References

Bryk, Anthony S., and Steven W. Raudenbush (1992). *Hierarchical Linear Models: applications and data analysis methods.* Newbury Park, California: Sage.

Bryk, A. S., Raudenbush, S. W., Seltzer, M., & Congdon, R. (1988). *An introduction to HLM: Computer program and user's guide* (2nd ed.). Chicago: University of Chicago Department of Education.

Hilden-Minton, J. (1993). Mixed Model Diagnostics via Case-Deletion. (Unpublished).

Longford, N. T. (1987). A Fast Scoring Algorithm for Maximum Likelihood Estimation in Unbalance Mixed Models with Nested Effects. *Biometrika*, 74, 817-827.

Tierney, L. (1990). *Lisp-Stat: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* Wiley: New York, NY.

```
         (n (send self :n-cases)))
    (send self :xx (- (+ xx (outer-product (* n c) c))
                      (+ xc (transpose xc))))
    (send self :xy (- xy (* (elt xy 0) c)))))

(defmeth reg-unit-proto :center-x (c)
"Method Args: (c)
Centers level-one variables about list or vector C."
  (let ((one (repeat 1 (send self :n-cases)))
        (x (send self :x)))
    (send self :x (- x (outer-product one c)))
    (call-method unit-proto :center-x c)))

(defmeth terrace-proto :center-x (c)
"Method Args: (c)
Centers level-one variables about list or vector C."
  (send self :map-units :center-x c))

(defmeth unit-proto :center-group-mean (&optional ind)
"Method Args: (&optional ind)
Centers level-one variables IND on group-means."
  (let* ((x-mean (/ (send self :x-sum) (send self :n-cases)))
         (ind (remove 0 (if ind ind (iseq (length x-mean)))))
         (c (* 0 x-mean)))
    (setf (select c ind) (select x-mean ind))
    (send self :center-x c)))

(defmeth terrace-proto :center-group-mean ()
"Method Args: ()
Dialog method for group-mean centering."
  (let ((choice (car (choose-subset-dialog
                       "Choose for group mean centering"
                       (rest (send self :x-labels))))))
    (if choice
      (send self :map-units :center-group-mean (+ 1 choice)))))

#|
(defmeth terrace-proto :model-criticism ()
"Method Args: ()
Returns predictive model criticism of each unit. This is a
measure of how well the empirical prior anticipates the
observations in a particular unit."
  (let ((sigma (send self :sigma)))
    (send self :map-units :model-criticism sigma)))

(defmeth terrace-proto :z-mean-wgt ()
"Method Args: ()
Returns case-weighted means of level-two variables."
  (let ((n (send self :map-units :n-cases))
        (z (send self :map-units :z)))
    (/ (apply #'+ (* n z)) (sum n))))

(defmeth terrace-proto :center-z (c)
"Method Args: (c)
Centers level-two variables about list or vector C."
  (mapcar #'(lambda (u z) (send u :z (- z c)))
    (send self :units)
    (send self :map-units :z)))

(defmeth unit-proto :x-sum ()
"Method Args: ()
Returns of the sum of level-one variables."
  (car (row-list (send self :xx))))

(defmeth terrace-proto :x-mean-wgt ()
"Method Args: ()
Returns case-weighted means of level-one variables."
  (/ (send self :sum-units :x-sum) (send self :n-cases)))

(defmeth unit-proto :center-x (c)
"Method Args: (c)
Centers level-one variables about list or vector C."
  (let* ((xx (send self :xx))
         (xy (send self :xy))
         (xc (outer-product (send self :x-sum) c))
    (send self :map-units :gamma-cd-diff dispersion)))

(defmeth cd-unit-proto :u-hat-cd-diff (&optional tau dispersion)
"Method Args: ()
Returns the case deleted differences in u-hat."
  (let ((tau (if tau tau (send self :terrace :tau)))
        (dis (if dispersion dispersion
               (send self :terrace :dispersion)))
        (x-check (send self :x-check dis))
        (theta (send self :theta)))
    (* theta (row-lists (matmult x-check tau)))))

(defmeth terrace-proto :u-hat-cd-diff ()
"Method Args: ()
Returns the case deleted differences in u-hat."
  (let ((tau (send self :tau))
        (dis (send self :dispersion)))
    (send self :map-units :u-hat-cd-diff tau)))

#|
```

7.2 Model Criticism

Code 7.2

```
|#
(defmeth unit-proto :model-criticism (&optional sigma)
"Method Args: ()
Returns predictive model criticism of unit. This is a measure
of how well the empirical prior anticipates the observations
in a particular unit."
  (let ((sig (if sigma sigma (send self :terrace :sigma)))
        (dmd (send self :dmd))
        (n (send self :n-cases)))
    (- 1 (chisq-cdf (/ dmd sig) n))))
  (let ((sigma (send self :sigma)))
    (send self :map-units :studentized-residuals sigma)))

(defmeth cd-unit-proto :cook-gamma (&optional sigma)
"Method Args: ()
Returns Cook like distances for influence of case-deletion on
estimates of gamma."
  (let ((sig (if sigma sigma (send self :terrace :sigma)))
        (theta (send self :theta))
        (h (send self :h)))
    (/ (* theta theta h) sig)))

(defmeth cd-unit-proto :cook-u (&optional sigma)
"Method Args: ()
Returns Cook-like distances for influence of case-deletion on
estimate of u."
  (let ((sig (if sigma sigma (send self :terrace :sigma)))
        (theta (send self :theta))
        (k (send self :k)))
    (/ (* theta theta k) sig)))

(defmeth cd-unit-proto :leverage-gamma ()
"Method Args: ()
Returns generalized leverages on gamma."
  (/ (send self :h) (send self :s)))

(defmeth cd-unit-proto :gamma-cd-diff (&optional dispersion)
"Method Args: ()
Returns the case deleted differences in estimates of gamma."
  (let ((d (if dispersion dispersion
             (send self :terrace :dispersion)))
        (wxt (send self :pre-w :x-tilde-trans))
        (theta (send self :theta)))
    (* theta (column-lists (matmult d wxt)))))

(defmeth terrace-proto :gamma-cd-diff ()
"Method Args: ()
Returns the case deleted differences in estimates of gamma."
  (let ((dispersion (send self :dispersion)))
           (if dispersion dispersion
             (send self :terrace :dispersion)))
         (wxt (send self :pre-w :x-tilde-trans))
         (dwxt (matmult dispersion wxt)))
    (setf (slot-value 'h)
      (mapcar #'inner-product
        (column-lists wxt) (column-lists dwxt)))))

(defmeth cd-unit-proto :k ()
"Method Args: ()
Returns the value of k."
  (let* ((xmx (send self :xmx))
         (wxmx (send self :pre-w :xmx))
         (dis (send self :terrace :dispersion))
         (mat (inverse
                (- xmx (matmult (transpose wxmx) dis wxm)))))
    (mapcar #'(lambda (r) (matmult r mat r))
      (send self :x-check))))

(defmeth cd-unit-proto :theta ()
"Method Args: ()
Returns the values of theta which is r/(s - h)."
  (let ((r (send self :residuals))
        (s (send self :s))
        (h (send self :h)))
    (/ r (- s h))))

(defmeth cd-unit-proto :studentized-residuals (&optional sigma)
"Method Args: ()
Returns internally studentized residuals, r/(sig*sqrt(s-h))."
  (let ((r (send self :residuals))
        (sig (if sigma sigma (send self :terrace :sigma)))
        (s (send self :s))
        (h (send self :h)))
    (/ r (sqrt (* sig (- s h))))))

(defmeth terrace-proto :studentized-residuals ()
"Method Args: ()
Returns internally studentized residuals, r/(sig*sqrt(s-h))."
```

```
(slot-value 'x-tilde))

(defmeth cd-unit-proto :x-tilde-trans ()
"Method Args: ()
Returns the transpose of the contents of slot x-tilde."
  (transpose (send self :x-tilde)))

(defmeth cd-unit-proto :x-check (&optional dispersion)
"Method Args: ()
Returns x-check."
  (let ((dis (if dispersion dispersion
               (send self :terrace :dispersion)))
        (wxt (send self :pre-w :x-tilde-trans))
        (wxmx (send self :pre-w :xmx))
        (x (send self :x-tilde)))
    (- x-tilde (matmult (transpose wxt) dis wxmx))))

(defmeth cd-unit-proto :h ()
"Method Args: ()
Returns the content of slot h."
(slot-value 'h))

(defmeth cd-unit-proto :s ()
"Method Args: ()
Returns the content of slot s."
(slot-value 's))

(defmeth cd-unit-proto :update-cd-stats (&optional dispersion)
"Method Args: ()
Updates the contents of slots x-tilde, h and s."
  (let* ((xx (send self :xx))
         (x (send self :x))
         (c-inv (send self :c-inv))
         (xc (matmult x c-inv)))
    (setf (slot-value 's)
      (- 1 (mapcar #'inner-product
             (row-lists xc) (row-lists x))))
    (setf (slot-value 'x-tilde) (- x (matmult xc xx))))
  (let* ((dispersion
```

where $\theta_i = \frac{\hat{r}_i}{s_i - h_i}$.

Code 7.1

```
|#

;;;;
;;;;    Define Unit Prototype with Case-Deletion Capabilities
;;;;

(defproto cd-unit-proto
 '(x-tilde       ;  X(I - C-inv X'X)
   h             ;  x-tilde WDW' x-tilde
   s             ;  1 - x C-inv x
  )
  ()
  reg-unit-proto
"Diagnosable unit prototype with case-deletion capabilities")

;;;
;;;   Basic Methods
;;;

(defmeth reg-unit-proto :begin-case-deletion ()
"Method Args: ()
Prepares an instance of reg-unit-proto for a
case-deletion analysis."
  (send self :reparent cd-unit-proto)
  (send self :update-cd-stats))

(defmeth terrace-proto :begin-case-deletion ()
"Method Args: ()
Prepares instances of reg-unit-proto in a terrace model
for a case-deletion analysis."
  (send self :map-units :begin-case-deletion))

(defmeth cd-unit-proto :x-tilde ()
"Method Args: ()
Returns the content of slot x-tilde."
each unit. Both X-MSG and Y-MSG must be a list containing the
appropriate message such as the respective defaults
'(:y-hat :post-beta) and '(:residuals :post-beta)."
  (let ((x (combine (apply #'send self :map-units x-msg)))
        (y (combine (apply #'send self :map-units y-msg)))
        (labels (combine (send self :map-units :case-labels))))
    (plot-points x y :point-labels labels)))

#|
```

7 Diagnostics

The authors motivation for writing TERRACE was the desire to create a multilevel modelling package with diagnostic capabilities. There are several different approaches to diagnostics: case-deletion, group-deletion, local model perturbation, predictive model criticism, and perhaps others.

7.1 Case-Deletion Diagnostics

First, I will define the prototype `cd-unit-proto` to inherit from `reg-unit-proto`. We include the additional slots `x-tilde`, `h` and `s` to contain

$$\tilde{X} = MX = X(I - C^{-1}X'X),$$

$$h_i = \tilde{x}_i' W D W' \tilde{x}_i,$$

and

$$s_i = 1 - x_i' C^{-1} x_i,$$

respectively, where D is the dispersion matrix. From these three quantities we may compute case-deleted estimates

$$\hat{\gamma} - \hat{\gamma}_{(i)} \approx \theta_i D W' \tilde{x}_i$$

and

$$\hat{u} - \hat{u}_{(i)} \approx \theta_i \tau \tilde{x}_i,$$

```
#|
```

6.3 Plotting Methods

Code 6.3

```
|#

(defmeth terrace-proto :normal-plot (&rest res-msg)
"Method Args: (&rest res-msg)
Produces a normal plot of residuals where the residuals are
the result of sending the message RES-MSG to each unit."
  (let* ((res (combine (apply #'send self :map-units res-msg)))
         (q (normal-quant (/ (1+ (rank res))
                             (1+ (length res)))))
         (labels (combine
                   (send self :map-units :case-labels))))
    (plot-points q res :point-labels labels)))

(defmeth terrace-proto :plot-by-unit (&rest res-msg)
"Method Args: (&rest res-msg)
Produces a plot of residuals, which are obtained by sending
the message RES-MSG to each of the units, against unit of
membership."
  (let ((res (combine (apply #'send self :map-units res-msg)))
        (ns (send self :map-units :n-cases))
        (labels (combine (send self :map-units :case-labels))))
    (plot-points (repeat (iseq 1 (length ns)) ns) res
      :point-labels labels)))

(defmeth terrace-proto :plot-residuals
  (&key (x-msg '(:y-hat :post-beta))
        (y-msg '(:residuals :post-beta)))
"Method Args: (&key x-msg y-msg)
Produces a plot of the result of sending the message X-MSG
to each of the units against the result of sending Y-MSG to
"Method Args: ()
Dialog method for centering level-one variables."
  (let* ((msg (send text-item-proto
                :new "Select a variable for centering"))
         (x-labels (send self :x-labels))
         (x-means (combine (send self :x-mean-wgt)))
         (strings (mapcar #'(lambda (x) (format nil "~8,4f" x))
                    x-means))
         (select-var (send list-item-proto :new x-labels))
         (mean-msgs (send list-item-proto :new strings))
         (dismiss (send modal-button-proto :new "Dismiss"))
         (the-dialog
           (send modal-dialog-proto
             :new (list msg
                        (list select-var mean-msgs)
                        dismiss)
             :title "Center X"
             :default-button dismiss)))
    (send select-var :action
      #'(lambda (x)
          (let* ((ind (send select-var :selection))
                 (string (format nil "Center ~a on the value:"
                           (elt x-labels ind)))
                 (c (get-value-dialog string
                      :initial (elt x-means ind)))
                 (1-list (* 0 x-means)))
            (setf (elt 1-list ind) 1)
            (if c (send self :center-x (* (car c) 1-list))))))
    (send the-dialog :modal-dialog)
    the-dialog))

(defmeth terrace-proto :center-group-mean ()
"Method Args: ()
Dialog method for group-mean centering."
  (let ((choice (car (choose-subset-dialog
                      "Choose for group mean centering"
                      (rest (send self :x-labels))))))
    (if choice
      (send self :map-units :center-group-mean (+ 1 choice)))))
```

```
              :default-button dismiss)))
    (send the-dialog :modal-dialog)
    (send dismiss :action #'(lambda ()
                              (send the-dialog :close)))
    the-dialog))

(defmeth terrace-proto :select-z-center ()
"Method Args: ()
Dialog method for centering level-two variables."
  (let* ((msg (send text-item-proto
               :new "Select a variable for centering"))
         (z-labels (send self :z-labels))
         (z-means (combine (send self :z-mean-wgt)))
         (strings (mapcar #'(lambda (z) (format nil "~8,4f" z))
                    z-means))
         (select-var (send list-item-proto :new z-labels))
         (mean-msgs (send list-item-proto :new strings))
         (dismiss (send modal-button-proto :new "Dismiss"))
         (the-dialog
           (send modal-dialog-proto
             :new (list msg
                        (list select-var mean-msgs)
                        dismiss)
             :title "Center Z"
             :default-button dismiss)))
    (send select-var :action
      #'(lambda (x)
          (let* ((ind (send select-var :selection))
                 (string (format nil "Center ~a on the value:"
                           (elt z-labels ind)))
                 (c (get-value-dialog string
                      :initial (elt z-means ind)))
                 (1-list (* 0 z-means)))
            (setf (elt 1-list ind) 1)
            (if c (send self :center-z (* (car c) 1-list))))))
    (send the-dialog :modal-dialog)
    the-dialog))

(defmeth terrace-proto :select-x-center ()
;    (send the-dialog :modal-dialog)
    (send dismiss :action #'(lambda ()
                              (send the-dialog :close)))
    the-dialog))

(defmeth terrace-proto :centering-dialog ()
"Method Args: ()
Main dialog method for centering variables of a terrace model."
  (let* ((lead-msg (send text-item-proto
                      :new "Centering Options"))
         (comp (send button-item-proto
                  :new "Recompute Estimates"
                  :action #'(lambda () (progn
                                        (send self :mlf)
                                        (send self :display)
                                        (gc)))))
         (center-z (send button-item-proto
                      :new "Center Level-Two Variables"
                      :action
                        #'(lambda () (send self :select-z-center))))
         (center-x (send button-item-proto
                      :new "Center Level-One Variables"
                      :action
                        #'(lambda () (send self :select-x-center))))
         (center-gm (send button-item-proto
                       :new "Group Mean Centering"
                       :action
                         #'(lambda ()
                             (send self :center-group-mean))))
         (dismiss (send modal-button-proto :new "Dismiss"))
         (the-dialog
           (send modal-dialog-proto
             :new (list
                    lead-msg
                    center-gm
                    center-x
                    center-z
                    (list comp dismiss))
               :title "TERRACE-TWO"
```

```
     (tau-msg (send text-item-proto
                :new "Covariance Parameters"))
     (x-labels (send self :x-labels))
     (select-gamma (send list-item-proto :new x-labels))
     (select-tau (send list-item-proto :new x-labels))
     (comp (send button-item-proto
             :new "Recompute Estimates"
             :action #'(lambda () (progn
                                    (send self :mlf)
                                    (send self :display)
                                    (gc)))))
     (rand (send button-item-proto
             :new "Random Effects Model"
             :action
               #'(lambda ()
                   (send self :select-random-effects))))
     (center (send button-item-proto
                 :new "Center Variables"
                 :action
                   #'(lambda ()
                       (send self :centering-dialog))))
     (dismiss (send modal-button-proto :new "Dismiss"))
     (the-dialog
       (send modal-dialog-proto
         :new (list
                lead-msg
                (list gamma-msg tau-msg)
                (list select-gamma select-tau)
                center
                rand
                (list comp dismiss))
           :title "TERRACE-TWO"
           :default-button dismiss)))
(send select-gamma :action
  #'(lambda (x) (send self :select-gamma
                  (send select-gamma :selection))))
(send select-tau :action
  #'(lambda (x) (send self :select-tau
                  (send select-tau :selection))))
```

```
    (if response
      (progn
        (setf ftp (* 0 ftp))
        (setf (select ftp new-ind) (repeat 1 (length new-ind)))
        (dotimes (i (length ftp))
          (setf (aref free-tau i p) (elt ftp i))
          (setf (aref free-tau p i) (elt ftp i)))
        (setf ftp (diagonal free-tau))
        (setf free-tau (matmult (diagonal ftp)
                                free-tau
                                (diagonal ftp)))
        (send self :free-tau free-tau)))))


(defmeth terrace-proto :select-random-effects ()
"Method Args: ()
Dialog method for select diagonal elements of tau for
estimation.  Off-diagonal elements are set to zero."
  (let* ((x-labels (send self :x-labels))
         (rand-ind (send self :rand-ind))
         (diag (repeat 0 (send self :p-reg)))
         (new-ind (choose-subset-dialog
                    "Choose simple random effects."
                    x-labels :initial rand-ind)))
    (if new-ind
      (progn
        (setf new-ind (car new-ind))
        (setf (select diag new-ind)
          (repeat 1 (length new-ind)))
        (send self :free-tau (diagonal diag))))))

(defmeth terrace-proto :model-dialog ()
"Method Args: ()
Main dialog method for estimating and altering terrace model."
  (let* ((lead-msg
           (send text-item-proto
             :new "Select Parameters for Multilevel Model"))
         (gamma-msg (send text-item-proto
                      :new "Linear Parameters"))
         (free-gamma (send self :free-gamma))
         (fgp (elt free-gamma p))
         (init-ind (which (= 1 fgp)))
         (gammap (elt (send self :gamma) p))
         (stderrp (elt (send self :gamma-stderr) p))
         (strings
           (mapcar #'(lambda (label g s)
             (if (> s 0)
               (format nil "~12a  t: ~8,4f" label (/ g s))
               (format nil "~12a  -  not estimated" label)))
             z-labels gammap stderrp))
         (caption (format nil
                    "Choose group variables to cross with ~a."
                    x-label))
         (new-ind (choose-subset-dialog caption strings
                          :initial init-ind)))
    (if new-ind
      (progn
        (setf new-ind (car new-ind))
        (setf fgp (* 0 fgp))
        (setf (select fgp new-ind) (repeat 1 (length new-ind)))
        (setf (elt free-gamma p) fgp)
        (send self :free-gamma free-gamma)))))


(defmeth terrace-proto :select-tau (p)
"Method Args: (p)
Dialog method for selecting parameters of the Pth row and
column of tau for estimation."
  (let* ((x-labels (send self :x-labels))
         (free-tau (send self :free-tau))
         (ftp (coerce (elt (column-list free-tau) p) 'list))
         (init-ind (which (= 1 ftp)))
         (caption (format nil
                    "Choose which covariance parameters ~
                     to estimate with ~a" (elt x-labels p)))
         (response (choose-subset-dialog caption x-labels
                         :initial init-ind))
         (new-ind (car response)))
      (list
        (format t "~%~%          Tau (correlation)~%~%")
        (dolist (i ind)
          (format t "~12a  " (elt x-labels i))
          (dolist (j ind)
            (format t "~8,4f " (aref cor i j)))
          (format t "~%"))))))

  #|
```

6.2 Dialog Methods

The easiest way to enteract with a TERRACE model is by way of dialog methods presented in this section. The user will find the method `:model-dialog` most helpful in that it will lead the user to all other dialog methods.

Code 6.2

```
|#

(defun group-mean (x group)
"Args: (data group)
Returns means of DATA indicated by GROUP. Both DATA and GROUP
should be lists of the same length."
  (let ((group-id (remove-duplicates group)))
    (mapcar
      #'(lambda (g)
          (mean (remove nil (if-else (= group g) x nil))))
                 group-id)))

(defmeth terrace-proto :select-gamma (p)
"Method Args: (p)
Dialog method for selecting variable to regress the Pth
level-one coefficient."
  (let* ((x-label (elt (send self :x-labels) p))
         (z-labels (send self :z-labels))
```

```
     (z-labels (send self :z-labels))
     (p-reg (send self :p-reg))
     (q-reg (send self :q-reg))
     (gamma (apply #'bind-rows (send self :gamma)))
     (free-gamma (apply #'bind-rows (send self :free-gamma)))
     (stderr (apply #'bind-rows (send self :gamma-stderr)))
     (tees (* 0 gamma))
     (cor (send self :tau-corr))
     (tau (send self :tau))
     (ind (send self :rand-ind))
     (free-tau (send self :free-tau))
     (flag (sum (- free-tau (diagonal (diagonal free-tau))))))
(dotimes (p p-reg)
  (dotimes (q q-reg)
    (unless (= (aref stderr p q) 0)
      (setf (aref tees p q)
        (/ (aref gamma p q)
           (aref stderr p q))))))
(format t
  "~%TERRACE-TWO:  Full Maximum Likelihood Estimates~%")
(format t
  "~%~%Parameters        Estimates     (S.E.)        T~%")
(dotimes (p p-reg)
  (format t "~%~a~%" (elt x-labels p))
  (dotimes (q q-reg)
    (if (= 1 (aref free-gamma p q))
      (format t "  By ~12a  ~8,4f   (~8,4f)   ~8,4f~%"
        (elt z-labels q)
        (aref gamma p q)
        (aref stderr p q)
        (aref tees p q)))))
(format t "~%~%Sigma^2:  ~8,4f~%" (send self :sigma))
(format t "~%          Tau (covariance) ~%~%")
(dolist (i ind)
  (format t "~12a  " (elt x-labels i))
  (dolist (j ind)
    (format t "~8,4f " (aref tau i j)))
  (format t "~%"))
(if (> flag 0)
      (format t "      Iteration ~3d: ~10,4f    ~a~%"
        count last method))
    (if history
      (let* ((dev1 (car history))
             (dev2 (car (send self :deviance-history)))
             (diff (- dev2 dev1))
             (df (- df2 df1))
             (p (unless (= df 0)
                  (- 1 (chisq-cdf (abs diff) (abs df))))))
        (if (and p (= n1 n2))
          (list
            (format t "~%Summary of Change in Deviance...~%~%")
            (format t "      Old Dev.:  ~10,4f   df: ~d~%"
              dev1 df1)
            (format t "      New Dev.:  ~10,4f   df: ~d~%"
              dev2 df2)
            (format t "        Change:  ~10,4f   df: ~d      "
              diff df)
            (if (> (* diff df) 0)
              (progn
                (format t "P: ~7,5f~%" p)
                (if (< df 0)
                  (format t "~%Beavis says, ~s~%"
                    (if (< p .05) "They rock."
                      (if (< p .15) "That was cool."
                        (if (< p .5) "Huh, eh eh eh."
                          "I hate variables that suck."))))
                  (format t "~%Butthead says, ~s~%"
                    (if (< p .05) "That really sucked."
                      (if (< p .15) "Huh, huh, huh."
                        (if (< p .5) "Cool."
                          "Yeah, good riddence."))))))
              (format t "~%"))))))
    (format t "~%")))

(defmeth terrace-proto :display ()
"Method Args: ()
Displays current estimates of parameters in a nice format."
  (let* ((x-labels (send self :x-labels))
|#

(defmeth terrace-proto :mlf
  (&key (precision 0.0001) (max-its 5))
"Method Args: (&key (precision 0.0001) (max-its 10))
Iterates Fisher steps until change in deviance is less than
PRECISION or the number of iterations has exceded MAX-ITS.
Prints results at each iteration."
  (format t "Maximizing Likelihood...~%~%")
  (format t "                       Deviance    Method~%~%")
  (let ((history (send self :deviance-history))
        (n1 (send self :n-cases :new nil))
        (n2 (send self :n-cases))
        (df1 (send self :df :new nil))
        (df2 (send self :df)))
    (do* ((resp (send self :step-em-mlf)
                (send self :step-fisher-mlf))
          (dev-hist resp (car resp))
          (count (length dev-hist) (1+ count))
          (last (car dev-hist) (car dev-hist))
          (method "EM, init." (cadr resp))
          (its 1 (1+ its))
          (max max-its
            (+ max
               (if (= its max)
                 (let ((reply (get-value-dialog
                      "How many more iterations do you wish?"
                                :initial max-its)))
                   (if reply (car reply) 0))
                 0)))
          (change (if (= 1 count) 1 (- (cadr dev-hist) last))
                  (- (cadr dev-hist) last)))
      ((or (< (abs change) precision)
           (= its max))
         (format t "Final Iteration ~3d: ~10,4f    ~a~%"
           count last method)
       (if (> change precision)
         (format t "~%   WARNING: Failed convergence criteria.~%")))
        (setf a (1+ a))))))

(defmeth unit-proto :info-mlf (&optional sig p)
"Method Args: ()
Return expected information matrix for sigma and tau per
unit under full maximum likelihood."
  (let* ((sig (if sig sig (send self :terrace :sigma)))
         (p (if p p (send self :terrace :p-reg)))
         (n (send self :n-cases))
         (xx (send self :xx))
         (c-inv (send self :c-inv))
         (xxc (matmult xx c-inv))
         (a11 (/ (+ n
                    (* -2 (sum (diagonal xxc)))
                    (sum (* (transpose xxc) xxc)))
                  2))
         (xmx (send self :xmx))
         (a22 (/ (dbl-vech xmx p) 4))
         (xmmx (send self :xmmx))
         (c (- 2 (identity-matrix p)))
         (a12 (/ (vech (* c xmmx) p) 2)))
    (/ (bind-rows (cons a11 a12)
                  (bind-columns a12 a22))
       (* sig sig))))
```

#|

6 Interative Methods

Here I will present methods which the user may use to interact with a terrace model.

6.1 Display Methods

Code 6.1

```
    (send self :update-units)
    (send self :dispersion-needs-update)
    (list
      (send self :add-to-history :deviance-mlf)
      meth)))

(defmeth unit-proto :score-mlf (&optional sig p)
"Method Args: ()
Returns score vector per unit under full maximum likelihood."
  (let* ((sig (if sig sig (send self :terrace :sigma)))
         (p (if p p (send self :terrace :p-reg)))
         (n (send self :n-cases))
         (xx (send self :xx))
         (c-inv (send self :c-inv))
         (dmmd (send self :dmmd))
         (d1 (- n (sum (* xx c-inv)) (/ dmmd sig)))
         (xmx (send self :xmx))
         (xmd (send self :xmd))
         (mat (- xmx (outer-product (/ xmd sig) xmd)))
         (c (- 2 (identity-matrix p))))
    (/ (cons d1 (vech (* c mat) p))
       (* -2 sig))))

(defun dbl-vech (q p)
  (let* ((d (/ (* p (1+ p)) 2))
         (m (make-array (list d d)))
         (a 0) (b 0)
         (ca 1) (cb 1))
    (dotimes (j p m)
      (dolist (i (iseq j (1- p)))
        (setf b 0)
        (setf ca (if (= i j) 1 2))
        (dotimes (k p)
          (dolist (l (iseq k (1- p)))
            (setf cb (if (= k l) 1 2))
            (setf (aref m a b)
              (* ca cb (+ (* (aref q i l) (aref q j k))
                          (* (aref q i k) (aref q j l)))))
            (setf b (1+ b))))
```

```
              (format t "~%~%")))
          (list method-string
                (send self :sigma new-sig)
                (send self :tau new-tau)))
      (if (> count 10) (error "Change your mind about tau"))
      (if (= count 0) (format t "~%Eigenvalues(tau): ~%"))
      (dolist (e (combine eignvals)) (format t " ~8,4f" e))
      (format t "~%"))))

;;;;;   Old half-step/EM routine   ;;;;;;;;;;
;
;    (if (and (> new-sig 0)
;              (sub-posdefp new-tau rand-ind))
;      (list
;       "Fisher"
;        (send self :sigma new-sig)
;        (send self :tau new-tau))
;      (do ((new-sig (/ (+ new-sig sig) 2)
;                    (/ (+ new-sig sig) 2))
;
;
;      (if (and (> (/ (+ sig new-sig) 2) 0)
;                (sub-posdefp (/ (+ tau new-tau) 2) rand-ind))
;        (list
;         "Fisher, half step"
;          (send self :sigma (/ (+ sig new-sig) 2))
;          (send self :tau (/ (+ tau new-tau) 2)))
;        (list
;          "EM"
;          (send self :sigma-em-mlf)
;          (send self :tau-em-mlf))))))
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmeth terrace-proto :step-fisher-mlf ()
"Method Args: ()
Complete one Fisher step under full maximum likelihood."
  (let ((meth (car (send self :sigma-tau-fisher-mlf))))
    (send self :gamma-weighted)
    (send self :sum-units :score-mlf sig p)))

(defmeth terrace-proto :info-mlf ()
"Method Args: ()
Return expected information matrix for sigma and tau under
full maximum likelihood."
  (let ((sig (send self :sigma))
        (p (send self :p-reg)))
    (send self :sum-units :info-mlf sig p)))

(defmeth terrace-proto :sigma-tau-fisher-mlf ()
"Method Args: ()
Updates sigma and tau by a fisher step, half-stepping as
neccessary, under full maximum likelihood."
  (let* ((score (send self :score-mlf))
         (info (send self :info-mlf))
         (tau-ind (send self :tau-ind))
         (ind (if tau-ind (cons 0 (1+ tau-ind)) (list 0)))
         (change (sub-solve info score ind))
         (sig (send self :sigma))
         (tau (send self :tau))
         (rand-ind (send self :rand-ind))
         (p (send self :p-reg)))
    (do* ((count 0 (1+ count))
          (new-sig (+ sig (elt change 0))
                   (/ (+ new-sig sig) 2))
          (new-tau (+ tau (anti-vech (rest change) p))
                   (if (< count 10) (/ (+ tau new-tau) 2)
                     tau))
          (eignvals
            (eigenvalues (select new-tau rand-ind rand-ind))
            (eigenvalues (select new-tau rand-ind rand-ind)))
          (method-string "Fisher"
                   (format nil "Fisher, ~d half-steps" count)))
         ((and (> new-sig 0) (> (min eignvals) 0))
          (if (> count 0)
            (list
              (dolist (e (combine eignvals))
                (format t " ~8,4f" e))
```

```
"Method Args: ()
Updates and returns covariance method for gamma estimate."
  (let ((disp (send self :dispersion :updata update))
        (sig (send self :sigma)))
    (* sig disp)))

(defmeth terrace-proto :gamma-stderr ()
"Method Args: ()
Returns estimated standard errors for gamma estimates."
  (let ((var (diagonal (send self :gamma-cov)))
        (q (send self :q-reg)))
    (split-list (sqrt var) q)))

(defun vech (m &optional p)
  (let ((p (if p p (array-dimension m 0)))
        (v nil))
    (dotimes (j p (reverse v))
      (dolist (i (iseq j (1- p)))
        (setf v (cons (aref m i j) v))))))

(defun anti-vech (v &optional p)
  (let* ((p (if p p
              (floor
                (/ (-
                (sqrt (+ (* 8 (length v)) 1)) 1) 2))))
         (m (make-array (list p p)))
         (k 0))
    (dotimes (j p m)
      (dolist (i (iseq j (1- p)))
        (setf (aref m i j) (elt v k))
        (unless (= i j)
          (setf (aref m j i) (elt v k)))
        (setf k (1+ k))))))

(defmeth terrace-proto :score-mlf ()
"Method Args: ()
Returns score vector under full maximum likelihood."
  (let ((sig (send self :sigma))
        (p (send self :p-reg)))
```

$$E\left[\frac{\partial^2\lambda}{\partial\tau_{ij}\partial\tau_{kl}}\right] = -\frac{2-\delta_{ij}}{2}u_j'X'V^{-1}X(\frac{2-\delta_{kl}}{2}) \tag{22}$$

$$(u_k u_l' + u_l u_k')X'V^{-1}Xu_i \tag{23}$$

$$= -\frac{(2-\delta_{ij})(2-\delta_{kl})\sigma^{-4}}{4}\{(u_i'X'MXu_k)(u_j'X'MXu_l) \tag{24}$$

$$+(u_i'X'MXu_l)u_j'X'MXu_k)\} \tag{25}$$

$$E\left[\frac{\partial^2\lambda}{\partial\sigma^2\partial\tau_{ij}}\right] = -\frac{2-\delta_{ij}}{2}u_j'X'V^{-2}Xu_i \tag{26}$$

$$= -\frac{2-\delta_{ij}}{2}\sigma^{-4}u_j'X'M^2Xu_i. \tag{27}$$

Also $E\left[\frac{\partial^2\lambda}{\partial\phi\partial\gamma}\right]$ is zero, and

$$E\left[\frac{\partial^2\lambda}{\partial\gamma\partial\gamma'}\right] = -W'X'V^{-1}XW = -\sigma^{-2}W'X'MXW. \tag{28}$$

Thus, the information matrix is block diagonal and we can update γ independent of σ^2 and τ.

Code 5.2

```
|#

(defmeth terrace-proto :gamma-weighted ()
"Method Args: ()
Updates gamma with standard weighted estimate."
  (let ((v (send self :sum-units :pre-post-w :xmx))
        (w (send self :sum-units :pre-w :xmy))
        (ind (send self :gamma-ind))
        (q (send self :q-reg)))
    (send self :gamma
      (split-list
        (sub-solve v w ind) q))))

(defmeth terrace-proto :gamma-cov (&key (update t))
```

Since $V = X\tau X' + \sigma^2 I$ and τ is symmetric,

$$\frac{\partial V}{\partial(\sigma^2)} = I$$

and

$$\frac{\partial V}{\partial \tau_{ij}} = \begin{cases} X u_i u_i' X' & if\ i = j \\ X u_i u_j' X' + X u_j u_i' X' & if\ i \neq j \end{cases},$$

where u_i is a column of the identity matrix of dimension P.

After some algebra, one obtains

$$\begin{aligned} \frac{\partial \lambda}{\partial(\sigma^2)} &= -\frac{1}{2}\{tr(V^{-1}) - d'V^{-2}d\} && (13) \\ &= -\frac{1}{2}\{n\sigma^{-2} - \sigma^{-2} tr(X'XC^{-1}) - \sigma^{-4} d'M^2 d\} && (14) \end{aligned}$$

and

$$\begin{aligned} \frac{\partial \lambda}{\partial \tau_{ij}} &= -\frac{1}{2}\{tr(V^{-1}Xu_iu_j'X') + (1-\delta_{ij})tr(V^{-1}Xu_ju_i'X') && (15) \\ &\quad -d'V^{-1}Xu_iu_j'X'V^{-1}d - (1-\delta_{ij})d'V^{-1}Xu_ju_i'X'V^{-1}d\} && (16) \\ &= -\frac{2-\delta_{ij}}{2}\{u_j'X'V^{-1}Xu_i - (u_i'X'V^{-1}d)(u_j'X'V^{-1}d)\} && (17) \\ &= -\frac{2-\delta_{ij}}{2}\{\sigma^{-2}u_j'X'MXu_i - \sigma^{-4}(u_i'X'Md)(u_j'X'Md)\} && (18) \end{aligned}$$

Setting $\frac{\partial\lambda}{\partial\tau} = (\frac{\partial\lambda}{\partial\tau_{ij}})$, we may write equation 18 as

$$\frac{\partial \lambda}{\partial \tau} = -\sigma^{-4}(J - \frac{1}{2}I) * (\sigma^2 X'MX - X'Md(X'Md)'),$$

where J is a matrix of ones and $*$ indicates element-wise multiplication. Also for γ, we have

$$\frac{\partial \lambda}{\partial \gamma} = W'X'V^{-1}d = \sigma^{-2}W'X'Md. \qquad (19)$$

Now we simplify the expected second derivatives.

$$\begin{aligned} E\left[\frac{\partial^2\lambda}{(\partial(\sigma^2))^2}\right] &= -\frac{1}{2}tr(V^{-2}) && (20) \\ &= -\frac{1}{2}\sigma^{-4}\{n - 2tr(X'XC^{-1}) + tr(X'XC^{-1}X'XC^{-1})\} && (21) \end{aligned}$$

#|

5.2 Fisher scoring

Implementation of the Fisher scoring algorithm involves computation of the first derivative and expectation of the second derivative of the log-likelihood in equation 6. These will be referred to as the score vector and information matrix. Longford (1987) and Bryk and Raudenbush (1992) provide these derivatives.

Let's simplify the log-likelihood a bit. Let

$$\lambda = -\frac{1}{2}\{\log|V| + d'V^{-1}d\},$$

where $V = X\tau X' + \sigma^2 I$ and $d = Y - XW\gamma$. Suppose that ϕ and ϕ' are arbitrary elements of (σ^2, τ). Then

$$\frac{\partial(\log|V|)}{\partial\phi} = tr\left(V^{-1}\frac{\partial V}{\partial\phi}\right), \tag{8}$$

and

$$\frac{\partial(V^{-1})}{\partial\phi} = -V^{-1}\frac{\partial V}{\partial\phi}V^{-1}. \tag{9}$$

Thus, we have first and second derivatives

$$\frac{\partial\lambda}{\partial\phi} = -\frac{1}{2}\left\{tr\left(V^{-1}\frac{\partial V}{\partial\phi}\right) - d'V^{-1}\frac{\partial V}{\partial\phi}V^{-1}d\right\} \tag{10}$$

and

$$\frac{\partial^2\lambda}{\partial\phi\partial\phi'} = -\frac{1}{2}\left\{-tr\left(V^{-1}\frac{\partial V}{\partial\phi'}V^{-1}\frac{\partial V}{\partial\phi}\right) + 2d'V^{-1}\frac{\partial V}{\partial\phi'}V^{-1}\frac{\partial V}{\partial\phi}V^{-1}d\right\}. \tag{11}$$

Now taking the expectation of equation 11, we get

$$E\left(\frac{\partial^2\lambda}{\partial\phi\partial\phi'}\right) = -\frac{1}{2}tr\left(V^{-1}\frac{\partial V}{\partial\phi'}V^{-1}\frac{\partial V}{\partial\phi}\right). \tag{12}$$

```
(defmeth unit-proto :pre-gamma-em-mlf ()
  (let ((z (send self :z))
        (xy (send self :xy))
        (xx (send self :xx))
        (u (send self :u-hat)))
    (if u (pre-w z (- xy (matmult xx u)))
          (pre-w z xy))))

(defmeth unit-proto :trace-xxc ()
  (sum (* (send self :xx) (send self :c-inv))))

(defmeth unit-proto :uu ()
  (let ((u (send self :u-hat)))
    (outer-product u u)))

(defmeth terrace-proto :deviance-mlf ()
  (let ((rand-ind (send self :rand-ind))
        (sig (send self :sigma))
        (det-tau (send self :det-tau)))
    (send self :sum-units
      :deviance-mlf rand-ind sig det-tau)))

(defmeth unit-proto :deviance-mlf
  (&optional rand-ind sig det-tau)
  (let ((n (send self :n-cases))
        (rand-ind (if rand-ind rand-ind
                    (send self :terrace :rand-ind)))
        (sig (if sig sig (send self :terrace :sigma)))
        (det-tau (if det-tau det-tau
                   (send self :terrace :det-tau)))
        (c-inv (send self :c-inv))
        (dmd (send self :dmd)))
    (+ (* n normal-const)
       (* (- n (length rand-ind)) (log sig))
       (log det-tau)
       (- (log (sub-determinant c-inv rand-ind)))
       (/ dmd sig))))

(defconstant normal-const (log (* 2 pi)))
```

```
         (ind (send self :rand-ind))
         (eigen (eigenvalues (select new-tau ind ind))))
    (if (> (min eigen) 0)
      (send self :tau new-tau)
      (send self :tau (diagonal (diagonal new-tau))))))

(defmeth terrace-proto :deviance ()
  (let ((ind (send self :rand-ind))
        (sig (send self :sigma))
        (det-tau (send self :det-tau)))
    (send self :sum-units :deviance ind sig det-tau)))

(defmeth terrace-proto :det-tau ()
  (let ((ind (send self :rand-ind))
        (tau (send self :tau)))
    (sub-determinant tau ind)))

(defmeth terrace-proto :update-units ()
  (let ((sig-tau-ind
          (list
            (* (send self :sigma)
               (send self :tau-inv))
            (send self :rand-ind))))
    (send self :map-units :c-inv sig-tau-ind)
    (send self :map-units :u-hat t)))

(defmeth terrace-proto :step-em-mlf ()
  (send self :gamma-em-mlf)
  (send self :sigma-em-mlf)
  (send self :tau-em-mlf)
  (send self :update-units)
  (send self :dispersion-needs-update)
  (send self :add-to-history :deviance-mlf))

(defmeth terrace-proto :add-to-history (d)
  (let ((hist (send self :deviance-history))
        (dev (send self d)))
    (send self :deviance-history (cons dev hist))))
```

to a unit or terrace. Unit deviance is

$$D_j = n\log(2\pi) + (n-r)\log(\sigma^2) + \log|\tau| - \log|C^{-1}| + \sigma^{-2}d'Md,$$

since

$$|V| = (\sigma^2)^{(n-r)}|\tau|/|C^{-1}|,$$

where r is the rank of τ and determinants are taken over the relevant submatrix. Next, `:update-units` causes the slots `c-inv` and `u-hat` to be updated. Method `:step-em-mlf` completes one full iteration of the EM algorithm.

Code 5.1

```
|#

(defmeth terrace-proto :gamma-em-mlf ()
  (let ((v (send self :sum-units :pre-post-w :xx))
        (w (send self :sum-units :pre-gamma-em-mlf))
        (ind (send self :gamma-ind))
        (q (send self :q-reg)))
    (send self :gamma
      (split-list
        (sub-solve v w ind)
        q))))

(defmeth terrace-proto :sigma-em-mlf ()
  (let ((n (send self :n-cases))
        (dmmd (send self :sum-units :dmmd))
        (sig (send self :sigma))
        (trace (send self :sum-units :trace-xxc)))
    (send self :sigma (/ (+ dmmd (* sig trace)) n))))

(defmeth terrace-proto :tau-em-mlf ()
  (let* ((j (send self :j-units))
         (uu (send self :sum-units :uu))
         (sig (send self :sigma))
         (c (send self :sum-units :c-inv))
         (free (send self :free-tau))
         (new-tau (* (/ (+ uu (* sig c)) j) free))
```

```
          (yy (send self :yy)))
      (send self :n-cases (1+ n))
      (send self :case-labels (append labels (list label)))
      (send self :x (if x-mat (bind-rows x-mat x) x))
      (send self :y (append y-list (list y)))
      (send self :xx (+ (outer-product x x) xx))
      (send self :xy (+ (* y x) xy))
      (send self :yy (+ (* y y) yy)))))

#|
```

5 ML Estimation

TERRACE-TWO provides two approaches to computing the full information ML estimates: EM algorithm and Fisher scoring. While the latter approach is faster than the former, the EM algorithm is quite stable. We may use the EM algorithm to obtain good initial estimates for the Fisher scoring algorithm. And should an iteration of Fisher scoring produce estimates outside of the parameter space, the iteration can be replaced with an EM step. First, I discuss the EM algorithm.

5.1 EM algorithm

The EM algorithm leads to update γ, σ^2 and τ with

$$\hat{\gamma} = (\sum W'X'XW)^{-1} \sum W'(X'Y - X'Xu),$$

$$\hat{\sigma}^2 = \frac{1}{N} \sum (d'M^2 d + \sigma^2 tr[X'XC^{-1}]),$$

and

$$\hat{\tau} = \frac{1}{J} \sum (uu' + \sigma^2 C^{-1}).$$

We also need some control methods. The message `:deviance-mlf` will compute unit or complete deviance depending on whether the message is sent

```
(defmeth unit-proto :include-case (case-data &optional p)
  (if (let ((data-id (elt case-data 0))
            (unit-id (send self :unit-id)))
        (or (eql data-id unit-id)
            (= data-id unit-id)))
    (let* ((p (if p p (1- (length case-data))))
           (y (elt case-data p))
           (x (concatenate 'vector
                '(1)
                (select case-data (iseq 1 (1- p)))))
           (n (send self :n-cases))
           (xx (send self :xx))
           (xy (send self :xy))
           (yy (send self :yy)))
      (send self :n-cases (1+ n))
      (send self :xx (+ (outer-product x x) xx))
      (send self :xy (+ (* y x) xy))
      (send self :yy (+ (* y y) yy)))))

(defmeth reg-unit-proto :include-case (case-data &optional p)
  (if (let ((data-id (elt case-data 0))
            (unit-id (send self :unit-id)))
        (or (eql data-id unit-id)
            (= data-id unit-id)))
    (let* ((p (if p p (1- (length case-data))))
           (y (elt case-data p))
           (x (concatenate 'vector
                '(1)
                (select case-data (iseq 1 (1- p)))))
           (n (send self :n-cases))
           (id (send self :unit-id))
           (label (if (integerp id)
                      (format nil "~d-~d" id n)
                      (format nil "~a-~d" id n)))
           (labels (send self :case-labels))
           (x-mat (send self :x))
           (y-list (send self :y))
           (xx (send self :xx))
           (xy (send self :xy))
;    (flet ((include-case (row)
;             (dolist (unit units)
;               (if (send unit :include-case row p)
;                   (return)))))
;      (mapcar #'include-case
;        (row-list case-data)))))

(defmeth terrace-proto :include-cases (case-data)
  (let ((p (send self :p-reg))
        (units (send self :units))
        (unit-ids (send self :map-units :unit-id)))
    (flet ((include-case (row)
             (let ((ind (car (which
                                (= (elt row 0) unit-ids)))))
               (if ind (send (elt units ind)
                             :include-case row p)))))
      (mapcar #'include-case
        (row-list case-data)))))

(defmeth terrace-proto :remove-empty-units ()
  (let* ((units (send self :units))
         (n (map-elements #'send units :n-cases))
         (ind (which (> n 0)))
         (change (- (length n) (length ind))))
    (if (> change 0)
      (progn
        (format t "Removing ~d empty unit(s).~%" change)
        (send self :units (select units (which (> n 0))))))))

(defmeth unit-proto :isnew (terrace unit-data)
  (setf (slot-value 'terrace) terrace)
  (setf (slot-value 'unit-id) (elt unit-data 0))
  (setf (elt unit-data 0) 1)
  (send self :z unit-data)
  (send self :n-cases 0)
  (send self :xx 0)
  (send self :xy 0)
  (send self :yy 0))
```

```
      (send ter :z-labels z-labels))
    (send ter :create-units unit-data)
    (send ter :include-cases case-data)
    (send ter :remove-empty-units)
    (send ter :free-gamma
      ;(mapcar #'(lambda (q) (repeat 1 q)) (repeat q p))
      ;(mapcar #'(lambda (q) (repeat '(1 0) (list 1 (1- q)))))
      (cons (repeat '(1 0) (list 1 (1- q)))
            (mapcar #'(lambda (q) (repeat 0 q))
              (repeat q (- p 1)))))
    (send ter :gamma-em-mlf)
    (send ter :sigma 1)
    (send ter :tau (identity-matrix p))
    (send ter :free-tau
      ;(make-array (list p p) :initial-element 1)
      ;(identity-matrix p)
      (diagonal (repeat '(1 0) (list 1 (- p 1))))
      )
    (send ter :model-dialog)
    ter))

(defmeth terrace-proto :create-units
  (unit-data-matrix &optional (proto reg-unit-proto))
  (let ((unit-ids
          (coerce (car (column-list unit-data-matrix))
            'list)))
    (unless (= (length unit-ids)
               (length (remove-duplicates unit-ids)))
      (error "TERRACE ERROR: unit-id's not unique")))
  (let ((units (send self :units))
        (new-units
          (mapcar #'(lambda (row)
                      (send proto :new self row))
            (row-list unit-data-matrix))))
    (send self :units (append units new-units))))

;(defmeth terrace-proto :include-cases (case-data)
;  (let ((p (send self :p-reg))
;        (units (send self :units)))
```

4 Initialization Methods

The function `make-terrace` takes in two matrices, *unit-data* and *case-data*, and returns a terrace object including the creation of related units. The first column of *unit-data* must contain a unique identification numbers or strings for each unit. The other columns of *unit-data* may be the unit background carriers. Do not include an intercept carrier as this will be created automatically. Likewise the first column of *case-data* must contain unit identifications for each case (row). The next columns may contain case background carriers. Again do not include an intercept carrier. And the last column of *case-data* must carry the response variable.

Code 4.1

```
|#

(defun make-terrace
  (unit-data case-data &key x-labels z-labels)
  (let* ((ter (send terrace-proto :new))
         (p (1- (array-dimension case-data 1)))
         (q (array-dimension unit-data 1))
         (x-labels-d
           (cons "Intercept"
             (mapcar
               #'(lambda (int)
                   (format nil "Case-var-~d" int))
               (iseq 1 (- p 1)))))
         (z-labels-d
           (cons "Intercept"
             (mapcar
               #'(lambda (int)
                   (format nil "Unit-var-~d" int))
               (iseq 1 (1- q))))))
    (send ter :slot-value 'x-labels x-labels-d)
    (send ter :y-label "Response-var")
    (send ter :slot-value 'z-labels z-labels-d)
    (if x-labels
      (send ter :x-labels x-labels))
    (if z-labels
    (sub-inverse (+ (diagonal diag) tau) ind)))

(defmeth terrace-proto :tau-eigenvalues ()
  (let ((tau (send self :tau))
        (ind (send self :rand-ind)))
    (eigenvalues (select tau ind ind))))

(defun cov-to-corr (a)
  (let ((n (array-dimension a 0))
        (d (sqrt (diagonal a)))
        (b (make-array (array-dimensions a))))
    (dotimes (i n b)
      (dotimes (j n)
        (setf (aref b i j) (/ (aref a i j)
                              (nth i d) (nth j d)))))))

(defmeth terrace-proto :tau-corr ()
  (let* ((tau (send self :tau))
         (diag (if-else (= (diagonal tau) 0) 1 0)))
    (cov-to-corr (+ (diagonal diag) tau))))

(defmeth terrace-proto :deviance-history
  (&optional (new nil set))
  (if set (setf (slot-value 'deviance-history) new)
    (slot-value 'deviance-history)))

(defmeth terrace-proto :df (&key (new t))
  (if new
    (setf (slot-value 'df)
      (- (send self :n-cases)
         (sum (send self :free-gamma))
         1
         (sum (vech (send self :free-tau)))))
    (slot-value 'df)))

#|
(defmeth terrace-proto :p-active ()
  (sum (if-else
        (> (mapcar #'sum (send self :free-gamma)) 0)
        1 0)))

(defmeth terrace-proto :sigma (&optional (new nil set))
  (if set (setf (slot-value 'sigma) new)
    (slot-value 'sigma)))

(defmeth terrace-proto :tau (&optional (new nil set))
  (if (and set (matrixp new))
    (let ((p (send self :p-reg)))
      (if (and (= p (array-dimension new 0))
               (= p (array-dimension new 1)))
        (setf (slot-value 'tau) new)))
    (slot-value 'tau)))

(defmeth terrace-proto :free-tau (&optional (new nil set))
  (if (and set (matrixp new))
    (let ((p (send self :p-reg)))
      (if (and (= p (array-dimension new 0))
               (= p (array-dimension new 1))
               (= 0 (sum (^ (- new (* new new)) 2))))
          (setf (slot-value 'free-tau) new))
      (send self :update-units)))
  (slot-value 'free-tau))

(defmeth terrace-proto :rand-ind ()
  (which (= 1 (diagonal (send self :free-tau)))))

(defmeth terrace-proto :tau-ind ()
  (let ((tau-list (vech (send self :free-tau))))
    (which (= 1 tau-list))))

(defmeth terrace-proto :tau-inv ()
  (let* ((tau (send self :tau))
         (diag (if-else (= (diagonal tau) 0) 1 0))
         (ind (send self :rand-ind)))
  (length (send self :z-labels)))

(defmeth terrace-proto :gamma (&optional (new nil set))
  (if set
    (let ((p (send self :p-reg))
          (q (send self :q-reg)))
      (if (and (= p (length new))
               (= q (length (car new))))
        (setf (slot-value 'gamma) new)
        (error "TERRACE ERROR: bad gamma dimensions")))
    (slot-value 'gamma)))

(defmeth terrace-proto :dispersion (&key update)
  (if update
    (let ((mat (send self :sum-units :pre-post-w :xmx))
          (ind (send self :gamma-ind)))
      (setf (slot-value 'dispersion) (sub-inverse mat ind)))
    (if (slot-value 'dispersion) (slot-value 'dispersion)
      (send self :dispersion :update t))))

(defmeth terrace-proto :dispersion-needs-update ()
  (setf (slot-value 'dispersion) nil))

(defmeth terrace-proto :free-gamma (&optional (new nil set))
  (if set
    (let ((p (send self :p-reg))
          (q (send self :q-reg)))
      (if (and (= p (length new))
               (= q (length (car new)))
               (= 0 (sum (^ (- new (* new new)) 2))))
        (setf (slot-value 'free-gamma) new)))
    (slot-value 'free-gamma)))

(defmeth terrace-proto :gamma-ind ()
  (which (= 1 (combine (send self :free-gamma)))))

(defmeth terrace-proto :p-ind ()
  (which (mapcar #'(lambda (x) (> (sum x) 0))
           (send self :free-gamma))))
  (flet ((send-message (unit) (apply #'send unit args)))
    (let* ((j 1)
           (units (send self :units))
           (c (send-message (car units)))
           (sum-diff 0))
      (dolist (unit (rest units) (+ (* j c) sum-diff))
        (setf j (1+ j))
        (setf sum-diff (+ sum-diff
                          (- (send-message unit) c)))))))

(defmeth terrace-proto :n-cases (&key (new t))
  (if new
    (setf (slot-value 'n-cases)
      (send self :sum-units :n-cases))
    (slot-value 'n-cases)))

(defmeth terrace-proto :j-units ()
  (length (send self :units)))

(defmeth terrace-proto :x-labels (&optional (new nil set))
  (if (and set (= (length new)
                  (length (slot-value 'x-labels))))
    (setf (slot-value 'x-labels) new)
    (slot-value 'x-labels)))

(defmeth terrace-proto :y-label (&optional (new nil set))
  (if set (setf (slot-value 'y-label) new)
    (slot-value 'y-label)))

(defmeth terrace-proto :z-labels (&optional (new nil set))
  (if (and set (= (length new)
                  (length (slot-value 'z-labels))))
    (setf (slot-value 'z-labels) new)
    (slot-value 'z-labels)))

(defmeth terrace-proto :p-reg ()
  (length (send self :x-labels)))

(defmeth terrace-proto :q-reg ()
```

free-gamma will contain a list of lists of 0's and 1's in the dimensions of gamma. A 1 will indicate a free parameter, and a 0 will indicate a fixed parameter. In a similar fashion, free-tau indicates which elements of τ are free parameters.

The slot deviance-history will contain a record of deviance ($D = -2L$) at each iteration in reverse chronological order.

Code 3.4

```
|#

(defproto terrace-proto
  '(units            ; list of objects for related units
    x-labels         ; labels for the level-one variables
    y-label          ; label for the response variable
    z-labels         ; labels for the level-two variables
    gamma            ; list of gamma_p estimates
    free-gamma       ; indicators for free gamma elements
    dispersion       ; dispersion of gamma estimate
    sigma            ; sigma^2 estimate
    tau              ; tau estimate
    free-tau         ; indicators of free tau elements
    deviance-history ; record of -2L
    n-cases          ; number of cases
    df               ; degrees of freedom
))

(defmeth terrace-proto :units (&optional (new nil set))
  (if set
    (list (setf (slot-value 'units) new)
          (send self :map-units :slot-value 'terrace self)))
  (slot-value 'units))

(defmeth terrace-proto :map-units (&rest args)
  (flet ((send-message (unit) (apply #'send unit args)))
    (mapcar #'send-message (slot-value 'units))))

(defmeth terrace-proto :sum-units (&rest args)
  (let ((res (send self :residuals :ols-beta))
        (lev (send self :ols-leverages))
        (sig (send self :ols-sigma)))
    (/ res (sqrt (* sig (pmax 0.000001 (- 1 lev)))))))

(defmeth reg-unit-proto :ols-externally-studentized-residuals ()
"Method Args: ()
Computes ordinary least squares externally studentized
residuals."
  (let ((res (send self :ols-studentized-residuals))
        (df (- (send self :n-cases)
               (send self :terrace :p-active))))
    (* res (sqrt (/ (- df 1)
                    (pmax 0.01 (- df (* res res))))))))

(defmeth reg-unit-proto :ols-cooks-distance ()
"Method Args: ()
Computes ordinary least squares Cook's distances."
  (let ((res (send self :residuals :ols-beta))
        (p (send self :terrace :p-active))
        (lev (send self :ols-leverages))
        (sig (send self :ols-sigma)))
    (* (^ (/ res (- 1 lev)) 2)
       (/ lev p sig))))

#|
```

3.3 Terrace-proto

Now I define `terrace-proto`. The slot `unit` will contain a list of objects representing the units in the model. Inserting or deleting an object is equivalent to including or deleting a unit from the model. The method `:map-units` passes messages to each unit an returns a list containing the responses of each unit. Similarly, the method `:sum-units` passes messages on to each unit and returns the sum of responses.

The slots `gamma`, `sigma`, and `tau` will contain representations of γ, σ^2, and τ. Actually, `gamma` will be a list of lists each representing γ_p. The slot

```
"Method Args: (&optional (beta-msg :post-beta))
Computes fitted values, Y-hat =  Xb, where b is estimated
according to BETA-MSG which should be :ols-beta, :prior-beta,
:post-beta, or a list for an arbitrary beta."
  (let ((x (send self :x))
        (b (if (listp beta-msg)
             beta-msg
             (send self beta-msg))))
    (matmult x b)))

(defmeth reg-unit-proto :residuals
  (&optional (beta-msg :post-beta))
"Method Args: (&optional (beta-msg :post-beta))
Computes residuals, Y - Xb, where b is estimated according
to BETA-MSG which should be :ols-beta, :prior-beta, or
:post-beta, or a list for an arbitrary beta."
  (let ((y (send self :y))
        (x (send self :x))
        (b (if (listp beta-msg)
             beta-msg
             (send self beta-msg))))
    (- y (matmult x b))))

(defmeth reg-unit-proto :ols-leverages ()
"Method Args: ()
Computes ordinary least squares leverage values, diagonal of
X(X'X)^{-1}X'."
  (let* ((xx (send self :xx))
         (ind (send self :terrace :p-ind))
         (xx-inv (sub-inverse xx ind))
         (x (send self :x)))
    (mapcar
      #'(lambda (r) (matmult r xx-inv r))
      (row-list x))))

(defmeth reg-unit-proto :ols-studentized-residuals ()
"Method Args: ()
Computes ordinary least squares internally studentized
residuals."
```

```
  (&optional (new nil set))
"Method Args: (&optional new)
Returns content of slot CASE-LABELS. Optionally,
sets the slot with NEW if supplied."
  (if set (setf (slot-value 'case-labels) new)
    (slot-value 'case-labels)))

(defmeth reg-unit-proto :update-xx ()
"Method Args: ()
Updates the content of slot xx with X'WX, where W is the
diagonal matrix of case weights."
  (let ((x (send self :x))
        (w (send self :w)))
     (send self :xx
       (if w (matmult (transpose x) (diagonal w) x)
             (matmult (transpose x) x)))))

(defmeth reg-unit-proto :update-xy ()
"Method Args: ()
Updates the content of slot xy with X'WY, where W is the
diagonal matrix of case weights."
  (let ((x (send self :x))
        (y (send self :y))
        (w (send self :w)))
     (send self :xy
       (if w (matmult (transpose x) (* w y))
             (matmult (transpose x) y)))))

(defmeth reg-unit-proto :update-yy ()
"Method Args: ()
Updates the content of slot yy with Y'WY, where W is the
diagonal matrix of case weights."
  (let ((y (send self :y))
        (w (send self :w)))
     (send self :yy
       (if w (sum (* y w y)) (sum (* y y))))))

(defmeth reg-unit-proto :y-hat
  (&optional (beta-msg :post-beta))
```

```
    (slot-value 'x)))

(defmeth reg-unit-proto :v ()
"Method Args: ()
Returns the variace matrix of Y given gamma, sigma, and tau."
  (let ((n (send self :n-cases))
        (sig (send self :terrace :sigma))
        (x (send self :x))
        (tau (send self :terrace :tau)))
    (+ (* sigma (identity-matrix n))
       (matmult x tau (transpose x)))))

(defmeth reg-unit-proto :v-inv ()
"Method Args: ()
Returns the inverse of the variance matrix of Y given gamma,
sigma, and tau."
  (let ((n (send self :n-cases))
        (x (send self :x))
        (c-inv (send self :c-inv))
        (sigma (send self :terrace :sigma)))
    (/ (- (identity-matrix n)
          (matmult x c-inv (transpose x)))
       sigma)))

(defmeth reg-unit-proto :y (&optional (new nil set))
"Method Args: (&optional new-y)
Returns content of slot Y, case level response variable.
Optionally, sets the slot with NEW-Y if supplied."
  (if set (setf (slot-value 'y) new)
    (slot-value 'y)))

(defmeth reg-unit-proto :weights (&optional (new nil set))
"Method Args: (&optional new)
Returns content of slot WEIGHTS, case weights. Optionally,
sets the slot with NEW if supplied."
  (if set (setf (slot-value 'weights) new)
    (slot-value 'weights)))

(defmeth reg-unit-proto :case-labels
```

posterior estimate $\hat{\beta} = \check{\beta} + \hat{u}$

ols estimate $\tilde{\beta} = (X'X)^{-1}X'Y$,

which correspond to three types of residuals

prior residual $\check{r} = Y - X\check{\beta}$

posterior residual $\hat{r} = Y - X\hat{\beta}$

ols residual $\tilde{r} = Y - X\tilde{\beta}$.

Code 3.3

```
|#

;;;;
;;;;     Diagnosable Regression Unit Prototype
;;;;

(defproto reg-unit-proto
 '(x                   ; case background matrix X
   y                   ; response vector Y
   weights             ; list of case weights
   case-labels         ; list of names for each case
  )
 ()
 unit-proto
 "Diagnosable Regression Unit Model for TERRACE-TWO")

;;;
;;;    Basic and Slot Accessor Methods
;;;

(defmeth reg-unit-proto :x (&optional (new nil set))
"Method Args: (&optional new-x)
Returns content of slot X, case level background variables.
Optionally, sets the slot with NEW-X if supplied."
  (if set (setf (slot-value 'x) new)
Returns XMMd, where d = Y - XWgamma and M = sig^2 V^-1.
This is equivalent to X'Mr where r = Y - XWgamma - Xu."
  (let* ((xd (send self :xd))
         (xx (send self :xx))
         (u-hat (send self :u-hat))
         (c-inv (send self :c-inv))
         (xxu (matmult xx u-hat)))
    (+ xd (* -2 xxu) (matmult xx (matmult c-inv xxu)))))

(defmeth unit-proto :dmmd ()
"Message Args: ()
Returns d'MMd, where d = Y - XWgamma and M = sig^2 V^-1.
This is equivalent to r'r where r = Y - XWgamma - Xu."
  (let* ((yy (send self :yy))
         (xx (send self :xx))
         (xy (send self :xy))
         (xd (send self :xd))
         (b (send self :prior-beta))
         (cxd (send self :u-hat)))
    (+ yy
       (- (matmult b (+ xy xd)))
       (* -2 (matmult xd cxd))
       (matmult cxd xx cxd))))

#|
```

3.2 Reg-unit-proto

Before going on to define `terrace-proto`, I will define an extension of `unit-proto`. The purpose of `reg-unit-proto` is to define a unit object which retains case level information. That is we will include slots for the matrices X and Y. While this is not neccessary for for the purpose of estimating model parameters, it is neccessary for computing case level diagnostics, such as various residuals and measures of influence.

There are three basic ways of estimating β

prior estimate $\check{\beta} = W\hat{\gamma}$

```
"Message Args: ()
Returns the estimated covariance of ols-beta."
  (let ((xx (send self :xx))
        (ind (send self :terrace :p-ind))
        (sig (send self :ols-sigma)))
    (* sig (sub-inverse xx ind))))

(defmeth unit-proto :xd ()
"Message Args: ()
Returns X'd, where d = Y - XWgamma."
  (let ((xx (send self :xx))
        (xy (send self :xy))
        (b (send self :prior-beta)))
    (- xy (matmult xx b))))

(defmeth unit-proto :xmd ()
"Message Args: ()
Returns X'Md, where d = Y - XWgamma and M = sig^2 V^-1.
This is equivalent to X'r where r = Y - XWgamma - Xu."
  (let* ((xx (send self :xx))
         (xd (send self :xd))
         (c-inv (send self :c-inv)))
    (- xd (matmult xx (matmult c-inv xd)))))

(defmeth unit-proto :dmd ()
"Message Args: ()
Returns d'Md, where d = Y - XWgamma and M = sig^2 V^-1.
This is equivalent to d'r where r = Y - XWgamma - Xu."
  (let ((yy (send self :yy))
        (xy (send self :xy))
        (xd (send self :xd))
        (b (send self :prior-beta))
        (u-hat (send self :u-hat)))
    (- yy
       (matmult b (+ xy xd))
       (matmult xd u-hat))))

(defmeth unit-proto :xmmd ()
"Message Args: ()
```

```
:x, :xmx, etc."
  (let ((stat (send self stat-msg))
        (z (send self :z)))
    (pre-w z (transpose (pre-w z stat)))))

(defmeth unit-proto :prior-beta ()
"Message Args:()
Computes Wgamma, which is the prior prediction of beta."
  (let* ((z (send self :z))
         (gamma (send self :terrace :gamma)))
    (mapcar #'(lambda (g) (sum (* z g))) gamma)))

(defmeth unit-proto :post-beta ()
"Message Args: ()
Return the current empirical Bayes estimate of beta."
  (let ((prior (send self :prior-beta))
        (u-hat (send self :u-hat)))
    (+ prior u-hat)))

(defmeth unit-proto :ols-beta ()
"Message Args: ()
Returns the ols estimate of beta, b = (X'X)^{-1}X'Y."
  (let ((xx (send self :xx))
        (xy (send self :xy))
        (ind (send self :terrace :p-ind)))
    (sub-solve xx xy ind)))

(defmeth unit-proto :ols-sigma ()
"Message Args: ()
Returns the ols estimate of sigma^2."
  (let ((yy (send self :yy))
        (xy (send self :xy))
        (b (send self :ols-beta))
        (n (send self :n-cases))
        (p (length (send self :terrace :p-ind))))
    (/ (- yy (sum (* b xy)))
       (- n p))))

(defmeth unit-proto :ols-beta-cov ()
Returns X'MY where M = sig^2 V^-1."
  (let ((xx (send self :xx))
        (xy (send self :xy))
        (c-inv (send self :c-inv)))
    (- xy (matmult xx (matmult c-inv xy)))))

(defmeth unit-proto :xmmx ()
"Message Args:()
Returns X'M^2X where M = sig^2 V^-1."
  (let* ((xx (send self :xx))
         (c-inv (send self :c-inv))
         (cxx (matmult c-inv xx))
         (xxcxx (matmult xx cxx)))
    (+ xx (* -2 xxcxx) (matmult xxcxx cxx))))

(defun pre-w (z mat)
"Args: (z matrix)
Computes outer-product of Z with each row of MATRIX and binds results
as a single matrix."
  (if (matrixp mat)
    (apply #'bind-rows
      (mapcar #'(lambda (r) (outer-product z r))
                   (row-list mat)))
    (combine
      (mapcar #'(lambda (r) (* z r))
        (coerce mat 'list)))))

(defmeth unit-proto :pre-w (stat-msg)
"Message Args: (stat-msg)
Computes W'S where S is a statistic returned by sending the message
STAT-MSG to the unit. STAT-MSG may be :xx, :xy, :xmx, etc."
  (let ((stat (send self stat-msg))
        (z (send self :z)))
    (pre-w z stat)))

(defmeth unit-proto :pre-post-w (stat-msg)
"Message Args: (stat-msg)
Computes W'SW where S is a statistic returned by sending
the message STAT-MSG to the unit. STAT-MSG may be :xx,
```

Then

$$\begin{aligned}
X'MX &= X'X - X'XC^{-1}X'X \\
X'MY &= X'Y - X'XC^{-1}X'Y \\
X'M^2X &= X'X - 2X'XC^{-1}X'X + X'XC^{-1}X'XC^{-1}X'X
\end{aligned}$$

We also want expressions involving d which involves the prior β, denoted $\check{\beta}$. So we compute

$$\begin{aligned}
\check{\beta} &= W\gamma \\
X'd &= X'Y - X'X\check{\beta} \\
X'Md &= X'd - X'XC^{-1}X'd \\
d'Md &= Y'Y - \check{\beta}'(X'Y + X'd) - \hat{u}'X'd \\
XM^2d &= X'd - 2X'X\hat{u} + X'XC^{-1}X'X\hat{u} \\
d'M^2d &= Y'Y - \check{\beta}'(X'Y + X'd) - 2\hat{u}'X'd + \hat{u}'X'X\hat{u}
\end{aligned}$$

Remember that $\hat{u} = C^{-1}X'd$.

Frequently, we would like to pre-multiply one of these statistics by W'. The method `:pre-w` accepts a message for a particular statistic, computes it and pre-multiplies by W'. Similarly, the method `:pre-post-w` pre-multiplies a statistic by W', and pre-multiplies the transpose of the product by W'. Hence, $W'X'MY$ is obtained by `(send unit :pre-w :xmy)`, and $W'XMXW$ by `(send unit :pre-post-w :xmx)`.

Code 3.2

```
|#

(defmeth unit-proto :xmx ()
"Message Args: ()
Returns X'MX where M = sig^2 V^-1."
  (let ((xx (send self :xx))
        (c-inv (send self :c-inv)))
    (- xx (matmult xx c-inv xx))))

(defmeth unit-proto :xmy ()
"Message Args: ()
"Args: (matrix ind)
Returns T if the submatrix of MATRIX indicated by IND
is positive definite."
  (let ((eigenvalues
          (if ind (eigenvalues (select m ind ind)) '(1))))
    (> (prod (if-else (> eigenvalues 0) 1 0)) 0)))

(defmeth unit-proto :c-inv (&optional (tau-inv-ind nil update))
"Message Args: ()
Returns the content of slot c-inv. Used internally to update c-inv."
  (if update
    (let* ((sent (and (listp tau-inv-ind)
                      (matrixp (car tau-inv-ind))))
           (tau-inv (if sent (car tau-inv-ind)
                       (* (send self :terrace :sigma)
                          (send self :terrace :tau-inv))))
          (ind (if sent (second tau-inv-ind)
                 (send self :terrace :rand-ind)))
          (xx (send self :xx)))
      (setf (slot-value 'c-inv)
            (sub-inverse (+ xx tau-inv) ind)))
    (slot-value 'c-inv)))

(defmeth unit-proto :u-hat (&optional update)
"Message Args: ()
Returns the current empirical Bayes estimate of u. Used internally
to update u-hat."
  (if update
    (let ((c-inv (send self :c-inv))
          (xd (send self :xd)))
      (setf (slot-value 'u-hat) (matmult c-inv xd)))
    (slot-value 'u-hat)))

#|
```

Next we develop some useful formulas and methods to compute them. Let

$$M = I - XC^{-1}X' = \sigma^2 V^{-1}.$$

```
"Message Args: (&optional new-z).
Sets and retrieves the contents of slot z."
  (if set (setf (slot-value 'z) new)
    (slot-value 'z)))

(defmeth unit-proto :n-cases (&optional (new nil set))
"Message Args: ()
Retrieves the number of cases in units. Used internally to set the
contents of slot n-cases."
  (if set (setf (slot-value 'n-cases) new)
    (slot-value 'n-cases)))

(defun sub-inverse (m ind)
"Args: (matrix ind)
Returns the inverse of the submatrix of MATRIX indicated by IND."
  (let ((m-inv (* 0 m))
        (sub (select m ind ind)))
    (if ind
      (setf (select m-inv ind ind) (inverse sub)))
    m-inv))

(defun sub-solve (a b ind)
"Args: (a b ind)
Solves the equation Ax = B for rows and columns indicated by IND."
  (let ((x (* 0 b))
        (sub-a (select a ind ind))
        (sub-b (select b ind)))
    (setf (select x ind) (solve sub-a sub-b))
    x))

(defun sub-determinant (m ind)
"Args: (matrix ind)
Returns the determinant of the submatrix of MATRIX indicated
by IND."
  (if ind
    (determinant (select m ind ind))
    1))

(defun sub-posdefp (m ind)
    c-inv          ; helpful matrix
    u-hat          ; predictor of u
   ) () ()
"TERRACE-TWO prototype for subject level model."
)

;;;
;;;    Basic and Slot Accessor Methods
;;;

(defmeth unit-proto :terrace (&rest args)
"Message Args: (message)
Sends message to related terrace."
  (apply #'send (slot-value 'terrace) args))

(defmeth unit-proto :unit-id ()
"Message Args: ()
Retrieves internal identification of unit."
  (slot-value 'unit-id))

(defmeth unit-proto :xx (&optional (new nil set))
"Message Args: (&optional new-xx).
Sets and retrieves the content of slot xx."
  (if set (setf (slot-value 'xx) new)
    (slot-value 'xx)))

(defmeth unit-proto :xy (&optional (new nil set))
"Message Args: (&optional new-xy).
Sets and retrieves the contents of slot xy."
  (if set (setf (slot-value 'xy) new)
    (slot-value 'xy)))

(defmeth unit-proto :yy (&optional (new nil set))
"Message Args: (&optional new-yy).
Sets and retrieves the contents of slot yy."
  (if set (setf (slot-value 'yy) new)
    (slot-value 'yy)))

(defmeth unit-proto :z (&optional (new nil set))
```

helpful. The meaning of the other slots should be clear as they form sufficient statistics.

When we consider the problem of computing for reduced models it will be necessary to compute inverses of submatrices, leaving the rest of the matrix zero. The function `sub-inverse` takes a matrix and index of a submatrix and returns the inverse of the indicated submatrix in original position. We use the notation M^- for the sub-inverse of M. Specifically, if M is partitioned

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix},$$

and we index the submatrix M_{11}, then

$$M^-_{\{1\}} = \begin{pmatrix} M_{11}^{-1} & 0 \\ 0 & 0 \end{pmatrix}.$$

The slot `c-inv` actually contains C^-. The functions `sub-solve` and `sub-determinant` are analogous adaptations of `solve` and `determinant`.

The slot `u-hat` contains the best linear unbiased predictor (BLUP) of u, which is $\hat{u} = C^{-1}X'd$.

Code 3.1

```
|#

(provide "ter2")

;;;;
;;;;    Unit Prototype for TERRACE
;;;;

(defproto unit-proto
  '(terrace         ; object of related terrace
    unit-id         ; number which identifies the unit
    xx              ; X'X
    xy              ; X'Y
    yy              ; Y'Y
    z               ; unit background variables
    n-cases         ; number of cases
```

zero). Such parameters will be called *fixed*. This is distinct from the usage in the context of mixed models where effects are said to be fixed, random, or non-randomly varying. Such modeling choices may be activated by *fixing* parameters or including new (indicator) variables. Thus, the computational problem of model specification becomes a problem of restricted parameter estimation.

3 Defining Prototypes

In TERRACE-TWO, there are two fundamental prototypes: `terrace-proto` and `unit-proto`. The latter will contain essential information for a single unit and represents the level-one model. And the former will communicate with with all the units in the model, maintain parameter estimates, and represent the level-two model. Virtually, our multilevel model will exist a network of many units related to one terrace.

Other prototype inherit from either of the two fundamental prototypes. Such hybrid will exist for specific tasks. For instance, `reg-unit-proto` will inherit from both `unit-proto` and `regression-model-proto`. Inheriting from the latter means that case data will be available along with all of level-one diagnostic methods.

3.1 Unit-proto

First, we need to define `unit-proto`. Since there is no non-trivial inheritance, we only specify the slots. This prototype will contain only slightly more information than necessary. It is intended to conserve memory. The slot `terrace` will enable the unit to send messages to its terrace, and the method `terrace` will facilitate this communication. The slot `c-inv` will contain

$$C_j^{-1} = (X_j' X_j + \sigma^2 \tau^{-1})^{-1},$$

a $(P \times P)$-matrix which is frequently called. Note that since we are speaking of a single unit, we may as well drop the subscript j. Nevertheless, the slot `unit-id` essentially contains j for circumstances where such identification is and

$$W_j = \begin{pmatrix} z_j & 0 & \cdots & 0 \\ 0 & z_j & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & z_j \end{pmatrix}.$$

Note that W_j is a $(P \times QP)$-matrix. Now our second level model may be completely specified by (3) when we impose that each $u_j \overset{iid}{\sim} N(0, \tau)$. Also we impose that $Cov(u_j, r_{j'}) = 0$ for any j and j'. This model will be referred to as level-two model, and we will also define an object to represent this model. Level-one objects will be embedded in a level-two object. Thus, the multilevel model will be represented as a network of models.

Combining (1) and (3), one obtains

$$Y_j = X_j W_j \gamma + X_j u_j + r_j. \qquad \textbf{(4)}$$

Evidently,

$$Y_j - X_j W_j \gamma = X_j u_j + r_j \sim N(0, V_j), \qquad \textbf{(5)}$$

where

$$V_j = X_j \tau X_j' + \sigma^2 I_{n_j}.$$

Thus, the full log-likelihood for the jth unit is

$$L_j(\sigma^2, \tau, \gamma) = -\frac{n_j}{2}\log(2\pi) - \frac{1}{2}\log|V_j| - \frac{1}{2}d_j' V_j^{-1} d_j, \qquad \textbf{(6)}$$

where $d_j = Y_j - X_j W_j \gamma$. Since the J units are independent, we write the log-likelihood for the entire model as a sum of unit log-likelihoods, *i.e.*,

$$L(\sigma^2, \tau, \gamma) = \sum_{j=1}^{J} L_j(\sigma^2, \tau, \gamma). \qquad \textbf{(7)}$$

This decomposition of the log-likelihood function, implies that the information matrix and score function (first and second derivatives) may likewise be written as sums of unit components. These facts motivate the object structure discussed in the next section.

Thus far, I have presented what may be called a maximal model, meaning all parameters in σ^2, τ, and γ are to be estimated or *free*. Reduced models may be specified by setting certain parameters to arbitrary values (usually scoring algorithm as outlined in the technical appendix of Bryk & Raudenbush (1992). The deletion and local perturbation diagnostics are developed in Hilden-Minton(1993, 1994) and are here adapted to TERRACE-TWO.

The purpose of this paper is to motivate the object structure and explicate the code of TERRACE-TWO. The basic bi-level model will be introduced.

2 Description of a multilevel model

Suppose we have N subjects naturally grouped into J units, where there are n_j subjects in the jth unit and $\sum_{j=1}^{J} n_j = N$. Further suppose that for the J units we want to regress the response variable Y_j on matrix of P predictor variables X_j. Thus, for the jth unit we model

$$Y_j = X_j \beta_j + r_j, \qquad \mathbf{(1)}$$

where each X_j has dimensions $n_j \times P$, and

$$r_j \sim N(0, \sigma^2 I_{n_j}).$$

These models will be referred to as level-one models, and we will define objects to represents these models.

At the next level, we want to model each β_{jp} ($j = 1, 2, \ldots, J$, and $p = 1, 2, \ldots, P$) with

$$\beta_{jp} = z_j' \gamma_p + u_{jp}, \qquad (2)$$

where z_j is an vector of Q background variable on the jth unit and $u_{jp} \sim N(0, \tau_{pp})$. But the u_{jp} are not independent; to get at their covariance structure, we "stack" the equations in (2) to obtain

$$\beta_j = W_j \gamma + u_j, \qquad \mathbf{(3)}$$

where

$$u_j = (u_{j1}, u_{j2}, \ldots, u_{jP})',$$

$$\gamma = (\gamma_1', \gamma_2', \ldots, \gamma_P')',$$

TERRACE-TWO: a new XLISP-STAT package for multilevel modeling with diagnostics

James Hilden-Minton

July 15, 1994

WORK IN PROGRESS

1 Introduction

TERRACE-TWO is an XLISP-STAT package of objects with which one may interactively construct, estimate, and diagnose multilevel models. As the next step in the evolution of TERRACES, TERRACE-TWO has a new, more natural object structure. New attention is given to the computation of Fisher and observed information matrices and to diagnostics via local perturbation schemes. The package accepts bi-level, or once clustered, data and estimates the parameters of the maximal hierarchical linear model using restricted or full maximum likelihood estimation. Reduced models can be obtained by *fixing* specific parameters. The user can interact with the model to extract estimates of the parameters, calculate diagnostic statistics, make graphs, and to perform whatever analyses the user may envision. The program (and notation) follows the implementation of Longword's (1987) Fisher