# Chapter 1

# Writing New Models

With Zelig, writing a new model in R is straightforward. (If you already have a model, see Chapter **??** for how to include it in Zelig.) With tools to streamline user inputs, writing a new model does not require a lot of programming knowledge, but lets developers focus on the model's math. Generally, writing a new statistical procedure or model comes in orderly steps:

1. Write down the mathematical model. Define the parameters that you need, grouping parameters into convenient vectors or matrices whenever possible (this will make your code clearer).
2. Write the code.
3. Test the code (usually using Monte Carlo data, where you know the true values being estimated ) and make sure that it works as expected.
4. Write some documentation explaining your model and the functions that run your model.

Somewhere between steps [1] and [2], you will need to translate input data into the mathematical notation that you used to write down the model. Rather than repeating whole blocks of code, use functions to streamline the number of commands that users will need to run your model.

With more steps being performed by fewer commands, the inputs to these commands become more sophisticated. The structure of those inputs actually matters quite a lot. If your function has a convoluted syntax, it will be difficult to use, difficult to explain, and difficult to document. If your function is easy to use and has an intuitive syntax, however, it will be easy to explain and document, which will make your procedure more accessible to all users.

## 1.1 Managing Statistical Model Inputs

Most statistical models require a matrix of explanatory variables and a matrix of dependent variables. Rather than have users create matrices themselves, R has a convenient user interface to create matrices of response and explanatory variables on the fly. Users simply specify a `formula` in the form of `dependent ~ explanatory variables`, and developers use the following functions to transform the formula into the appropriate matrices. Let `mydata` be a data frame.

```
> formula <- y ~ x1 + x2                      # User input

# Given the formula above, programmers can use the following standard commands
> D <- model.frame(formula, data = mydata) # Subset & listwise deletion
> X <- model.matrix(formula, data = D)     # Creates X matrix
> Y <- model.response(D)                   # Creates Y matrix
```

where

- `D` is a subset of `mydata` that contains only the variables specified in the formula (`y`, `x1`, and `x2`) with listwise deletion performed on the subset data frame;
- `X` is a matrix that contains a column of 1's, and the explanatory variables `x1` and `x2` from `D`; and
- `Y` is a matrix containing the dependent variable(s) from `D`.

Depending on the model, `Y` may be a column vector, matrix, or other data structure.

### 1.1.1 Describe the Statistical Model

After setting up the $X$ matrix, the next step for most models will be to identify the corresponding vector of parameters. For a single response variable model with no ancillary parameters, the standard R interface is quite convenient: given $X$, the model's parameters are simply $\beta$.

There are very few models, however, that fall into this category. Even Normal regression, for example, has two sets of parameters $\beta$ and $\sigma^2$. In order to make the R formula format more flexible, Zelig has an additional set of tools that lets you describe the inputs to your model (for multiple sets of parameters).

After you have written down the statistical model, identify the parameters in your model. With these parameters in mind, the first step is to write a `describe.*()` function for your model. If your model is called `mymodel`, then the `describe.mymodel()` function takes no arguments and returns a list with the following information:

- `category`: a character string that describes the dependent variable. See Section **??** for the current list of available categories.

- `parameters`: a list containing parameter sets used in your model. For each parameter (e.g., theta), you need to provide the following information:
  - `equations`: an integer number of equations for the parameter. For parameters that can take, for example, two to four equations, use `c(2, 4)`.
  - `tagsAllowed`: a logical value (`TRUE`/`FALSE`) specifying whether a given parameter allows constraints.
  - `depVar`: a logical value (`TRUE`/`FALSE`) specifying whether a parameter requires a corresponding dependent variable.
  - `expVar`: a logical value (`TRUE`/`FALSE`) specifying whether a parameter allows explanatory variables.

(See Section **??** for examples and additional arguments output by `describe.mymodel()`.)

### 1.1.2 Single Response Variable Models: Normal Regression Model

Let's say that you are trying to write a Normal regression model with stochastic component

$$\text{Normal}(y_i \mid \mu_i, \sigma^2) \;=\; \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\left(\frac{(y_i - \mu_i)^2}{2\sigma^2}\right)\right)$$

with scalar variance parameter $\sigma^2 > 0$, and systematic component $E(Y_i) = \mu_i = x_i\beta$. This implies two sets of parameters in your model, and the following `describe.normal.regression()` function:

```
describe.normal.regression <- function() {
  category <- "continuous"
  mu <- list(equations = 1,              # Systematic component
             tagsAllowed = FALSE,
             depVar = TRUE,
             expVar = TRUE)
  sigma2 <- list(equations = 1,          # Scalar ancillary parameter
                 tagsAllowed = FALSE,
                 depVar = FALSE,
                 expVar = FALSE)
  pars <- list(mu = mu, sigma2 = sigma2)
  list(category = category, parameters = pars)
}
```

To find the log-likelihood:

$$\begin{aligned}
\text{L}\ (\beta, \sigma^2 \mid y) &= \prod_{1=1}^{n} \text{Normal}(y_i \mid \mu_i, \sigma^2) \\
&= \prod_{i=1}^{n} (2\pi\sigma^2)^{-1/2} \exp\left(\frac{-(y_i - \mu_i)^2}{2\sigma^2}\right) \\
&= (2\pi\sigma^2)^{-n/2} \prod_{i=1}^{n} \exp\left(\frac{-(y_i - \mu_i)^2}{2\sigma^2}\right) \\
&= (2\pi\sigma^2)^{-n/2} \prod_{i=1}^{n} \exp\left(\frac{-(y_i - x_i\beta)^2}{2\sigma^2}\right) \\
\ln \text{L}\ (\beta, \sigma^2 \mid y) &= -\frac{n}{2}\ln(2\pi\sigma^2) - \sum_{i=1}^{n} \frac{(y_i - x_i\beta)^2}{2\sigma^2} \\
&= -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - x_i\beta)^2 \\
&\propto -\frac{1}{2}\left(n \ln \sigma^2 + \frac{\sum_{i=1}^{n}(y_i - x_i\beta)^2}{\sigma^2}\right)
\end{aligned}$$

In R code, this translates to:

```
ll.normal <- function(par, X, Y, n, terms) {
  beta <- parse.par(par, terms, eqn = "mu")              # [1]
  gamma <- parse.par(par, terms, eqn = "sigma2")         # [2]
  sigma2 <- exp(gamma)
  -0.5 * (n * log(sigma2) + sum((Y - X %*% beta)^2 / sigma2))
}
```

At Comment [1] above, we use the function `parse.par()` to pull out the vector of parameters `beta` (which relate the systematic component $\mu_i$ to the explanatory variables $x_i$). No matter how many covariates there are, the `parse.par()` function can use `terms` to pull out the appropriate parameters from `par`. We also use `parse.par()` at Comment [2] to pull out the scalar ancillary parameter that (after transformation) corresponds to the $\sigma^2$ parameter.

To optimize this function, simply type:

```
out <- optim(start.val, ll.normal, control = list(fnscale = -1),
             method = "BFGS", hessian = TRUE, X = X, Y = Y, terms = terms)
```

where

- `start.val` is a vector of starting values for `par`. Use `set.start()` to create starting values for all parameters, systematic and ancillary, in one step.
- `ll.normal` is the log-likelihood function derived above.

- "BFGS" specifies unconstrained optimization using a quasi-Newton method.
- control = list(fnscale = -1) specifies that R should maximize the function (omitting this causes R to minimize the function by default).
- hessian = TRUE instructs R to return the Hessian matrix (from which you may calculate the variance-covariance matrix).
- X and Y are the matrix of explanatory variables and vector of dependent variables, used in the ll.normal() function.
- terms are meta-data constructed from the model.frame() command.

Please refer to the R-help for optim() for more options.

To make this procedure generalizable, we can write a function that takes a user-specified data frame and formula, and optional starting values for the optimization procedure:

```
normal.regression <- function(formula, data, start.val = NULL, ...) {

  fml <- parse.formula(formula, model = "normal.regression") # [1]
  D <- model.frame(fml, data = data)
  X <- model.matrix(fml, data = D)
  Y <- model.response(D)
  terms <- attr(D, "terms")
  n <- nrow(X)

  start.val <- set.start(start.val, terms)

  res <- optim(start.val, ll.normal, method = "BFGS",
               hessian = TRUE, control = list(fnscale = -1),
               X = X, Y = Y, n = n, terms = terms, ...)      # [2]

  fit <- model.end(res, D)                                   # [3]
  fit$n <- n
  class(fit) <- "normal"                                     # [4]
  fit
}
```

The following comments correspond to the bracketed numbers above:

1. The parse.formula() command looks for the describe.normal.regression() function, which changes the user-specified formula into the following format:

   ```
   list(mu = formula,         # where `formula' was specified by the user
        sigma = ~ 1)
   ```

2. The `...` here indicate that if the user enters any additional arguments when calling `normal.regression()`, that those arguments should go to the `optim()` function.

3. The `model.end()` function takes the optimized output and the listwise deleted data frame `D` and creates an object that will work with `setx()`.

4. Choose a class for your model output so that you will be able to write an appropriate `summary()`, `param()`, and `qi()` function for your model.

### 1.1.3 Multivariate models: Bivariate Normal example

Most common models have one systematic component. For $n$ observations, the systematic component varies over observations $i = 1, \ldots, n$. In the case of the Normal regression model, the systematic component is $\mu_i$ ($\sigma^2$ is not estimated as a function of covariates).

In some cases, however, your model may have more than one systematic component. In the case of bivariate probit, we have a dependent variable $Y_i = (Y_{i1}, Y_{i2})$ observed as (0,0), (1,0), (0,1), or (1,1) for $i = 1, \ldots, n$. Similar to a single-response probit model, the stochastic component is described by two latent (unobserved) continuous variables ($Y^*_{i1}$, $Y^*_{i2}$) which follow the bivariate Normal distribution:

$$\begin{pmatrix} Y^*_{i1} \\ Y^*_{i2} \end{pmatrix} \sim \text{Normal}\left\{ \begin{pmatrix} \mu_{i1} \\ \mu_{i2} \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right\},$$

where for $j = 1, 2$, $\mu_{ij}$ is the mean for $Y^*_{ij}$ and $\rho$ is a correlation parameter. The following observation mechanism links the observed dependent variables, $Y_{ij}$, with these latent variables

$$Y_{ij} = \begin{cases} 1 & \text{if } Y^*_{ij} \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

The systemic components for each observation are

$$\begin{aligned} \mu_{ij} &= x_{ij}\beta_j \quad \text{for} \quad j = 1, 2, \\ \rho &= \frac{\exp(x_{i3}\beta_3) - 1}{\exp(x_{i3}\beta_3) + 1}. \end{aligned}$$

In the default specification, $\rho$ is a scalar (such that $x_{i3}$ only contains an intercept term).

If so, we have two sets of parameters: $\mu_i = (\mu_{i1}, \mu_{i2})$ and $\rho$. This implies the following `describe.bivariate.probit()` function:

```
describe.bivariate.probit <- function() {
  category <- "dichotomous"
  package <- list(name = "mvtnorm",       # Required package and
                  version = "0.7")        #  minimum version number
  mu <- list(equations = 2,               # Systematic component has 2
             tagsAllowed = TRUE,          #  required equations
```

```
               depVar = TRUE,
               expVar = TRUE),
  rho <- list(equations = 1,              # Optional systematic component
               tagsAllowed = FALSE,        #   (estimated as an ancillary
               depVar = FALSE,             #    parameter by default)
               expVar = TRUE),
  pars <- parameters(mu = mu, rho = rho)
  list(category = category, package = package, parameters = pars)
}
```

Since users may choose different explanatory variables to parameterize $\mu_{i1}$ and $\mu_{i2}$ (and sometimes $\rho$), the model requires a minimum of *two* formulas. For example,

```
formulae <- list(mu1 = y1 ~ x1 + x2,                            # User input
                 mu2 = y2 ~ x2 + x3)
fml <- parse.formula(formulae, model = "bivariate.probit")   # [1]
D <- model.frame(fml, data = mydata)
X <- model.matrix(fml, data = D)
Y <- model.response(D)
```

At comment [1], `parse.formula()` finds the `describe.bivariate.probit()` function and parses the formulas accordingly.

If $\rho$ takes covariates (and becomes a systematic component rather than an ancillary parameter), there can be three sets of explanatory variables:

```
formulae <- list(mu1 = y1 ~ x1 + x2,
                 mu2 = y2 ~ x2 + x3,
                 rho = ~ x4 + x5)
```

From the perspective of the programmer, a nearly identical framework works for both single and multiple equation models. The (`parse.formula()`) line changes the class of `fml` from `"list"` to `"multiple"` and hence ensures that `model.frame()` and `model.matrix()` go to the appropriate methods. `D`, `X` , and `Y` are analogous to their single equation counterparts above:

- `D` is the subset of `mydata` containing the variables `y1`, `y2`, `x1`, `x2`, and `x3` with listwise deletion performed on the subset;

- `X` is a matrix corresponding to the explanatory variables, in one of three forms discussed below (see Section **??**).

- `Y` is an $n \times J$ matrix (where $J = 2$ here) with columns (`y1`, `y2`) corresponding to the outcome variables on the left-hand sides of the formulas.

Given for the bivariate probit probability density described above, the likelihood is:

$$L(\pi|Y_i) = \prod_{i=1}^{n} \pi_{00}^{\mathrm{I}\{Y_i=(0,0)\}} \pi_{10}^{\mathrm{I}\{Y_i=(1,0)\}} \pi_{01}^{\mathrm{I}\{Y_i=(0,1)\}} \pi_{11}^{\mathrm{I}\{Y_i=(1,1)\}}$$

where I is an indicator function and

- $\pi_{00} = \int_{-\infty}^{0}\int_{-\infty}^{0} \mathrm{Normal}(Y_{i1}^*, Y_{i2}^* \mid \mu_{i1}, \mu_{i2}, \rho) dY_{i2}^* dY_{i1}^*$
- $\pi_{10} = \int_{0}^{\infty}\int_{-\infty}^{0} \mathrm{Normal}(Y_{i1}^*, Y_{i2}^* \mid \mu_{i1}, \mu_{i2}, \rho) dY_{i2}^* dY_{i1}^*$
- $\pi_{01} = \int_{-\infty}^{0}\int_{0}^{\infty} \mathrm{Normal}(Y_{i1}^*, Y_{i2}^* \mid \mu_{i1}, \mu_{i2}, \rho) dY_{i2}^* dY_{i1}^*$
- $\pi_{11} = 1 - \pi_{00} - \pi_{10} - \pi_{01}$

This implies the following log-likelihood:

$$\begin{aligned} \log L(\pi|Y_i) &= \sum_{i=1}^{n} \mathrm{I}\{Y_i = (0,0)\} \log \pi_{00} + \mathrm{I}\{Y_i = (1,0)\} \log \pi_{10} \\ &\quad + \mathrm{I}\{Y_i = (0,1)\} \log \pi_{01} + \mathrm{I}\{Y_i = (1,1)\} \log \pi_{11} \end{aligned}$$

(For the corresponding R code, see Section **??** below.)

## 1.2 Easy Ways to Manage Matrices

Most statistical methods relate explanatory variables $x_i$ to a dependent variable of interest $y_i$ for each observation $i = 1, \ldots, n$. Let $\beta$ be a set of parameters that correspond to each column in $X$, which is an $n \times k$ matrix with rows $x_i$. For a single equation model, the linear predictor is

$$\eta_i = x_i\beta = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_k x_{ik}$$

Thus, $\eta$ is the set of $\eta_i$ for $i = 1, \ldots, n$ and is usually represented as an $n \times 1$ matrix.

For a two equation model such as bivariate probit, the linear predictor becomes a matrix with columns corresponding to each dependent variable $(y_{1i}, y_{2i})$:

$$\eta_i = (\eta_{i1}, \eta_{i2}) = (x_{i1}\beta_1, x_{i2}\beta_2)$$

With $\eta$ as an $n \times 2$ matrix, we now have a few choices as to how to create the linear predictor:

1. An **intuitive** layout, which stacks matrices of explanatory variables, provides an easy visual representation of the relationship between explanatory variables and coefficients;
2. A **computationally-efficient** layout, which takes advantage of computational vectorization; and
3. A **memory-saving** layout, which reduces the overall size of the $X$ and $\beta$ matrices.

Using the simple tools described in this section, you can pick the best matrix management method for your model.

In addition, the way in which $\eta$ is created also affects the way parameters are estimated. Let's say that you want two parameters to have the same effect in different equations. By setting up $X$ and $\beta$ in a certain way, you can let users set constraints across parameters. Continuing the bivariate probit example above, let the model specification be:

```
formulae <- list(mu1 = y1 ~ x1 + x2 + tag(x3, "land"),
                 mu2 = y2 ~ x3 + tag(x4, "land"))
```

where `tag()` is a special function that constrains variables to have the same effect across equations. Thus, the coefficient for `x3` in equation `mu1` is constrained to be equal to the coefficient for `x4` in equation `mu2`, and this effect is identified as the "land" effect in both equations. In order to consider constraints across equations, the structure of both $X$ and $\beta$ matter.

### 1.2.1 The Intuitive Layout

A stacked matrix of $X$ and vector $\beta$ is probably the most visually intuitive configuration. Let $J = 2$ be the number of equations in the bivariate probit model, and let $v_t$ be the total number of unique covariates in both equations. Choosing `model.matrix(..., shape = "stacked")` yields a $(Jn \times v_t) = (2n \times 6)$ matrix of explanatory variables. Again, let $x_1$ be an $n \times 1$ vector representing variable `x1`, $x_2$ `x2`, and so forth. Then

$$X = \begin{pmatrix} 1 & 0 & x_1 & x_2 & 0 & x_3 \\ 0 & 1 & 0 & 0 & x_3 & x_4 \end{pmatrix}$$

Correspondingly, $\beta$ is a vector with elements

$$(\beta_0^{\mu_1} \; \beta_0^{\mu_2} \; \beta_{x_1}^{\mu_1} \; \beta_{x_2}^{\mu_1} \; \beta_{x_3}^{\mu_2} \; \beta_{\text{land}})'$$

where $\beta_0^j$ are the intercept terms for equation $j = \{\mu_1, \mu_2\}$. Since $X$ is $(2n \times 6)$ and $\beta$ is $(6 \times 1)$, the resulting linear predictor $\eta$ is also stacked into a $(2n \times 1)$ matrix. Although difficult to manipulate (since observations are indexed by $i$ and $2i$ for each $i = 1, \ldots, n$ rather than just $i$), it is easy to see that we have turned the two equations into one big $X$ matrix and one long vector $\beta$, which is directly analogous to the familiar single-equation $\eta$.

### 1.2.2 The Computationally-Efficient Layout

Choosing array $X$ and vector $\beta$ is probably the the most computationally-efficient configuration: `model.matrix(..., shape = "array")` produces an $n \times k_t \times J$ array where $J$ is the total number of equations and $k_t$ is the total number of parameters across all the equations. Since some parameter values may be constrained across equations, $k_t \leq \sum_{j=1}^{J} k_j$. If a variable is not in a certain equation, it is observed as a vector of 0s. With this option, each $i = 1, \ldots, n$ $x_i$ matrix becomes:

$$\begin{pmatrix} 1 & 0 & x_{i1} & x_{i2} & 0 & x_{i3} \\ 0 & 1 & 0 & 0 & x_{i3} & x_{i4} \end{pmatrix}$$

By stacking each of these $x_i$ matrices along the first dimension, we get $X$ as an array with dimensions $n \times k_t \times J$.

Correspondingly, $\beta$ is a vector with elements

$$(\beta_0^{\mu_1} \; \beta_0^{\mu_2} \; \beta_{x_1}^{\mu_1} \; \beta_{x_2}^{\mu_1} \; \beta_{x_3}^{\mu_2} \; \beta_{\text{land}})'$$

To multiply the $X$ array with dimensions $(n \times 6 \times 2)$ and the $(6 \times 1)$ $\beta$ vector, we *vectorize* over equations as follows:

```
eta <- apply(X, 3, '%*%', beta)
```

The linear predictor `eta` is therefore a $(n \times 2)$ matrix.

### 1.2.3 The Memory-Efficient Layout

Choosing a "compact" $X$ matrix and matrix $\beta$ is probably the most memory-efficient configuration: `model.matrix(..., shape = "compact")` (the default) produces an $n \times v$ matrix, where $v$ is the number of unique variables (5 in this case)[1] in all of the equations. Let $x_1$ be an $n \times 1$ vector representing variable `x1`, $x_2$ `x2`, and so forth.

$$X = (1 \; x_1 \; x_2 \; x_3 \; x_4) \qquad \beta = \begin{pmatrix} \beta_0^{\mu_1} & \beta_0^{\mu_2} \\ \beta_{x_1}^{\mu_1} & 0 \\ \beta_{x_2}^{\mu_1} & 0 \\ \beta_{\text{land}} & \beta_{x_3}^{\mu_2} \\ 0 & \beta_{\text{land}} \end{pmatrix}$$

The $\beta_{\text{land}}$ parameter is used twice to implement the constraint, and the number of empty cells is minimized by implementing the constraints in $\beta$ rather than $X$. Furthermore, since $X$ is $(n \times 5)$ and $\beta$ is $(5 \times 2)$, $X\beta = \eta$ is $n \times 2$.

### 1.2.4 Interchanging the Three Methods

Continuing the bivariate probit example above, we only need to modify a few lines of code to put these different schemes into effect. Using the default (memory-efficient) options, the log-likelihood is:

[1] Why 5? In addition to the intercept term (a variable which is the same in either equation, and so counts only as one variable), the *unique* variables are $x_1$, $x_2$, $x_3$, and $x_4$.

```
bivariate.probit <- function(formula, data, start.val = NULL, ...) {
  fml <- parse.formula(formula, model = "bivariate.probit")
  D <- model.frame(fml, data = data)
  X <- model.matrix(fml, data = D, eqn = c("mu1", "mu2"))       # [1]
  Xrho <- model.matrix(fml, data = D, eqn = "rho")
  Y <- model.response(D)
  terms <- attr(D, "terms")
  start.val <- set.start(start.val, terms)
  start.val <- put.start(start.val, 1, terms, eqn = "rho")

  log.lik <- function(par, X, Y, terms) {
    Beta <- parse.par(par, terms, eqn = c("mu1", "mu2"))         # [2]
    gamma <- parse.par(par, terms, eqn = "rho")
    rho <- (exp(Xrho %*% gamma) - 1) / (1 + exp(Xrho %*% gamma))
    mu <- X %*% Beta                                             # [3]
    llik <- 0
    for (i in 1:nrow(mu)){
      Sigma <- matrix(c(1, rho[i,], rho[i,], 1), 2, 2)
      if (Y[i,1]==1)
        if (Y[i,2]==1)
          llik <- llik + log(pmvnorm(lower = c(0, 0), upper = c(Inf, Inf),
                                     mean = mu[i,], corr = Sigma))
        else
          llik <- llik + log(pmvnorm(lower = c(0, -Inf), upper = c(Inf, 0),
                                     mean = mu[i,], corr = Sigma))
      else
        if (Y[i,2]==1)
          llik <- llik + log(pmvnorm(lower = c(-Inf, 0), upper = c(0, Inf),
                                     mean = mu[i,], corr = Sigma))
        else
          llik <- llik + log(pmvnorm(lower = c(-Inf, -Inf), upper = c(0, 0),
                                     mean = mu[i,], corr = Sigma))
        }
    return(llik)
  }
  res <- optim(start.val, log.lik, method = "BFGS",
               hessian = TRUE, control = list(fnscale = -1),
               X = X, Y = Y, terms = terms, ...)
  fit <- model.end(res, D)
  class(fit) <- "bivariate.probit"
  fit
}
```

If you find that the default (memory-efficient) method isn't the best way to run your model, you can use either the intuitive option or the computationally-efficient option by changing just a few lines of code as follows:

- **Intuitive option** At Comment [1]:

  ```
  X <- model.matrix(fml, data = D, shape = "stacked", eqn = c("mu1", "mu2"))
  ```

  and at Comment [2],

  ```
  Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
  ```

  The line at Comment [3] remains the same as in the original version.

- **Computationally-efficient option** Replace the line at Comment [1] with

  ```
  X <- model.matrix(fml, data = D, shape = "array", eqn = c("mu1", "mu2"))
  ```

  At Comment [2]:

  ```
  Beta <- parse.par(par, terms, shape = "vector", eqn = c("mu1", "mu2"))
  ```

  At Comment [3]:

  ```
  mu <- apply(X, 3, '%*%', Beta)
  ```

Even if your optimizer calls a C or FORTRAN routine, you can use combinations of `model.matrix()` and `parse.par()` to set up the data structures that you need to obtain the linear predictor (or your model's equivalent) before passing these data structures to your optimization routine.