
- rebound.cpp
- ??

- ??
- truncresp.cpp
- unpack.cpp
- vecsel.cpp
- wmc.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  $X_j$ ,  $0 \leq j < n$  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  $\eta$ ,  $\gamma_1$ ,  $\gamma_2$ , and  $\kappa$  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  $O$ .

#### **maxlinq2.cpp**

The function `maxlinq2.cpp` provides a line search for maximization algorithms. Only function values and gradients are used when `order` is 1, but Hessian matrices are computed if `order` is greater than 1. The function declaration is

```
maxf2v maxlinq2(const int & order, const params & mparams, const vec & v,
const maxf2v & vary0, const function<f2v(const int &, const vec &)>f).
```

Here the definition of `maxf2v` is

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

`vary0.locmax` is the starting vector for the line search, `vary0.max` is the value of  $f.value$  at the starting vector, `maxlinq2.grad` is the gradient of  $f.value$  at `vary0.locmax`, `maxlinq2.hess`, if computed, is the Hessian of  $f.value$  at `vary0.locmax`, and `maxlinq2.locmax` is the approximate location of the maximum of  $f.value$  on the half-line that starts at `vary0.locmax` and has direction  $v$ , `maxlinq2.max` is the approximate maximum of  $f.value$  on the half-line, `maxlinq2.grad` is the gradient of  $f.value$  at `maxlinq2.locmax`, and `maxlinq2.hess`, if computed, is the Hessian of  $f.value$  at `vary0.locmax`,

The definition of `params` is

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

Here *mparams.print* is used for output of the iteration number and function value at the end of the iteration, *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  $\eta$ , *mparams.gamma1* is  $\gamma_1$ , *mparams.gamma2* is  $\gamma_2$ , and *mparams.kappa* is  $\kappa$ . Iterations cease if the function value changes less than *mparams.tol* after a primary iteration.

The definition of *f2v* is

```
struct f2v{double value; vec grad; vec hess};,
```

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

The functions *maxf2vvar.cpp*, *maxquad.cpp*, *modit.cpp*, and *rebound.cpp* are all used.

### **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  $\infty$ . 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  $\eta$ , *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  $\infty$ . 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 int & order, 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 *fy.value*, *maxf2vvar.grad* is *fy.grad*, and *maxf2vvar.hess* is *fy.hess* at *y*. If *order* is less than 1, only *fy.value* is considered

If *order* is 1, *fy.value* and *fy.grad* are considered. If *order* exceeds 1, then *fy.value*, *fy.grad*, and *fy.hess* are used.

### **ngh.cpp**

The function *ngh.cpp* finds gradients and Hessian matrices via numerical differentiation. The function declaration is

```
f2v ngh(const int & order, const double & delta, const vec & x,  
const function <f2v(const int & , const vec & )>f).
```

If the variable *order* is 0, only the function value *f.value* at *x* is reported. If *order* is 1, then *f.grad* at *x* is also reported by a numerical approximation with step size *delta*. If *order* is at least 2, then both *f.grad* and *f.hess* at *x* are reported by use of numerical approximations with step size *delta*. Values of *f.grad* are found by evaluation of *f.value* at vectors that differ from *x* in only one element. When an element of these vectors differ from those of *x*, the difference has absolute value *delta*. Values of *f.hess* are found by evaluation of *f.value* at vectors that differ from *x* in no more than two elements. When an element of these vectors differ from those of *x*, the difference has absolute value *delta*.

### **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 int & order, const params & mparams, const vec & start,  
const function<f2v(const int &, vec &)> f).
```

The structs *f2v*, *maxf2v*, and *params* are defined as in *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*. The value of *order* should be at least 2.

### **nrvn.cpp**

The function *nrvn.cpp* uses numerical differentiation for a modified version of the Newton-Raphson algorithm to maximization of *f.value*. The function *nrvn.cpp* has declaration

```
maxf2v nrvn(const int & order, const params & mparams, const vec & start,  
const double & step, const function<f2v(const int &, vec &)> f).
```



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

The function *nrvn.cpp* uses *nrvcpp*, *ngh.cpp*, *maxf2vvar.cpp*, *maxlinq2.cpp*, *maxquad.cpp*, *modit.cpp*, and *rebound.cpp*. The value of *order* should be at least 2.

### 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

```
maxf2v conjgrad(const int & order, const params & mparams,  
const vec & start, const function<f2v(const int & , const vec &)> f).
```

The starting vector is *start*. The value of *order* must be at least 1. If *order* is at least 2, Hessian matrices are computed even though not used in the algorithm.

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

#### **conjgradn.cpp**

The function *conjgradn.cpp* uses numerical differentiation to implement a conjugate gradient algorithm for maximization of *f.value*. The function declaration is

```
maxf2v conjgradn(const int & order, const params & mparams,  
const vec & start, const double & step, const function<f2v(const int & , const vec  
&)> f).
```

The starting vector is *start*. The value of *order* must be at least 1. If *order* is at least 2, Hessian matrices are computed even though not used in the algorithm. The positive step size for differentiation is *step*.

The function *conjgradn.cpp* uses *conjgrad.cpp*, *ngh.cpp*, *maxf2vvar.cpp*, *maxlinq2.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

```
maxf2v gradascent(const order & , const params & mparams,  
const vec & start, const function<f2v(const int & , const vec & )> f).
```

The functions `maxf2vvar.cpp`, `maxlinq2.cpp`, `maxquad.cpp`, `modit.cpp`, and `rebound.cpp` are used. Definitions are as in `conjgradn.cpp`.

### **gradascentn.cpp**

The function `gradascentn.cpp` uses numerical differentiation to implement a gradient-ascent algorithm for maximization of *f.value*. The function declaration for `gradascentn.cpp` is

*maxf2v gradascentn(const order & , const params & mparams,*  
*const vec & start, const double & step, const function<f2v(const int & , const vec &*  
*)> f).*

The functions `gradascent.cpp`, `ngh.cpp`, `maxf2vvar.cpp`, `maxlinq2.cpp`, `maxquad.cpp`, `modit.cpp`, and `rebound.cpp` are used. Definitions are as in `conjgradn.cpp`.

## **Log-likelihood Components**

In this section, components of log-likelihood functions are provided. A component has the form  $\ell_c(\beta; \mathbf{Y}, A, F, q, r)$ . Here the character *A* defines the type of model component involved, *F* is a distribution function with a positive and twice-continuously differential derivative  $F_1$  such that  $\log F_1$  has a negative second derivative. The integer  $q > 0$  is the parameter dimension, and the integer  $r > 0$  is the data dimension. The character *A* is in the set  $\mathcal{A}$  with elements *B*(logit beta), *C* (cumulative), *D* (continuous), *E* (logit Dirichlet), *G* (graded), *H* (log gamma), *L* (multinomial logit), *M* (maximum of two independent Bernoulli variables), *N* (multivariate normal), *P* (log-mean Poisson case), *R* (rank logit), *S* (Bernoulli), and *T* (censored continuous). Distribution functions used in this section are in the set  $\mathcal{F}$  with three members, *G*, the standard Gumbel distribution function with value  $G(y) = \exp(-\exp(-y))$  for *y* real,  $\Psi$ , the standard logistic distribution function with value  $\Psi(y) = 1/[1+\exp(-y)]$  for *y* real, and  $\Phi$ , the standard normal distribution function with derivative  $\Phi_1(y) = \exp(-y^2/2)/(2\pi)^{1/2}$  for real *y*. The value of *F* is only relevant in the cumulative, continuous, graded, Bernoulli, and censored continuous cases. The variables *M*, *F*, *q*, and *r* then define an open convex subset  $O(A, F, q, r)$  of *q*-dimensional vectors and a set  $\mathcal{Y}(A, F, q, r)$  of *r*-dimensional vectors. The vector  $\beta$  is in  $O(A, F, q, r)$ , and  $\mathbf{Y}$  is in  $\mathcal{Y}(A, F, q, r)$ .

To treat both continuous and discrete log-likelihood components, the integral symbol  $\int$  is used in the following sense. Consider a real function *g* on a nonempty finite-dimensional set *C*. If *C* is convex and has a nonempty interior and *g* is integrable, then  $\int(g)$  denotes the integral of *g* over *C*. If *C* is finite or countably infinite and *g* is summable, then  $\int(g)$  is the sum of  $g(\mathbf{c})$  over  $\mathbf{c}$  in *C*. More generally, let  $\mathcal{D}$  be a finite or countably infinite collection of nonempty disjoint sets *D* that are either convex

sets with nonempty interior or finite or countably-infinite sets. Let  $C$  be the union of the sets in  $\mathcal{D}$ , and let  $g_D$  denotes the restriction of  $g$  to  $D$  in  $\mathcal{D}$ . Let  $f(g_D)$  be defined for  $D$  in  $\mathcal{D}$ , and let the  $f(g_D)$ ,  $D$  in  $\mathcal{D}$ , be summable. Then  $f(g)$  is the sum of  $f(g_D)$  over  $D$  in  $\mathcal{D}$ . Similar conventions apply if  $g$  is vector-valued or matrix-valued. The requirement is imposed here that, for  $\beta$  in  $O(A, F, q, r)$ ,  $f(\exp(\ell_c(\beta; \cdot, A, F, q, r))) = 1$ . Here  $\exp(\ell_c(\beta; \cdot, A, F, q, r))$  is the function on  $\mathcal{Y}(A, F, q, r)$  equal to  $\exp(\ell_c(\beta; \mathbf{y}, A, F, q, r))$  if  $\mathbf{y}$  is in  $\mathcal{Y}(A, F, q, r)$ . The gradient function of  $\ell_c(\cdot; \mathbf{y}, A, F, q, r)$  is  $\nabla \ell_c(\cdot; \mathbf{y}, A, F, q, r)$  and the corresponding Hessian matrix is  $\nabla^2 \ell_c(\cdot; \mathbf{y}, A, F, q, r)$ .

For a positive integer  $n$  and an observation  $i$ ,  $0 \leq i < n$ , positive integers  $q_i$  and  $r_i$  and character variables  $A_i$  in  $\mathcal{A}$  and  $F_i$  in  $\mathcal{F}$  are given. The component of the log likelihood for observation  $i$  involves the predicted random vector  $\mathbf{Y}_i$  in  $\mathcal{Y}(A_i, F_i, q_i, r_i)$ , 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  $\boldsymbol{\lambda}_i(\boldsymbol{\tau}) = \mathbf{o}_i + \mathbf{X}_i \boldsymbol{\tau}$  be in  $O(A_i, F_i, q_i, r_i)$  for  $0 \leq i < n$ , and let the log-likelihood function under study have the form

$$\ell(\boldsymbol{\tau}) = \sum_{i=0}^{n-1} w_i \ell_c(\boldsymbol{\lambda}_i(\boldsymbol{\tau}); \mathbf{Y}_i, A_i, F_i, q_i, r_i). \quad (1)$$

It follows that the gradient of  $\ell$  at  $\boldsymbol{\tau}$  in  $O$  is

$$\nabla \ell(\boldsymbol{\tau}) = \sum_{i=0}^{n-1} w_i \mathbf{X}_i^T \nabla \ell_c(\boldsymbol{\lambda}_i(\boldsymbol{\tau}); \mathbf{Y}_i, A_i, F_i, q_i, r_i), \quad (2)$$

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

$$\nabla^2 \ell(\boldsymbol{\tau}) = \sum_{i=0}^{n-1} w_i \mathbf{X}_i^T \nabla^2 \ell_c(\boldsymbol{\lambda}_i(\boldsymbol{\tau}); \mathbf{Y}_i, A_i, F_i, q_i, r_i) \mathbf{X}_i. \quad (3)$$

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

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

(Haberman, 2013; Louis, 1982) .

The functions  $\ell_c(\cdot; \mathbf{y}, A, F, q, r)$  used are considered in this section. Some 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 `f2v` are defined as in `maxling2.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  $\mathbf{y}$  is in  $\mathcal{Y}_i$ . In the cases under study in

this section, unless otherwise stated, the components are strictly concave, so that  $\ell$  is strictly concave whenever  $\mathbf{X}_i$ ,  $0 \leq i < n$ , 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  $\ell$  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  $\ell$  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  $\ell$  on  $O$  and, for  $\boldsymbol{\tau}$  in  $O$ ,  $\nabla \ell(\boldsymbol{\tau})$  is only the vector with all elements 0 if  $\boldsymbol{\beta}$  equals  $\hat{\boldsymbol{\beta}}$ . In all component functions, *order* is less than 1 if only the component value is computed, 1 if the component value and gradient are found, and greater than 1 if the component value, gradient, and Hessian are found. If *order* exceeds 2, the approximation of the Hessian by Equation 4 is employed. Repeated use is made of the struct *resp* with *vec* component *dresp* and *ivec* component *iresp*. In typical cases, *resp.dresp* or *resp.iresp* has no elements; however, exceptions do exist.

### berresp.cpp

The function *berresp.cpp* is used to handle standard models for Bernoulli random variables. Here  $q = r = 1$ ,  $A$  is  $S$ ,  $\mathcal{Y}(A, F, q, r)$  is the set of one-dimensional vectors  $\mathbf{y}$  with  $y(0)$  equal 0 or 1, and  $O(A, F, q, r)$  is the set of all one-dimensional vectors, and  $F$  is in  $\mathcal{F}$ . For  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$  and  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$ ,

$$\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r) = \begin{cases} \log(F(\beta(0))), & y(0) = 1, \\ \log(1 - F(\beta(0))), & y(0) = 0. \end{cases} \quad (5)$$

The function declaration is

```
f2v berresp(const int & order, const char & transform, const resp & y,
const vec & beta).
```

If *transform* is  $G$ , then  $F = G$ , If *transform* is  $L$ , then  $F = \Psi$ . If *transform* is  $N$ , then  $F = \Phi$ . The function *berresp.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ .

The function *berresp.cpp* requires *loglog.cpp*, *logit.cpp*, and *probit.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. Here  $r = 1$ ,  $q = 2$ ,  $A$  is  $D$ ,  $\mathcal{Y}(A, F, q, r)$  is the set of all one-dimensional vectors,  $O(A, F, q, r)$  is the set of all two-dimensional vectors  $\boldsymbol{\beta}$  with element  $\beta(1) > 0$ , and  $F$  is in  $\mathcal{F}$ . For  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$  and  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$ ,

$$\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r) = \log(\beta(1)) + \log(F_1(\beta(0) + \beta(1)y(0))). \quad (6)$$

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

For all cases, the function declaration is

*f2v contresp(const int & order, const char & transform, const resp & y,  
const vec & beta).*

The variable *transform* is defined as in *berresp.cpp*. The function *contresp.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.dresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ . The function *contresp.cpp* requires *gumbel.cpp*, *logistic.cpp*, and *normal.cpp*.

### **cumresp.cpp**

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

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

The function declaration is

*f2v cumresp(const int & order, const char & transform, const resp & y,  
const vec & beta).*

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

### **gradresp.cpp**

The function *gradresp.cpp* computes the function value, gradient, and Hessian matrix associated with a graded response transformation. Then  $r = 1$ ,  $q \geq 1$ ,  $A$  is  $G$ ,  $F$  is in  $\mathcal{F}$ ,  $O(A, F, q, r)$  is the set of all vectors of dimension  $q$  with strictly decreasing

elements,  $\mathcal{Y}(A, F, q, r)$  is the set of one-dimensional vectors  $\mathbf{y}$  with  $y(0)$  a nonnegative integer no greater than  $q$ , and, for  $\beta$  in  $O(A, F, q, r)$  and  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$ ,

$$\ell_i(\beta; \mathbf{y}) = \begin{cases} \log(1 - F(\beta(y(0)))), & y(0) = 0, \\ \log(F(\beta(y(0) - 1)) - F(\beta(y(0))))), & 0 < y(0) < q, \\ \log(F(\beta(y(0) - 1))), & y(0) = q. \end{cases} \quad (8)$$

The function declaration is

*f2v gradresp(const int & order, const char & transform, const resp & y,  
const vec & beta).*

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

### **gumbel.cpp**

The function *gumbel.cpp* provides the computations required in *contresp.cpp* for  $\ell_c(\cdot; \mathbf{y}, A, F, q, r)$  for the simple Gumbel case of  $F = G$ ,  $A$  with value  $D$ ,  $q = 2$ ,  $r = 1$ ,  $\mathcal{Y}(A, F, q, r)$  the set of real numbers, and  $O(A, F, q, r)$  the set of two-dimensional vectors  $\beta$  with  $\beta(1) > 0$ . The function declaration is

*f2v gumbel(const int & order, const resp & y, const vec & beta).*

The function *gumbel.value* is then  $\ell_c(\beta; \mathbf{y}, A, F, q, r)$  if *y.dresp* is  $\mathbf{y}$  and *beta* is  $\beta$ .

### **loggamma.cpp**

The function *loggamma.cpp* provides the computations required for  $\ell_c(\beta; \mathbf{y}, A, F, q, r)$  for the log-gamma distribution with  $A$  with value  $H$ ,  $q = 2$ ,  $r = 1$ ,  $\mathcal{Y}(A, F, q, r)$  the set of real numbers, and  $O(A, F, q, r)$  the set of two-dimensional vectors  $\beta$  with positive elements. The function declaration is

*f2v loggamma(const int & order, const resp & y, const vec & beta).*

The function *loggamma.value* is then  $\ell_c(\beta; \mathbf{y}, A, F, q, r)$  if *y.dresp* is  $\mathbf{y}$  and *beta* is  $\beta$ .

### **logistic.cpp**

The function *logistic.cpp* provides the computations required in *contresp.cpp* for  $\ell_c(\cdot; \mathbf{y}, A, F, q, r)$  for the logistic case  $F = \Psi$  in *contresp.cpp* with  $A$  with value  $D$ ,  $q = 2$ ,  $r = 1$ ,  $\mathcal{Y}(A, F, q, r)$  the set of real numbers, and  $O(A, F, q, r)$  the set of two-dimensional vectors  $\beta$  with  $\beta(1) > 0$ . The function declaration is

*f2v logistic(const int & order, const resp & y, const vec & beta).*

The function *logistic.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.dresp* 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 *A* equal to *S*, *F* =  $\Psi$ , and *q* = *r* = 1. The function declaration is

*f2v logit(const int & order, const resp & y, const vec & beta).*

The function *logit.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ .

### **logitbeta.cpp**

The function *logitbeta.cpp* computes the function value, gradient, and Hessian matrix associated with the logit of a beta distribution with a two-dimensional parameter vector  $\boldsymbol{\beta}$  with positive elements. Here  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  has *A* with value *H*, *q* = 2, *r* = 1,  $\mathcal{Y}(A, F, q, r)$  the set of real numbers, and  $O(A, F, q, r)$  the set of two-dimensional vectors  $\boldsymbol{\beta}$  with positive elements. The function declaration is

*f2v logitbeta(const int & order, const resp & y, const vec & beta).*

The function *logitbeta.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *q* = 2, *r* = 1, *y.iresp* is  $\mathbf{y}$ , *A* is *B*, and *beta* is  $\boldsymbol{\beta}$ .

### **logitdirichlet.cpp**

The function *logitdirichlet.cpp* computes the function value, gradient, and Hessian matrix associated with the logits of a Dirichlet distribution with a *q* = *r* + 1-dimensional parameter vector  $\boldsymbol{\beta}$  with positive elements. Here  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  has *A* with value *E*,  $\mathcal{Y}(A, F, q, r)$  the set of *r*-dimensional vectors, and  $O(A, F, q, r)$  the set of *q*-dimensional vectors  $\boldsymbol{\beta}$  with positive elements. The function declaration is

*f2v logitdirichlet(const int & order, const resp & y, const vec & beta).*

The function *logitdirichlet.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$ , *beta* is  $\boldsymbol{\beta}$ , and *A* is *E*. The *q*-dimensional random variable  $\mathbf{u}$  has a Dirichlet distribution with parameter vector  $\boldsymbol{\beta}$  if 1 is the sum of the elements of  $\mathbf{u}$  and  $y_i = \log(u_i/u_q)$  for integers *i* from 1 to *r*. If *r* = 1, then the logit Dirichlet case reduces to the case of a logit 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  $A$  equal to  $S$ ,  $F = G$ , and  $q = r = 1$ . The function declaration is

*f2v loglog(const int & order, const resp & y, const vec & beta).*

The function `loglog.value` is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if `y.iresp` 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,  $q = r = 1$ ,  $A$  is  $P$ , the value of  $F$  in  $\mathcal{F}$  is irrelevant,  $\mathcal{Y}(A, F, q, r)$  is the set of one-dimensional vectors  $\mathbf{y}$  such that  $y(0)$  is a nonnegative integer, and  $O(A, F, Q, R)$  is the set of all one-dimensional vectors. For  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$  and  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$ ,

$$\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r) = y(0)\beta(0) - \exp(\beta(0)) - \log([y(0)]!). \quad (9)$$

The function declaration is

*f2v logmean(const int & order, const resp & y, const vec & beta).*

The function `logmean.value` is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if `y.iresp` 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. Here  $q = 2$ ,  $r = 1$ ,  $A$  is  $M$ ,  $F$  is in  $\mathcal{F}$ ,  $O(A, F, q, r)$  is the set of two-dimensional vectors, and  $\mathcal{Y}(A, F, q, r)$  is the set of one-dimensional vectors  $\mathbf{y}$  with  $y(0)$  equal 0 or 1. For  $y$  in  $\mathcal{Y}(A, F, q, r)$  and  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$ ,

$$\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r) = \begin{cases} \log(F(\beta(0) + F(\beta(1) - F(\beta(0)F(\beta(1))), & y(0) = 1, \\ \log(1 - F(\beta(0))) + \log(1 - F(\beta(1))), & y(0) = 0. \end{cases} \quad (10)$$

It should be noted that

$$F(\beta(0) + F(\beta(1) - F(\beta(0)F(\beta(1))) = 1 - [1 - F(\beta(0))][1 - F(\beta(1))] \quad (11)$$

and

$$\log(1 - F(\beta(0))) + \log(1 - F(\beta(1))) = \log([1 - F(\beta(0))][1 - F(\beta(1))]). \quad (12)$$

The function  $\ell_c(\cdot; \mathbf{y}, A, F, q, r)$  is not necessarily concave if  $y(0) = 1$ .

The function declaration is



*f2v maxberresp(const int & order, const char & transform, const resp & y, const vec & beta).*

The variable *transform* is defined as in *berresp.cpp*. The function *maxberesp.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ . The functions *berresp.cpp*, *logit.cpp*, *loglog.cpp*, and *probit.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,  $r = 1$ ,  $q \geq 1$ ,  $F$  is irrelevant,  $A$  is  $L$ ,  $\mathcal{Y}(A, F, q, r)$  is the set of one-dimensional vectors  $\mathbf{y}$  such that  $y(0)$  is a nonnegative integer no greater than  $q$ , and  $O(A, F, q, r)$  is the set of all  $q$ -dimensional vectors. For  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$  and  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$ ,

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

The function declaration is

*f2v multlogit(const int & order, const resp & y, const vec & beta).*

The function *multlogit.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ . If  $q = 1$ , 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  $L$ .

### **normal.cpp**

The function *normal.cpp* computes the function value, gradient, and Hessian matrix associated with the normal case in *contresp.cpp*. Thus  $A$  is  $D$ ,  $F = \Phi$ ,  $q = 2$ ,  $r = 1$ ,  $\mathcal{Y}(A, F, q, r)$  is the space of one-dimensional vectors, and  $O(A, F, q, r)$  is the set of two-dimensional vectors  $\boldsymbol{\beta}$  with  $\beta(1) > 0$ . The function declaration is

*f2v normal(const int & order, const vec & y, const vec & beta).*

The function *normal.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.dresp* 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 log-likelihood component associated with a multivariate normal model with  $r$  positive,  $q = r(r + 3)/2$ ,  $A$  equal to  $N$ ,  $F$  is irrelevant,  $\mathcal{Y}(A, F, q, r)$  the set of all  $r$ -dimensional real vectors, and  $O(A, F, q, r)$  the set of  $q$ -dimensional vectors  $\boldsymbol{\beta}$  with elements  $\beta_h$ ,  $0 \leq h < q$  such that  $\beta_h > 0$  if

$h = r + j(j + 3)/2$  and  $0 \leq j < r$ . For such  $\beta$ , let  $\mathbf{a}(\beta)$  be the  $r$ -dimensional vector with elements  $a_j(\beta) = \beta_j$  for  $0 \leq j < r$ , and let  $\mathbf{B}(\beta)$  be the symmetric positive-definite  $r$  by  $r$  matrix with row  $j$  and column  $k$  equal to  $\beta_h$  if  $0 \leq k \leq j < r$  and  $h = r + k + (j(j + 1)/2)$ . For an  $r$ -dimensional vector  $\mathbf{z}$  with elements  $z_j$ ,  $0 \leq j < r$ , let  $\phi(\mathbf{z}; r)$  be the product of the  $\Phi_1(z_j)$ ,  $0 \leq j < r$ .

For  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$ ,

$$\ell_c(\beta; \mathbf{y}) = \left[ \sum_{j=0}^{r-1} \log(\beta(r + j(j + 3)/2)) \right] + \log(\phi(\mathbf{a}(\beta) + \mathbf{B}(\beta)\mathbf{y}; r)). \quad (14)$$

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

*f2v normalv(const int & order, const resp & y, const vec & beta).*

The function *normalv.value* is  $\ell_c(\beta; \mathbf{y}, A, F, q, r)$  if *y.dresp* is  $\mathbf{y}$  and *beta* is  $\beta$ . If  $r$  is 1, then *normalv.cpp* reduces to *normal.cpp*. The function *normalv.cpp* requires *pack.cpp* and *unpack.cpp*.

### **pack.cpp**

For the struct *vecmat* defined by

```
struct vecmat{vec v; mat m;};
```

the function *pack.cpp* converts a  $d$ -dimensional vector *pack.v* and a  $d$  by  $d$  symmetric matrix *pack.m* to a vector with dimension  $d(d+3)/2$  with  $d$  initial elements the vector *pack.v* and element  $h = d + k + (j(j + 1)/2)$  equal to row  $j$  and column  $k$  of *pack.m* for nonnegative  $k$  no greater than  $j < d$ . The function declaration is

*vec pack(const vecmat & u).*

Diagonal elements of the matrix equal the corresponding elements of *u*, and off-diagonal elements are twice the corresponding elements of the vector.

### **probit.cpp**

The function *probit.cpp* computes the function value, gradient, and Hessian matrix associated with a probit transformation in *berresp.cpp* with  $A$  equal to  $S$ ,  $F = \Psi$ , and  $q = r = 1$ . The function declaration is

*f2v probit(const int & order, const resp & y, const vec & beta).*

The function *probit.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ .  $F = \Phi$ . The function declaration is

*f2v probit(const int & order, const resp & y, const vec & beta).*

The function *probit.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ .

### **ranklogit.cpp**

The function *ranklogit.cpp* computes the function value, gradient, and Hessian matrix associated with a model for discrete choice in which  $q + 1$  objects are ranked for some positive integer  $q$  and the  $r$  most-preferred objects are recorded for some positive integer  $r \leq q$ . Here  $A$  has value  $R$ ,  $F$  is irrelevant, the set  $\mathcal{Y}(A, F, q, r)$  consists of the vectors  $\mathbf{y}$  of dimension  $r$  with distinct nonnegative integer elements that are no greater than  $q$ , and  $O(A, F, q, r)$  is the set of all  $q$ -dimensional vectors. To describe the model, consider the standard Gumbel distribution function  $G$ . Consider  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$ . Let  $U(j)$ ,  $0 \leq j \leq q$ , be independent random variables such that  $U(0)$  and  $U(j) - \beta(j)$ ,  $1 \leq j \leq q$ , have the common distribution function  $G$ . Let  $\mathbf{Y}$  be a random vector with values in  $\mathcal{Y}(A, F, q, r)$  such that  $\mathbf{Y}$  is the member  $\mathbf{y}$  of  $\mathcal{Y}(A, F, q, r)$  with elements  $y(j)$ ,  $0 \leq j < r$ , if  $U(y(j))$  is nonincreasing in  $j$  and  $U(y(j)) \geq U(k)$  if  $k$  is a nonnegative integer no greater than  $q$  that does not equal  $y(h)$  for any nonnegative integer element  $h < r$ . For  $\boldsymbol{\beta}$  in  $O(A, F, q, r)$  and  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$ , let  $\boldsymbol{\alpha}(\boldsymbol{\beta})$  be the vector of dimension  $q + 1$  such that element  $j$ ,  $0 \leq j \leq q$ , is  $\alpha(j; \boldsymbol{\beta}) = 0$  if  $j = 0$  and  $\alpha(j; \boldsymbol{\beta}) = \beta(j - 1)$  if  $j > 0$ . For  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$  and  $0 \leq j < r$ , let  $K(j; \mathbf{y})$  be the set of nonnegative integers no greater than  $q$  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  $q$ . Then the log-likelihood component is

$$\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r) = \sum_{j=0}^{r-1} \left[ \alpha(y_j; \boldsymbol{\beta}) - \log \left( \sum_{h \in K(j; \mathbf{y})} \exp(\alpha(h; \boldsymbol{\beta})) \right) \right]. \quad (15)$$

The function declaration is

*f2v ranklogit(const int & order, const resp & y, const vec & beta).*

The function *ranklogit.value* is  $\ell_c(\boldsymbol{\beta}; \mathbf{y}, A, F, q, r)$  if *y.iresp* is  $\mathbf{y}$  and *beta* is  $\boldsymbol{\beta}$ . If  $r = 1$ , use of *ranklogit.cpp* gives the same result as use of *multlogit.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  $F$  in  $\mathcal{F}$ . In this case,  $q = r = 2$ ,  $A$  is  $T$ .  $\mathcal{Y}(A, F, q, r)$  consists of two-dimensional vectors  $\mathbf{y}$  such that  $y(0)$  is a real number and  $y(1)$  is 0 or 1, and  $O(A, F, q, r)$  is the set of all two-dimensional vectors  $\beta$  with element  $\beta(1) > 0$ . For  $\beta$  in  $O(A, F, q, r)$  and  $\mathbf{y}$  in  $\mathcal{Y}(A, F, q, r)$ , if  $y(1) = 0$ , then the observation is not censored and the corresponding log-likelihood component is

$$\ell_c(\beta; \mathbf{y}, A, F, q, r) = \log(\beta(1)) + \log(F_1(\beta(0) + \beta(1)y(0))), \quad (16)$$

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

$$\ell_c(\beta; \mathbf{y}, A, F, q, r) = \log(1 - F(\beta(0) + \beta(1)y(0))). \quad (17)$$

The function declaration is

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

Here *y.iresp* has the single element  $y(1)$ , *y.dresp* has the single element  $y(0)$ , *beta* is  $\beta$ , *transform* is defined as in `berresp.cpp`, and *truncresp.value* is  $\ell_c(\beta; \mathbf{y}, A, F, q, r)$ . Functions required are `berresp.cpp`, `contresp.cpp`, and their respective required functions.

### **unpack.cpp**

The function `unpack.cpp` converts a vector of dimension  $d(d+3)/2$  to the *vecmat* format described in `pack.cpp`. The function declaration is

*vecmat unpack(const int & d, const vec & beta).*

The vector *unpack.v* contains the first  $d$  elements of *beta* and row  $j$  and column  $k$  of *unpack.m* is element  $d + k + j(j+3)/2$  of *beta* for nonnegative integers  $k \leq j < d$ .

## **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. The function declaration is

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

Here *model* has the definition

```
struct model{char type; char transform};
```

In *choice*, *choice.type* has value *B* for logit beta, *C* for a cumulative case, *D* for a continuous case, *E* for logit Dirichlet, *G* for a graded response, *H* for log gamma, *L* for the multinomial logit case, *M* for the maximum of two independent Bernoulli variables, *N* for the multivariate normal case, *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`, `loggamma.cpp`, `logitbeta.cpp`, `logitdirichlet.cpp`, `logmean.cpp`, `maxberresp.cpp`, `multlogit.cpp`, `ranklogit.cpp`, and `truncresp.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 int & order, const vector<dat> & data, const xsel & obssel,
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; uvec 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* is 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. If `obs.sel.all` is `true`, then all observations `data[i]` are used. Otherwise, `data[i]` is only used for  $i$  in `obs.sel.list`.

The function `genresplik.cpp` uses `addsel.cpp`, `linsel.cpp`, `vecsel.cpp`, `genresp.cpp`, and all C++ functions `genresp.cpp` requires.

### **genrespmle.cpp**

The function `genrespmle.cpp` applies maximizes the log-likelihood function, gradient, and Hessian matrix of `genresplik.cpp`. The function declaration is

```
maxf2v genrespmle(const int & order, const params & mparams,  
const char & algorithm, const vector<dat> & data, const xsel & obs.sel, const vec &  
start).
```

Here the structs `maxf2v` and `mparams` are defined as in `maxlinq2.cpp` and `maxf2vvar.cpp`. The vector `start` is the starting vector. The variable `algorithm` determines the algorithm, with `N` for Newton-Raphson, `L` for Newton-Raphson with the Hessian approximation of Equation 4, `C` for conjugate gradient, and `G` for gradient ascent. The functions `nrv.cpp`, `conjgrad.cpp`, `gradascent.cpp`, and `genresplik.cpp` are required, together with all C++ functions that these four functions need.

## **Tools for Computation of Log Likelihood Functions**

### **addsel.cpp**

The function `addsel.cpp` is used to add `f2v` structures. The function declaration is

```
void addsel(const int & order, const xsel & xselect,  
const f2v & x, f2v & y, const double & a).
```

Here `order` and `xselect` are defined as in Log-likelihood Components. The struct `y` is modified by use of the struct `x` and the multiplier `a`. In all cases, `ax.value` is added to `y.value`. If `xselect.all` is `true`, then `x` and `y` have compatible dimensions, `ax.grad` is added to `y.grad` if `order` is at least 1, and `ax.hess` is added to `y.hess` if `order` is at least 2. If `xselect.all` is `false`, then `ax.grad` is added to `y.grad.elem(xselect.list)` if `order` is at least 1 and `ax.hess` is added to `y.hess.submat(xselect.list,xselect.list)` if `order` is at least 2.

**ivecsel.cpp**

The function `ivecsel.cpp` is employed to create a new integer vector from an old vector by extracting of elements of the old integer vector. The function declaration is

*ivec ivecsel(const xsel & xselect, const ivec & y).*

Here the struct `xsel` is defined as in `genresplik.cpp`. If `xselect.all` is *true*, then `ivecsel` is `y`. Otherwise, `ivecsel` is a vector with the number of elements in `xselect.list`, and element  $i$  of `ivecsel` is element `xselect.list(i)` of `y`.

**linsel.cpp**

The function `linsel.cpp` is used to apply a linear transformation to an `f2v` struct. The function declaration is

*f2v linsel(const int & order, const f2v & x, const mat & a).*

Here `order` is defined as in Log-likelihood Components. It is always the case that `linsel.value` is `x.value`. If `order` is positive, then `linsel.grad` is the product of the transpose of `a` and the gradient `x.grad`. If `order` exceeds 1, then `linsel.hess` is the product of the transpose of `a`, the Hessian `x.hess`, and the matrix `a`.

**vecsel.cpp**

The function `vecsel.cpp` is employed to create a new vector from an old vector by extracting of elements of the old vector. The function declaration is

*vec vecsel(const xsel & xselect, const vec & y).*

Here the struct `xsel` is defined as in `genresplik.cpp`. If `xselect.all` is *true*, then `vecsel` is `y`. Otherwise, `vecsel` is a vector with the number of elements in `xselect.list`, and element  $i$  of `vecsel` is element `xselect.list(i)` of `y`.

**Latent Structures**

In this section, functions useful for analysis of latent structures are considered. The log-likelihood function in this section is defined based on the definitions in Log-likelihood Components; however, use of latent variables is involved. In typical cases, data involve multiple responses for each individual observation. For a positive integer  $m$ ,  $m$  observations are present. For observation  $h$ ,  $0 \leq h < m$ , the observation has weight  $w_{h*} > 0$ , and  $n_h$  responses are observed. In addition, a latent vector appears in the model. Associated with the latent vector are positive integers  $q_*$  and

$r_*$ ,  $A_*$  in  $\mathcal{A}$ , and  $F_*$  in  $\mathcal{F}$ . The latent vector  $\boldsymbol{\theta}_h$  is in  $\mathcal{Y}(A_*, F_*, q_*, r_*)$ . The latent variable is predicted by the  $q_*$  by  $p$  predicting matrix  $\mathbf{X}_{h*}$  in the nonempty set  $\mathcal{X}_*$  and the fixed  $q_*$ -dimensional vector  $\mathbf{o}_*$ . It is assumed that  $\boldsymbol{\lambda}_*(\boldsymbol{\tau}) = \mathbf{o}_* + \mathbf{X}_*\boldsymbol{\tau}$  is in  $O(A_*, F_*, q_*, r_*)$  as long as  $\mathbf{X}_*$  is in  $\mathcal{X}_*$  and  $\boldsymbol{\tau}$  is in  $O$ . For response  $i$ ,  $0 \leq i < n_h$ , positive integers  $q_{hi}$  and  $r_{hi}$  are given. The variable  $A_{hi}$  is in  $\mathcal{A}$  and  $F_{hi}$  is in  $\mathcal{F}$ . The component of the log likelihood for response  $i$  involves the predicted random vector  $\mathbf{Y}_{hi}$  in  $\mathcal{Y}(A_{hi}, F_{hi}, q_{hi}, r_{hi})$ , the latent vector  $\boldsymbol{\theta}_h$ , the  $q_{hi}$  by  $p$  predicting matrix  $\mathbf{X}_{hi}$  in a nonempty set  $\mathcal{X}_{hi}$ , the  $q_{hi}$ -dimensional vector  $\mathbf{o}_{hi}$ , the  $q_{hi}$  by  $q_*$  matrix  $\mathbf{D}_{hi}$ , the positive real weight  $w_{hi}$ , the  $q_{hi}$  by  $p$  matrix  $\mathbf{D}_{hik}$ ,  $0 \leq k < q_*$ , and the function  $\ell_c(\cdot; \mathbf{y}, A_{hi}, F_{hi}, q_{hi}, r_{hi})$  on  $O(A_{hi}, F_{hi}, q_{hi}, r_{hi})$  defined for  $\mathbf{y}$  in  $\mathcal{Y}(A_{hi}, F_{hi}, q_{hi}, r_{hi})$ . For any  $\boldsymbol{\tau}$  in  $O$ ,  $\mathbf{X}$  in  $\mathcal{X}_{hi}$ , and  $\boldsymbol{\theta}$  in  $\mathcal{Y}(A_*, F_*, q_*, r_*)$ ,

$$\boldsymbol{\lambda}_{hi}(\boldsymbol{\tau}|\boldsymbol{\theta}) = \mathbf{o}_{hi} + \mathbf{X}_{hi}\boldsymbol{\tau} + \mathbf{D}_{hi}\boldsymbol{\theta} + \sum_{k=0}^{p-1} \theta_k \mathbf{D}_{hik}\boldsymbol{\tau} \quad (18)$$

is in  $O(A_{hi}, F_{hi}, q_{hi}, r_{hi})$ .

For  $\boldsymbol{\tau}$  in  $O$ , the log-likelihood has the form

$$\ell(\boldsymbol{\tau}) = \sum_{h=0}^{m-1} w_{h*} \ell_h(\boldsymbol{\tau}), \quad (19)$$

where  $\ell_h(\boldsymbol{\tau})$  is the component of the log-likelihood for observation  $h$ . Thus the gradient function of  $\ell$  at  $\boldsymbol{\tau}$  satisfies

$$\nabla \ell(\boldsymbol{\tau}) = \sum_{h=0}^{m-1} w_{h*} \nabla \ell_h(\boldsymbol{\tau}), \quad (20)$$

where  $\nabla \ell_h(\boldsymbol{\tau})$  is the gradient function of  $\ell_h$  at  $\boldsymbol{\tau}$ . The Hessian function of  $\ell$  at  $\boldsymbol{\tau}$  satisfies

$$\nabla^2 \ell(\boldsymbol{\tau}) = \sum_{h=0}^{m-1} w_{h*} \nabla^2 \ell_h(\boldsymbol{\tau}), \quad (21)$$

where  $\nabla^2 \ell_h(\boldsymbol{\tau})$  is the Hessian function of  $\ell_h$  at  $\boldsymbol{\tau}$ . The approximation

$$\tilde{\nabla}^2 \ell(\boldsymbol{\tau}) = - \sum_{h=0}^{m-1} w_{h*} \nabla \ell_h(\boldsymbol{\tau}) [\nabla \ell_h(\boldsymbol{\tau})]^T, \quad (22)$$

may also be considered.

In turn,  $\ell_h(\boldsymbol{\tau})$  involves the product

$$\ell_h(\boldsymbol{\tau}|\boldsymbol{\theta}) = \ell_c(\boldsymbol{\lambda}_*(\boldsymbol{\tau}); \boldsymbol{\theta}, A_*, F_*, q_*, r_*) \sum_{i=0}^{n_h-1} w_{hi} \ell_c(\boldsymbol{\lambda}_{hi}(\boldsymbol{\tau}|\boldsymbol{\theta}); \mathbf{Y}_{hi}, A_{hi}, F_{hi}, q_{hi}, r_{hi}) \quad (23)$$

for  $\boldsymbol{\theta}$  in  $\mathcal{Y}(A_*, F_*, q_*, r_*)$ . The component

$$\ell_h(\boldsymbol{\tau}) = \log \int (\exp(\ell_h(\boldsymbol{\tau}|\cdot))), \quad (24)$$



where  $\exp(\ell_h(\boldsymbol{\tau}|\cdot))$  is the function with value  $\exp(\ell_h(\boldsymbol{\tau}|\boldsymbol{\theta}))$  for  $\boldsymbol{\theta}$  in  $\mathcal{Y}(A_*, F_*, q_*, r_*)$ . In practice,  $\ell_h(\boldsymbol{\tau})$  is evaluated by

$$\tilde{\ell}_h(\boldsymbol{\tau}) = \log \left[ \sum_{k=1}^Q u_{hk} \exp(\ell_h(\boldsymbol{\tau}|\boldsymbol{\theta}_{hk})) \right], \quad (25)$$

for some positive weights  $u_{hk}$  and elements  $\boldsymbol{\theta}_{hk}$  in  $\mathcal{Y}(A_*, F_*, q_*, r_*)$ .

### **irtc.cpp**

The function `irtc.cpp` finds the conditional log likelihood component  $\ell_h(\boldsymbol{\beta}|\boldsymbol{\theta})$  and associated gradient and Hessian matrix for a latent structure model. The function declaration is

```
f2v irtc (const int & order, const vector<dat> & data,
const vector<thetamaps> & thetamaps, const resp & theta,
const xsel & datasel, const vec & beta).
```

In this declaration, *order* is less than 1 if only the function value is returned, at least 1 if the gradient is required, 2 if the Hessian is produced, and more than 2 if the approximate Hessian matrix is found. The use of *dat* is as in `genresplik.cpp`, the latent vector value is provided by *theta*, and *beta* is defined as usual as the parameter vector. Use of *datasel* is the same as use of *obssel* in `genresplik.cpp`.

The struct *thetamaps* is defined by

```
struct thetamaps{bool dep; xsel drespcols; xsel irespcols; mat offsets; cube indeps;}.
```

Here *dep* is true if the response is derived from *theta*, *drespcols* gives the elements of *theta.dresp* used in the response, and *irespcols* gives the elements of *theta.iresp* in the response. For item *h* and observation *i*, the matrix *thetamaps[i].offsets* contains each column *k* of  $\mathbf{D}_{hi}$  such that *k* is specified in *thetamaps[i].drespcols*, and other columns of  $\mathbf{D}_{hi}$  are zero vectors. In the cube *thetamaps[i].indeps*, each  $\mathbf{D}_{hik}$  is given for *k* specified in *thetamaps[i].drespcols*, and column *j* of  $\mathbf{D}_{hik}$  is provided if *j* is in the elements specified in *theta[i].drespcols* for item *h* and observation *i*. Other columns of  $\mathbf{D}_{hik}$  are vectors of zeros. Element *i* of *data* corresponds to  $\mathbf{Y}_{hi}$ .

The functions `ivecsel.cpp` and `genresplik.cpp`, together with their associated functions, are required by `irtc.cpp`.

### **irtcomps.cpp**

The function `irtcomps.cpp` finds the log likelihood components  $\ell_h(\boldsymbol{\tau})$  and associated gradients and Hessian matrices for a latent structure model. The function uses numerical integration if  $\mathcal{Y}(A_*, F_*, q_*, r_*)$  is not finite or countably infinite. The function declaration is

*vector<f2v> irtcomps (const int & order, const vector<vector<dat> > & obsdata, const vector<vector<thetamap> > & obstheta maps, const vector<xsel> & dataset, const xsel & obs sel, const adq & scale, vector<rescale> & obsscale, const vector<pwr> & thetas, const vector<xsel> & betasel, const vec & beta).*

In this declaration, only the function value is returned if *order* is 0, the function value and gradient are returned if *order* is 1, and the function value, gradient, and Hessian matrix are returned if *order* is 2. The function value, gradient, and Louis approximation for the Hessian matrix are returned if *order* is 3. The use of *dat* is as in *genresplik.cpp*, and *beta* is defined as usual as the parameter vector. The struct *thetamap* is defined as in *irtc.cpp*. The definition of *dataset[h]* corresponds in *irtc.cpp* to the definition of *dataset*. The struct *obs sel* indicates which observations *h* are used, with *obs sel.all* equal *true* if all observations are used. If *obs sel.all* is *false*, then the observations specified by *obs sel.list* are employed. For each *h*, *betasel[h]* indicates the elements of *beta* applied to  $\ell_h(\beta)$ .

The struct *pwr* is defined by

```
struct pwr{double weight; resp theta;},
```

the struct *adq* is

```
struct adq{bool adapt; xsel xselect; double step;},
```

and the struct *rescale*

```
struct rescale{double mult; vecmat tran},
```

The structs *scale* and *rescale* are important in adaptive quadrature (Naylor & Smith, 1982) and are used in *irtmle.cpp* to improve efficiency and accuracy of computations involving continuous latent vectors. Let *thetas* have *Q* elements *thetas[k]* such that *thetas[k].theta* corresponds to the  $q_*$ -dimensional vector  $\theta_k$ . If *scale.adapt* is *false*, *scale.xselect.all* is *false* and *scale.xselect.list* has no elements, or *scale.xselect.all* is *true* and *thetas[0].theta.dresp* has no elements, then  $u_{hk}$  is *thetas[k].weight* and  $\theta_{hk}$  is  $\theta_k$ . Consider the case of *scale.adapt* equal *true*, *thetas[0].theta.dresp* has a positive number of elements, and either *scale.xselect.all* is *true* or *scale.xselect.list* has a positive number of distinct nonnegative integer elements less than  $q_*$  such that each element of *scale.xselect.list* corresponds to an element of *thetas[0].theta.dresp*. Then  $\theta_{hk} = \mathbf{v}_h + \mathbf{M}_h \theta_k$  for a  $q_*$ -dimensional vector  $\mathbf{v}_h$  and a symmetric  $q_*$  by  $q_*$  matrix  $\mathbf{M}_h$ . An element of  $\mathbf{v}_h$  is forced to be 0 if the corresponding element of  $\theta_0$  does not correspond to an element of *thetas[0].theta.dresp* and, in the case of *scale.xselect.all* equal *false*, does not correspond to an element of *thetas[0].theta.dresp* specified by *scale.xselect.list*. All members of a row of  $\mathbf{M}_h$  are forced to be 0 if the corresponding

element of  $\mathbf{v}_h$  is forced to be 0. The elements of  $\mathbf{v}_h$  not forced to be 0 are given in *obsscale[h].tran.v*, and elements of  $\mathbf{M}_h$  not forced to be 0 are given by *obsscale[h].tran.m*. The matrix *obsscale[h].tran.m* is positive-definite and *obsscale[h].mult* is its determinant. In *irtcomps.cpp*, *obsscale* is given. Its computation is discussed in *irtm.cpp*. The function *irtcomps.cpp* uses the functions *irtc.cpp*, *irtm.cpp*, *adaptpwr.cpp*, and *nrvn.cpp*, together with their required functions.

### **irtm.cpp**

The function *irtm.cpp* finds the log likelihood component  $\ell_h(\boldsymbol{\tau})$  and associated gradient and Hessian matrix for a latent structure model. The function uses numerical integration if  $\mathcal{Y}(A_*, F_*, q_*, r_*)$  is not finite or countably infinite. The function declaration is

```
f2v irtm (const int & order, const vector<dat> & data,
const vector<thetamap> & thetamaps, const adq & scale, const params & mparamsn,
rescale & newscale,
const vector<pwr> & thetas, const xsel & datasel, const vec & beta).
```

In this declaration, only the function value is returned if *order* is 0, the function value and gradient are returned if *order* is 1, and the function value, gradient, and Hessian matrix are returned if *order* is 2. The function value, gradient, and Louis approximation for the Hessian matrix are returned if *order* is 3. The use of *dat* is as in *genresplik.cpp*, and *beta* is defined as usual as the parameter vector. The struct *thetamap* is defined as in *irtc.cpp*. The definition of *datasel* is as in *irtc.cpp*. The definitions of *scale*, *rescale*, and *thetas* are the same as in *irtcomps.cpp*. The definition of *params* is the same as in *maxlinq2.cpp*.

In contrast to *irtcomps.cpp*, computation of *newscale* is undertaken in *irtm.cpp* in the case of *scale.adapt* equal *true*, *thetas[0].theta.dresp* with a positive number of elements, and either *scale.xselect.all* equal *true* or *scale.xselect.list* with a positive number of distinct nonnegative integer elements less than  $q_*$  such that each element of *scale.xselect.list* corresponds to an element of *thetas[0].theta.dresp*. Consider the case that *scale.adapt* equals *true*, *thetas[0].theta.dresp* has a positive number of elements, and either *scale.xselect.all* is *true* or *scale.xselect.list* has a positive number of distinct nonnegative integer elements less than  $q_*$  such that each element of *scale.xselect.list* corresponds to an element of *thetas[0].theta.dresp*. The vector  $\mathbf{v}_h$  is selected to maximize  $\ell_h(\boldsymbol{\tau}|\boldsymbol{\theta})$  subject to the constraint that an element of  $\boldsymbol{\theta}$  must equal the corresponding element of  $\boldsymbol{\theta}_0$  when that element of  $\mathbf{v}_h$  must be 0. Let this location of the maximum be  $\hat{\boldsymbol{\theta}}_{h0}$ . This maximization is normally accomplished with *nrvn.cpp* with step size *scale.mult*. The starting value is a previously specified *newscale.tran.v*. The matrix *newscale.tran.m* is normally the symmetric matrix square root of the inverse of the matrix formed from the rows and columns of the negative Hessian matrix of  $\ell_h(\boldsymbol{\tau}|\cdot)$  at  $\hat{\boldsymbol{\theta}}_{h0}$  that correspond to elements of  $\mathbf{v}_h$  not restricted to

be 0, and *rescale.mult* is the corresponding determinant. If the desired symmetric square root does not exist, then *newscale* is unchanged from its previous value. The function *irtm.cpp* uses the functions *irtc.cpp*, *adaptpwr.cpp*, and *nrvn.cpp*, together with their required functions. The parameters in *mparamsn* are used with *nrvn.cpp* to find *newscale.tran.v*.

## **irtmle.cpp**

The function *irtmle.cpp* finds the maximum likelihood estimate for a latent structure model. The function uses numerical integration if  $\mathcal{Y}(A_*, F_*, q_*, r_*)$  is not finite or countably infinite. The function declaration is

```
maxf2v irtmle (const int & order, const params & mparams,
const char & algorithm, const vec & obsweight,
const vector<vector<dat> > & obsdata,
const vector<vector<thetamap> > & obstheta maps,
const vector<xsel> & datasel, const xsel & obssel,
const adq & scale, const params & mparamsn, vector <rescale> & obsscale,
const vector<pwr> & thetas, const vector<xsel> & betasel, const vec & start).
```

In this declaration, *maxf2v*, *mparams*, and *mparamsn* are defined as in *maxliq2.cpp* and *maxf2vvar.cpp*, while *order* is defined as in *irtc.cpp*. The parameters in *mparamsn* are used when adaptive quadrature is specified, whereas *mparams* is used for the basic iterations used for determination of the maximum-likelihood estimate. The variable *algorithm* is defined as in *genrespmle.cpp*. The vector *obsweight* provides the weights  $w_{h*}$  for  $0 \leq h \leq m - 1$ . The struct *dat* is defined as in *genresplik.cpp*, *obsdata[h]* is the data specification for observation  $h$ , the struct *thetamap* is defined as in *irtc.cpp*, *obstheta maps[h]* provides the mapping of the latent vector for observation  $h$ , the struct *pwr* is defined as in *irtm.cpp*, *thetas* is defined as in *irtm.cpp*, the struct *adq* is defined as in *irtm.cpp*, the scale adjustment for observation  $h$  is defined by *obsscale[h]*, the elements of *data[h]* to be used are defined as in *datasel[h]*, and the observations  $h$  used are defined by *obssel*. If *obssel.all* is *true*, then all observations are used. Otherwise, observation  $h$  only is used if in *obssel.list*. The vector *beta* is defined as usual as the parameter vector. The value of *betasel[h].all* is *true* if all elements of *beta* are used for the log likelihood for observation  $h$ , and the elements of *beta* corresponding to *betasel[h].list* are used for this log likelihood if *betasel[h].all* is *false*.

The functions *irtms.cpp* and its required functions and the functions that are prerequisites for *genrespmle.cpp* are required by *irtmle.cpp*.

**irtms.cpp**

The function `irtms.cpp` finds the log likelihood component  $\ell(\boldsymbol{\tau})$  and associated gradient and Hessian matrix for a latent structure model. The function uses numerical integration if  $\mathcal{Y}(A_*, F_*, q_*, r_*)$  is not finite or countably infinite. The function declaration is

```
f2v irtms (const int & order, const vec & obsweight,  
const vector<vector<dat> > & obsdata,  
const vector<vector<thetamap> > & obsthetamaps,  
const vector<xsel> & datasel, const xsel & obsel, const adq & obsscale,  
const params & mparamsn, vector<rescale> obsscales, const vector<pwr> & thetas,  
const vector<xsel> & betasel, const vec & beta).
```

In this declaration, `order` is less than 1 if only the function value is returned, at least 1 if the gradient is required, 2 if the Hessian is produced, and more than 2 if the approximate Hessian matrix is found. The use of `obsweight`, `obsdata`, `obsthetamaps`, `datasel`, `obsel`, `scale`, `mparamsn`, `obsscales`, `thetas`, and `betasel` are as in `irtmle.cpp`, and `beta` is defined as usual as the parameter vector. The function `irtm.cpp` and its required functions are used by `irtms.cpp`.

**posterior.cpp**

For an observation, the function `posterior.cpp` finds the posterior distribution of the latent vector given the observed responses. The function declaration is

```
vecmat posterior (const vector<dat> & data,  
const vector<thetamap> & thetamaps, const adq & scale, const rescale & newscale,  
const vector<pwr> & thetas, const xsel & datasel, const vec & beta).
```

Definitions are consistent with those in `irtm.cpp`. The posterior is presented as  $q$  points, each of which has  $r$  elements. Output is based on `thetas`. The number  $q$  of points is the number of elements of `thetas`, and  $r$  is the number of elements of `thetas[0].theta.dresp`. The output consists of the vector `posterior.v` with  $q$  elements corresponding to the probabilities assigned to each point and the  $q$  by  $r$  matrix `posterior.m` with each row corresponding to a point of dimension  $r$ . All functions required by `adaptpwr.cpp` and `irtc.cpp` are required by `posterior.cpp`.

**posteriors.cpp**

For each observation used in `irtms.cpp`, the function `posteriors.cpp` finds the posterior distribution of the latent vector given the observed responses. The function declaration is

*vector<vecmat> posteriors (const vector<vector<dat> > & obsdata,  
const vector<vector<thetamap> > & obstheta, const vector<xsel> & dataset,  
const xsel obsel, const adq & scale, const vector<rescale> & obsscale,  
const vector<pwr> & thetas, const vector<xsel> & betasel, const vec & beta).*

Definitions are consistent with those in irtms.cpp. The posteriors for each observation are presented as in posterior.cpp. All functions required by posterior.cpp are required by posteriors.cpp.

## Integration Tools

### adaptpwr.cpp

The function adaptpwr.cpp provides a linear transformation of a quadrature point. 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

*pwr adaptpwr(const pwr & oldtheta, const adq & scale, const rescale & newscale).*

The structs *pwr*, *adq*, and *rescale* are defined as in irtm.cpp. If *scale.adapt* is *false*, *oldtheta.theta.dresp* has no elements, or *scale.xselect.all* is *false* and *scale.xselect.list* has no elements, then *adaptpwr* is *oldtheta*. In other cases, *adaptpwr.weight* is the product of *oldtheta.weight* and *newscale.mult*, *adaptpwr.theta.iresp* is *adaptpwr.theta.iresp*, elements of *adaptpwr.theta.dresp* that do not correspond to elements of *scale.tran.v* are the same as corresponding elements of *oldtheta.theta.dresp*, and the remaining elements of *adaptpwr.theta.dresp* correspond to *rescale.tran.v* plus the product of *rescale.tran.m* and elements of *oldtheta.theta.dresp* that correspond to elements of *rescale.tran.v*.

### eaps.cpp

For each observation  $i$ , the function eaps.cpp generates a posterior weighted mean *eaps[i].v* and covariance matrix *eaps[i].m* that correspond to a discrete posterior distribution with points *posts[i].m.col(j)* with probabilities *posts[i].v(j)* for  $j$  from 1 to *posts[i].m.n\_cols*. The function declaration is

*vector<vecmat> eaps(const vector<vecmat> & posts).*

The definition of *vecmat* is as in pack.cpp. The function *wmc.cpp* is used.

### 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

*vecmat* *genprods*(*const imat* & *indices*, *const vector*<*pw*> & *pws*).

The struct *vecmat* is defined as in *pack.cpp*, and the struct *pw* has *vec* elements *points* and *weights*. 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.m* has  $Q$  columns and *genprods.v* has  $Q$  elements. The matrix *genprods.m* 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.m* is *pws*[ $d$ ].*points*(*indices*( $d,k$ )). Element  $k$  of *genprods.v* 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 degree. 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  $\alpha_i$ . 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 degree at a specified real value. The function declaration is

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

The degree 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(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*. The weights are relative to the standard normal density.

**normwt.cpp**

The function `normwt.cpp` divides quadrature weights by the standard normal density to facilitate use with latent-structure models with latent variables that are normally distributed. The function declaration is

*pw normwt(const pw & pwi).*

The struct *normwt* has vector elements *normwt.points* and *normwt.weights*. The input *pwi* has elements *pwi.points* and *pwi.weights*. The ordered quadrature points are in *normwt.points*. The corresponding weights are in *normwt.weights*. The result *normwt.points* is the same as *pwi.points*, but the weights *normwt.weights* are obtained from *pwi.weights* by dividing by the standard normal density at the corresponding points *pwi.points*.

**qnormpw.cpp**

The function `qnormpw.cpp` provides normal-scores quadrature of a given order. The function declaration is

*pw qnormpw(const int &n).*



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

### **wmc.cpp**

The function `wmc.cpp` computes a weighted mean vector and covariance matrix. The function declaration is

```
vecmat wmc(const vecmat & wx).
```

The elements of `wx.v` are probabilities corresponding to the rows of `wx.m`.

### **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 No. 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.
- Naylor, J. C., & Smith, A. F. M. (1982). Applications of a method for the efficient computation of posterior distributions. *Applied Statistics*, *31*, 214–225. <https://doi.org/10.2307/2347995>.
- 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++. In *Mathematical software – ICMS 2018* (pp. 422–430). [https://doi.org/10.1007/978-3-319-96418-8\\_50](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>.