# Zelig Developers' Manual `v1.0`

January 27, 2012

# Contents

# Chapter 1

# Quick Start Guide to Making a Zelig Model

## 1.1 Introduction

Programming a Zelig module is a simple procedure. By following several steps, any statistical model can be implemented in the Zelig software suite. The following document places emphasis on speed and practicality, rather than the numerous, technical details involved in developing statistical models. That is, this guide will explain how to quickly and most simply include existing statistical software in the Zelig suite.

## 1.2 Overview

In order for a Zelig model to function correctly, four components need to exist:

***a statistical model*:** This can be any statistical model of the developer's choosing, though it is suggested that it be written in R. Examples of statistical models already implemented in Zelig include: Brian Ripley's `glm` and Kosuke Imai's `MNP` models.

`zelig2`***model*** This method acts as a bridge between the external statistical model and the Zelig software suite

`param.`***model*** This method specifies the simulated parameters used to compute quantities of interest

`qi.`***model*** This method computes - using the fitted statistical model, simulated parameters, and explanatory data - the *quantities of interest*. Compared with the `zelig2` and `param` methods,

In the above description, replace the italicized *model* text with the name of the developer's model. For example, if the model's name is "logit", then the corresponding methods will be titled `zelig2logit`, `param.logit`, and `qi.logit`.

## 1.3 `zelig.skeleton`: Automating Zelig Model Creation

The fastest way to setup and begin programming a Zelig model is the use the `zelig.skeleton` function, available within the `Zelig` package. This function allows a fast, simple way to create the `zelig2`, `describe`, `param`, and `qi` methods with the necessary boilerplate. As a result, `zelig.skeleton` closely mirrors the `package.skeleton` method included in core R.

### 1.3.1 A Demonstrative Example

```
library(Zelig)  # [1]

zelig.skeleton(
               "my.zelig.package",                # [2]
               models = c("gamma", "logit"),      # [3]
               author = "Your Name",              # [4]
               email = "your.email@someplace.com" # [5]
               )
```

### 1.3.2 Explanation of the `zelig.skeleton` Example

The above numbered comments correspond to the following:

**[1]** The Zelig package must be imported when using `zelig.skeleton`.

**[2]** The first parameter of `zelig.skeleton` specifies the name of the package

**[3]** The `models` parameter specifies the titles of the Zelig models to be included in the package. In the above example, all necessary files and methods for building the "gamma" and "logit" models will be included in Zelig package.

**[4]** Specify the author's name

**[5]** Specify the email address of the software maintainer

### 1.3.3 Conclusion

The `zelig.skeleton` function provides a way to automatically generate the necessary methods and file to create an arbitrary Zelig package. The method body, however, will be incomplete, save for some light documentation additions and

programming boilerplate. For a detailed specification of the `zelig.skeleton` method, refer to Zelig help file by typing:

```
library(Zelig)

?zelig.skeleton
```

in an interactive R-session.

## 1.4 *zelig2*: Interacting with Existing Statistical Models in Zelig

The `zelig2` function acts as the bridge between the Zelig module and the existing statistical model. That is, the results of this function specify the parameters to be passed to a *previously completed* statistical model-fitting function. In this sense, there is nothing tricky about the `zelig2` function. Simply construct a list with key-value pairs in the following fashion:

- **Keys** (names on the lefthand-side of an equal sign) represent parameters that are submitted to the existing model function
- **Values** (variables, etc. on the righthand-side of an equal sign) represent values to set the corresponding the parameter to.
- **Keys with leading periods** are typically reserved for specific `zelig2` purposes. In particular, the key `.function` specifies the name of the function that calls the existing statistical model.

an ellipsis (...) specifies that all additional, optional parameters not specified in the signature of the `zelig2model_function` method, will be included in the external method's call, despite not being specifically set.

### 1.4.1 A Simple Example

For example, if a developer wanted to call an existing model `"SomeModel"` with the parameter `weights` set to `1`, the appropriate return-value (a list) for the `zelig2` function would be:

```
zelig2some.model <- function(formula, data) {
    list(.function = "SomeModel",
         formula   = formula,
         weights   = 1
         )
}
```

### 1.4.2 A More Detailed Example

A more typical example would be the case of fitting a basic logistic regression. The following code, already implemented in Zelig, acts as an interface between Zelig packages and R's built-in `glm` function:

```
zelig2logit <- function (formula, weights = NULL, ..., data) {
  list(.function = "glm",    # [1]

       formula = formula,    # [2]
       weights = weights,    # ...
       data    = data,       # ...

       family  =             # [3]
                 binomial(link="logit"),
       model   = FALSE       # ...
       )
}
```

The comments in the above code correspond to the following:

**[1]** Pass all parameters to the `glm` function

**[2]** Specify that the parameters `formula`, `weights`, and `data` be given the same values as those passed into the `zelig2` function itself. That is, whichever values the end-user passes to `zelig` will be passed to the `glm` function

**[3]** Specify that the parameters `family` and `model` *always* be given the corresponding values - `binomial(link="logit")` and `FALSE` - regardless of what the end-user passes as a parameter.

Note that the parameters - `formula`, `weights`, `data`, `family`, `model` - correspond to those of the `glm` function. In general, this will be the case for any `zelig2` method. That is, every `zelig2` method should return a list containing the parameters belonging to the external model, as well as, the reserved keyword `.function`.

If you are unsure about the parameters that are passed to an existing statistical model, simply use the `args` or `formals` functions (included in R). For example, to get a list of acceptable parameters to the `glm` function, simply type:

```
args(glm)
```

### 1.4.3 An Even-More Detailed Example

Occasionally the statistical model and the standard style of Zelig input differ. In these instances, it may be necessary to manipulate information about the `formula` and `constraints`. This additional step in building the `zelig2` method is common only amongst multivariate models, as seen below in the `bprobit` model (bivariate probit regression for Zelig).

```
zelig2bprobit <- function(formula, ..., data) {

  # [1]
  formula <- parse.formula(formula, "bprobit")

  # [2]
  tmp <- cmvglm(formula, "bprobit", 3)


  # return list
  list(
       .function = "vglm",    # [3]

       formula = tmp$formula, # [4]
       family  = bprobit,     # [5]
       data = data,
       # [6]
       constraints = tmp$constraints
       )
}
```

**The following is an explanation of the above code:**

[**1**] Convert Zelig-style `formula` data-types into the style that the `vglm` function understands

[**2**] Extract constraint information from the `formula` object, as is the style commonly supported by Zelig

[**3**] Specify the `vglm` as the statistical model fitting function

[**4**] Specify the formula to be used by the `vglm` function when performing the model fitting. Note that this object is created by using both the `parseformula` and `cmvglm` functions

[**5**] Specify the `family` of the model

[**6**] Specify the constraints to be used by the `vglm` function when performing the model fitting. Note that this object is created by using both the `parseformula` and `cmvglm` functions

Note that the functions `parse.formula` and `cmvglm` are included in the core Zelig software package. Information concerning these functions can be found by typing:

```
library(Zelig)

?parase.formula
?cmvglm
```

in an interactive R-session.

### 1.4.4 Summary and More Information aboput `zelig2` Methods

`zelig2` functions can be of varying difficulty - from simple parameter passing to reformatting and creating new data objects to be used by the external model-fitting function. To see more examples of this usage, please refer to the `survey.zelig` and `multinomial.zelig` packages. Regardless of the model's complexity, it ends with a simple list specifying which parameters to pass to a preexisting statistical model.

For more information on the `zelig2` function's full features, see the *Advanced zelig2 Manual*, or type:

```
library(Zelig)

?zelig2
```

within an interactive R-session.

## 1.5 *param*: Simulating Parameters

The `param` function simulates and specifies parameters necessary for computing *quantities of interest*. That is, the `param` function is the ideal place to specify information necessary for the `qi` method. This includes:

**Ancillary parameters** These parameters specifying information about the underlying probability distribution. For example, in the case of the Normal Distribution, $\sigma$ (standard deviation) and $\mu$ (mean) would be considered ancillary parameters.

**Link function** That is, the function providing the relationship between the predictors and the mean of the distribution function. This is typically of very little importance (compared to the inverse link function), but is frequently included for completeness. For Gamma distribution, the link function is the inverse function: $f(x) = \frac{1}{x}$

**Inverse link function** Typically crucial for simulating *quantities of interest* of *Generalized Linear Models*. For the binomial distribution, the inverse-link function is the logit function: $f(x) = \frac{e^x}{1+e^x}$

**Simulated Parameters** These random draws simulate the parameters of the fitted statistical model. Typically, the `qi` method uses these to simulate *quantities of interest* for the given model. As a result, these are of paramount importance.

The following sections describe how these ideas correspond to the structure of a well-written `param` function.

### 1.5.1 The Function Signature

The `param` function takes only two parameters, but outputs a wealth of information important in computing *quantities of interest*. The following is the function signature:

```
param.logit <- function (obj, num)
```

The above parameters are:

**obj** An object of class `zelig` [1]. This contains the fitted statistical model and associated information.

**num** An integer specifying the number of simulations to be drawn. This value is specified by the end-user, and defaults to `1000` if no value is specified.

### 1.5.2 The Function Return Value

In similar fashion to the `zelig2` method, the `param` method takes return values as a list of key-value pairs. However, the options are not as diverse. That is, the list can only be given a set of specific values: `ancillary`, `coef`, `simulations`, `link`, `linkinv`, and `family`.

In most cases, however, the parameters `ancillary`, `simulations`, and `linkinv` are sufficient. The following is an example take from Zelig's `gamma` model:

[1] For a detailed specification of the `zelig` class, type: `?zelig` within a interactive Zelig-session.

```
# Simulate Parameters for the gamma Model
param.gamma <- function(obj, num) {

# NOTE: gamma.shape is a method belonging to the
#       GLM class, specifying maximum likelihood
#       estimates of the distribution's shape
#       parameter. It is a list containing two
#       values: 'alpha' and 'SE'

  shape <- gamma.shape(obj)

  # simulate ancillary parameters
  alpha <- rnorm(n=num, mean=shape$alpha, sd=shape$SE)

  # simulate maximum
  sims <- mvrnorm(n = num, mu = coef(obj), Sigma = vcov(obj))

# return results
  list(
       alpha = alpha,       # [1]
       simulations  = sims, # [2]
                            # ...

                            # [3]
       linkinv = function (x) 1/x
       )
}
```

The above code does the following:

**[1]** Specify the ancillary parameters, typically referred to as the greek letter $\alpha$. In the above example, `alpha` is the *shape* of the model's underlying gamma distribution.

**[2]** Specify the parameter simulations, typically referred to as the greek letter $\beta$, to be used in the `qi` function.

**[3]** Specify the inverse-link function [2], used to compute *expected values* and a variety of other *quantities of interest*, once samples are extracted from the model's statistical distribution.

[2] The "inverse-link" function is also commonly referred to as the "mean" function. Typically, this function specifies the relationship between linear predictors and the mean of a distribution function. As a result, it is only used in describing *generalized linear models*

### 1.5.3 Summary and More Information `param` Methods

The `param` method's basic purpose is to describe the statistical and systematic variables of the Zelig model's underlying distribution. Defining this method is an important step towards simplifying the `sim` method. That is, by specifying features of the model - coefficients, systematic components, inverse link functions, etc. - and simulating specific parameters, the `sim` method can focus entirely on simulating *quantities of interest*.

## 1.6 qi: Simulating Quantities of Interest

The `qi` function of any Zelig model simulates *quantities of interest* using the fitted statistical model, taken from the `zelig2` function, and the simulated parameters, taken from the `param` function. As a result, the `qi` function is the most important component of a Zelig model.

### 1.6.1 The `qi` Function Signature

While the implementation of the `qi` function can differ greatly from one model to another, the signature always remains the same and closely parallels the signature of the `sim` function.

```
qi.logit <- function(obj, x=NULL, x1=NULL, y=NULL, param=NULL)
```

### 1.6.2 The `qi` Function Return Values

Similar to the return values of both the `zelig2` and `param` function, the `qi` function takes an list of key-value pairs as a return value. The keys, however, follow a much simpler convention, and a single rule: the key (left-side of the equal sign) is a *quoted* character-string naming the *quantity of interest* and the value (right-side of the equal sign) are the actual simulations.

The following is a short example:

```
list(
     "Expected Value"  = ev,
     "Predicted Value" = pv
     )
```

where `ev` and `pv` are respectively simulations of the model's *expected values* and *predicted values*.

### 1.6.3 Coding Conventions for the `qi` Function

While the following is unnecessary, it provides a few simple guidelines to simplifying and improving readability of a model's `qi` function:

- Divide repetitive work amongst other functions. For example, if you simulate an *expected value* for both the `x` and `x1`, it is better to write a `.compute.ev` function and simply call it twice

- Always compute an *expected values* and *predicted values* independently and before writing code to create *first differences*, *risk ratios*, and *average treatment effects*

- Write code for *average treatment effects* only after all the other code has been debugged and completed

### 1.6.4 A Simplified Example

The following is a simplified example of the `qi` function for the logit model. Note that the example is divided into two sections: one specifying the return values and titles of the *quantities of interest* (see Section 1.6.4) and one computing the simulated *expected values* of the model (see Section 1.6.4).

**`qi.logit` Function**

```
#' simulate quantities of interest for the logit models
qi.logit <- function(obj, x=NULL, x1=NULL, y=NULL, num=1000,
                     param=NULL) {
  # [1]
  ev1 <- .compute.ev(obj, x, num, param)
  ev2 <- .compute.ev(obj, x1, num, param)

  # [2]
  list(
       "Expected Values: E(Y|X)"  = ev1,
       "Expected Values (for X1)" = ev2,

  # [3]
       "First Differences: E(Y|X1) - E(Y|X)" = ev2 - ev1
       )
}
```

**`.compute.ev` Function**

```
.compute.ev <- function(obj, x=NULL, num=1000, param=NULL) {
  # values of NA are ignored by the summary function
  if (is.null(x))
    return(NA)

  # extract simulations
  coef <- coef(param)

  link.inverse <- linkinv(param)

  eta <- coef %*% t(x)

  # invert link function
  theta <- matrix(link.inverse(eta), nrow = nrow(coef))
  ev <- matrix(theta, ncol=ncol(theta))

  ev
}
```

The above code illustrates a few of the ideas:

**[1]** Compute *quantities of interest* using re-usable functions that express the idea clearly. This both reduces the amount of code necessary to produce the simulations, and improves readability of the source code.

**[2]** Return *quantities of interest* as a list. Note: titles of *quantities of interest* are on the left of the equal signs, while simulated values are on the right.

**[3]** Simulate *first differences* by using two previous computed *quantities of interest*.

**[4]** Define an additional function that simulates *expected values*, rather than placing such code in the actual `qi` method.

In addition, this function two *generic functions* that are defined in the Zelig software suite, and are particularly used with the `param` class:

**coef** Extract the simulations of the parameters. Specifically, this returns the simulations produced in the `param` function

**linkinv** Return the inverse of the link function. Specifically, this returns the inverse-link functions specified in the `param` function

### 1.6.5 Summary and More Information about `qi` Methods

The `qi` function offers a simple template for computing *quantities of interest*. Particularly, if a few a coding conventions are followed, the `qi` function can provide transparent, easy-to-read simulation methods.

## 1.7 Conclusion

The above sections detail the fastest way to develop Zelig models. For the vast majority of applications and external statistical packages, this should suffice. However, at times, more elaborate measures may need to be taken. If this is the case, the API specifications for each particular methods should be read, since a wealth of information has been omitted in order to simplify this tutorial.

For more detailed information, consult the `zelig2`, `param`, and `qi` sections of the Zelig Development manual.

# Chapter 2

# `describe`: Describing a Zelig Model

## 2.1 Introduction

When developing a Zelig model, developers have the ability to specify citation and parameter information directly into their model's code. This allows external API's, such as those provided by the Dataverse[1], to make use of Zelig models without any additional programming on the part of the developer. In this capacity, the `describe` method has two purposes:

1. to give correct citation information for Zelig models
2. to declare the type of data that can be processed by the Zelig model

## 2.2 `describe` Method Signature

The `describe` method takes a single parameter - the dots argument. This does not vary between Zelig models. As an example, the `logit` model's method signature is:

```
describe.logit <- function (...) {
  # ...
}
```

## 2.3 `describe` Method Return Value

The `describe` method's signature is simply a list containing several required key-value pairs:

[1] `http://thedata.org/`

- `author`: a vector of character-strings specifying author names
- `text`: a character-string specifying a longer, more specific title to the Zelig model. For example, the `logit` models `text` field contains the value: "Logistic Regression for Dichotomous Dependent Variables"
- `year`: an integer specifying the year that the Zelig model was originally developed

In addition to these key-value pairs, several optional parameters may be included:

- `category`: a character-string specifying the category of statistical regression to which the given model belongs
- `package`: a list-style object specifying information for integration with the Dataverse[2]

## 2.4 `describe` Notable Features

The `describe` method notably does very little computation. That is, while it allows the developer to specify a large amount of citation information, it does not necessarily compute anything. Typically, this equates to the `describe` method consisting entirely of a return-value.

## 2.5 Example of a `describe` Function

The following sections detail the `describe` method of the `logit` Zelig model.

### 2.5.1 `describe.logit.R`

The following example is an excerpt from Zelig's `logit` model:

```
describe.logit <- function(...)
  list(
       # [1]
       authors = c("Kosuke Imai", "Olivia Lau", "Gary King"),
       # [2]
       year    = 2008,
       # [3]
       text    = "Logistic Regression for Dichotomous Dependent Variables"
       )
```

[2]The Dataverse (`http://thedata.org`) is a platform for extracting, sharing, and storing research data. With the correct implementation of the `describe` method, any Zelig model can interact directly and seamlessly with data stored on the Dataverse.

### 2.5.2 Explanation of `describe.logit.R`

The above code corresponds to the following ideas:

[**1**] Specify that the model had three contributors - Kosuke Imai, Olivia Lau and Gary King

[**2**] Specify the original year of authorship as 2008

[**3**] Specify the complete name of the model, that should be used for citation purposes

### 2.5.3 Results of `describe.logit.R`

The citation resulting from the above code example

```
How to cite this model in Zelig:
  Kosuke Imai, Gary King, and Olivia Lau. 2008.
  "logit: Logistic Regression for Dichotomous Dependent Variables"
  in Kosuke Imai, Gary King, and Olivia Lau,
  "Zelig: Everyone's Statistical Software,"
  http://gking.harvard.edu/zelig
```

## 2.6 Summary and Conclusion of the `describe` Method

The `describe` method provides a simple mechanism for specifying citation information. As a result, this method can be written extremely tersely.

# Chapter 3

# The zelig2 Method API

## 3.1 Introduction

Developers can develop a model, write the model-fitting function, and test it within the Zelig framework without explicit intervention from the Zelig team. This modularity relies on two R programming conventions:

1. **wrappers**, which pass arguments from R functions to other R functions or foreign function calls (such as in C, C++, or Fortran). This step is facilitated by - as will be explained in detail in the upcoming chapter - the `zelig2` function.

2. **classes**, which tell generic functions how to handle objects of a given class. For a statistical model to be compliant with Zelig, the model-fitting function *must* return a classed object.

Zelig implements a unique and simple method for incorporating existing statistical models which lets developers test *within* the Zelig framework *without* any modification of both their own code or the `zelig` function itself. The heart of this procedure is the `zelig2` function, which acts as an interface between the `zelig` function and the existing statistical model. That is, the `zelig2` function maps the user-input from the `zelig` function into input for the existing statistical model's constructor function. Specifically, a Zelig model requires:

1. An existing statistical model, which is invoked through a function call and returns an object

2. A `zelig2` function which maps user-input from the `zelig` function to the existing statistical model

3. A name for the **zelig** model, which can differ from the original name of the statistical model.

## 3.2 zelig2 Method Signature

The `zelig2` method's signature typically differs between Zelig models. This is essential, since statistical models generally have a wide-array of available parameters. To accommodate this, the `zelig2` method's signature can be any legal function declaration that adhere to the following guidelines:

1. The `zelig2` method should be simply named `zelig2`*model*, where *model* is the name of the model, that will be used by zelig to reference it
2. The first parameter *must* be titled `formula`
3. The final parameter *must* be titled `data`
4. The ellipsis parameter *must* exist somewhere between the `formula` and `data` parameters
5. Any parameter necessary for use by the external model should be included in the method's parameters

The following is an example taken from the `logit` model:

```
zelig2logit <- function(formula, weights=NULL, ..., data) {
  # ...
}
```

## 3.3 zelig2 Return Value

The `zelig2` method's return value should be a list. The return value of the `zelig` method has two reserved keywords:

1. `.function`: the name of the external method[1] as a character-string
2. `data`: the `data.frame` used by the external method to compute the statistical fit
3. **all other keywords without a leading dot** specify a parameter to be passed to the external. This

In addition to these parameters, two other other *optional* reserved keywords exist:

1. `.hook`: a character-string specifying a function to run immediately *after* the external function executes

[1] "The external method" can be any model-fitting function which returns a valid `formula` object, when the `formula` method is called with it as a parameter. That is, basically any R object can be used as a fitted model, so long as it as a valid slot that can be considered a formula.

2. `.post`: a character-string specifying a function to run immediately *before* the `param` method executes

For more details on the `.hook` and `.post` reserved keywords, see Zelig's hook API specification.

## 3.4 Notable Features of the `zelig2` Method

The `zelig2` method is designed to closely resemble the function call to the external model. That is, this method's parameters and return value should always closely resemble the allowable parameters of the external model's function. As a result, a good rule of thumb is to include the exact parameters from the model in the `zelig2` method, and only remove those that are irrelevant in the developer's Zelig implementation.

## 3.5 Details in Coding the `zelig2` Method

Typically, the `zelig2` method can be coded in a straightforward manner, needing little additional code aside from its return-value. This may however not be the case for models that contain an atypical style of formula. These types of `formula` include:

- multivariate and multinomial regressions
- regressions containing unique syntax, such as special tags
- regressions that accept lists of formulas

One of the main goals of the Zelig software suite is to unify the language and syntax used in the man disparate statistical models. As a result, Zelig models that fall into the above categories often benefit from the existence of helper functions that convert the Zelig-style formula syntax into that used by the external model. Useful examples of this type of `zelig2` method can be found in both the `mixed.zelig` and `bivariate.zelig` packages.

## 3.6 Example of a `zelig2` Method

The following is an illustrative example taken from the `Zelig` core package and its explanation.

### 3.6.1 zelig2logit.R

```
# [1]
zelig2logit <- function(formula, ..., weights=NULL, data)
  list(
       # [2]
       .function = "glm",

       # [3]
       formula = formula,
       data    = data,
       weights = weights,

       # [4]
       family = binomial(link="logit"),
       model  = FALSE,

       # [5]
       ...
       )
```

### 3.6.2 Explanation of zelig2logit.R

The following correspond to the above example:

**[1]** The method name and parameter list specify two things:

- the name of the `zelig2` method by naming the function `zelig2`*model*, where `model` is the name of the model being developed
- the list of acceptable parameter to pass to this Zelig model

**[2]** Specify the name of the external model

**[3]** Specify parameters that are user-defined. Note that the value of `formula`, `data`, and `weights` vary with user-input, because they are part of the `zelig2` method's signature

**[4]** Specify parameter that do not vary with user-input.

**[5]** Specify that any additional parameters to the `zelig2` method be include in the call to the external model

A `zelig2model` function must always return a list as its return value. The entries of the returned list have the following format:

# Chapter 4

# The param Method API

## 4.1 Introduction

Several general features - sampling distribution, link function, systematic component, ancillary parameters, etc. - comprise statistical models. These features, while vastly differing between any two given specific models, share features that are easily classifiable, and usually necessary in the simulation of *quantities of interest*. That is, all statistical models have similar traits, and can be simulated using similar methods. Using this fact, the *parameters* class provides a set of functions and data-structures to aid in the planning and implementation of the statistical model.

## 4.2 `param` Method Signature

The signature of the `param` method is straightforward and does not vary between differ Zelig models.

```
param.logit <- function (obj, num, ...) {
  # ...
}
```

## 4.3 `param` Return Value

The return value of a `param` method is simply a list containing several entries:

**simulations** A vector or matrix of random draws taken from the model's distribution function. For example, a logit model will take random draws from a Multivariate Normal distribution.

**alpha** A vector specifying parameters to be passed into the distribution function. Values for this range from scaling factors to statistical means.

**fam** An optional parameter. *fam* must be an object of class "family". This allows for the implicit specification of the link and link-inverse function. It is recommend that the developer set either this, the link, or the linkinv parameter explicitly. Setting the family object implicitly defines *link* and *linkinv*.

**link** An optional parameter. *link* must be a function. Setting the link function explicitly is useful for defining arbitrary statistical models. *link* is used primarily to numerically approximate its inverse - a necessary step for simulating *quantities of interest*.

**linkinv** An optional parameter. *linkinv* must be a function. Setting the link's inverse explicitly allows for faster computations than a numerical approximation provides. If the inverse function is known, it is recommended that this function is explicitly defined.

## 4.4 Notable Features of the `param` Method

The `param` method is typically a brief, highly structured method. This is because simulating parameters of a statistical model is frequently straightforward, and reduced to simply computing maximum likelihood estimates based on information from the fitted, external model.

## 4.5 Details in Coding the `param` Method

The `param` method's main purpose is to compute or simulate ancillary parameters from the statistical model. In practice, this step is typically accomplished by sampling a particular distribution or via *maximum-likelihood estimation*.

While the simple method of returning a vector or matrix from a *param* function is extremely simple, it has no method for setting link or link-inverse functions for use within the actual simulation process. That is, it does not provide a clear, easy-to-read method for simulating *quantities of interest*. By returning an indexed list - or a parameters object - the developer can provide clearly labeled and stored link and link-inverse functions, as well as, ancillary parameters.

## 4.6 Example of setting the `fam` attribute

The following are two slightly different `param` methods.

### 4.6.1 `param.logit.R`

```
param.logit <- function(z, x, x1=NULL, num=num)
  list(
       coef  = mvrnorm(n=num, mu=coef(z), Sigma=vcov(z)),
```

```
alpha = NULL,
fam   = binomial(link="logit")
)
```

### 4.6.2 Explanation of `param.logit.R`

The above example shows how link and link-inverse functions (for a "logit" model) can be set using a "family" object. Family objects exist for most statistical models - logit, probit, normal, Gaussian, et cetera - and come preset with values for link and link-inverses. This method does not differ immensely from the simple, vector-only method; however, it allows for the use of several API functions - *link*, *linkinv*, *coef*, *alpha* - that improve the readability and simplicity of the model's implementation.

The *param* function and the *parameters* class offer methods for automating and simplifying a large amount of repetitive and cumbersome code that may come with building the arbitrary statistical model. While both are in principle entirely optional - so long as the *qi* function is well-written - they serve as a means to quickly and elegantly implement Zelig models.

## 4.7 Example of setting the link and linkinv attributes

### 4.7.1 param.poisson.R

```
param.normal.survey <- function(obj, num=1000, ...) {
  # [1]
  df <- obj$result$df.residual
  sig2 <- summary(obj)$dispersion
  alpha <- sqrt(df*sig2/rchisq(num, df=df))


  # [2]
  simulations <- mvrnorm(num, coef(obj), vcov(obj))


  # [3]
  fam <- gaussian()


  # [4]
  list(
       simulations = simulations,
       alpha = alpha,
       fam   = fam
       )
}

}
```

### 4.7.2 Explanation of param.poisson.R

The above example shows how a *parameters* object can be created with by explicitly setting the statistical model's link function. The *linkinv* parameter is purely optional, since Zelig will create a numerical inverse if it is undefined. However, the computation of the inverse is typically slower than non-iterative methods. As a result of this, if the link-inverse is known, it should be set, using the *linkinv* parameter.

The above example can also contain an *alpha* parameter, in order to store important ancillary parameters - mean, standard deviation, gamma-scale, etc. - that would be necessary in the computation of *quantities of interest*.

# Chapter 5

# The qi Method API

## 5.1 Introduction

For any Zelig module, the *qi* function is ultimately the most important piece of code that must be written; it describes the actual process which simulates the *quantities of interest*. Because of the nature of this process - and the gamut of statistical packages and their underlying statistical model - it is rare that the simulation process can be generalized for arbitrary fitted models. Despite this, it is possible to break down the simulation process into smaller steps.

## 5.2 `qi` Method Signature

The *qi* function's signature accepts 4 parameters:

**obj:** An object of type `zelig`. This wraps the fitted model in the slot "result"

**x:** An object of type `setx`. This object is used to compute important coefficients, parameters, and features of the data.frame passed to the function call

**x1:** Also an object of type "*setx*". This object is used in a similar fashion, however its presence allows a variety of *quantities of interest* to be computed. Notably, this is a necessary parameter to compute first-differences

**num:** The number of simulations to compute

**param:** An object of type `param`. This is the resulting object from the `param` function, typically containing a variety of important quantities - `simulations`, the `inverse link function`,

```
qi.your_model_name <- function(z, x=NULL, x1=NULL, num=1000) {
# start typing your code here
# ...
# ...
```

## 5.3 qi Method Return Value

In order for Zelig to process the simulations, they must be returned in one of several formats:

- ```
  list(
        "TITLE OF QI 1" = val1,
        "TITLE OF QI 2" = val2,
        # any number of title-val pairs
        # ...
        "TITLE OF QI N" = val.n
        )
  ```

- ```
  make.qi(
           titles = list(title1, title2),
           stats  = list(val1, val2)
           )
  ```

In the above example,*val1, val2*are data.frames, matrices, or lists representing the simulations of the *quantities of interests*, and *title1, title2* - and any number of titles - are character-strings that will act as human-readable descriptions of the *quantities of interest*. Once results are returned in this format, Zelig will convert the results into a machine-readable format and summarize the simulations into a comprehensible format.

NOTE: Because of its readability, it is suggested that the first method is used when returning *quantities of interest*.

## 5.4 Notable Features of *qi* Function

The typical *qi* function has several basic procedures:

1. *Call the param function*: This is entirely optional but sometimes important for the clarity of your algorithm. This step typically consists of taking random draws from the fitted model's underlying probability distribution.

2. *Compute the Quantity of Interest*: Depending on your model, there are several ways to compute necessary quantities of interest. Typical methods for computing quantities of interest include:

   (a) Using the sample provided by 'param' to generate simulations of the *quantities of interest*

   (b) Using a Maximum-likelihood estimate on the fitted model

3. *Create a list of titles for your Quantities of Interest*:

4. *Generate the Quantity of Interest Object*: Finally, with the computed Quantities of Interest, you must

## 5.5 Details in Coding the qi Method

The function body of *qi* function varies largely from model to model. As a result, it is impossible to create general guidelines to simulate *quantities of interest* - or even determine what the *quantity of interest* is. Typical methods for computing *quantities of interest* include:

- Implementing sampling algorithms based on the underlying fitted model, or
- "Predicting" a large number of values from the fitted model

## 5.6 Simple Example qi function (qi.logit.R)

```
#' simulate quantities of interest for the logit models
qi.logit <- function(z, x=NULL, x1=NULL, y=NULL, num=1000, param=NULL) {

  # compute expected values using the function ".compute.ev"
  ev1 <- .compute.ev(obj, x, num, param)
  ev2 <- .compute.ev(obj, x1, num, param)

  # return simulations of quantities of interest
  list(
       "Expected Values: E(Y|X)"  = ev1,
       "Expected Values (for X1): E(Y|X1)" = ev2,
       "First Differences: E(Y|X1) - E(Y|X)" = ev2 - ev1
       )
}
```

# Chapter 6

# Zelig Style Guide

## 6.1 Mandatory Naming Conventions

When developing a Zelig package, it is important to write code in a style that is consistent with existing code and clearly understandable. The following specifies guidelines that are suggested for the any Zelig package, and strictly enforced for commits to Zelig's main trunk.

### 6.1.1 Function and Method Naming

The Zelig developer writes Functions and Class Methods as verbs in camel-case verbs with leading capital letters. That is, a function name must:

1. `be a verb`
2. `have a leading capital letter`
3. `contain only alphanumeric characters`
4. `use a capital letter to denote the beginning of a new word within the function name`
5. `treat all abbreviations as being lowercase`

For example, if the developer is writing a function that converts results to HTML, then the function should be named: `ResultsToHtml`.

### 6.1.2 Generic Function Naming

Generic functions, by Zelig's convention, are preferably single-word lowercase verbs. If it is impossible to phrase the generic function as a single-word, then it should be named as a verb in camel case with a leading lowercase letter. It is highly recommended that careful consideration be taken when naming generic functions. That is, a generic function name must:

1. be a verb
2. have a leading lowercase letter
3. contain only alphanumeric characters
4. use a capital letter to denote the presence of a space between words
5. treat all abbreviations as being lowercase

For example, if the developer is writing a function that stores class information in a log-file for a variety of datatypes, then the function be named: `store` or `writeToLog`.

### 6.1.3 Class Naming

Class names are preferably single-word, alphanumeric, and camel-case nouns. This convention may be loosely followed. The name of the constructor function and the class name must be identical. That is, a class name must:

1. be a noun
2. contain only alphanumeric characters
3. match their function/constructors name

For example, if the developer writes a class that represents a polygon, then it should be named: `polygon` or `Polygon`. The constructor function should be named to match this. Furthermore, classes should be kept in their own R file which is similarly named.

### 6.1.4 Variables

Variables follow very different conventions that functions, methods, classes, and generics. Few rules govern their naming, except that they are must be all lowercase and descriptive. That is, a variable name must:

- contain only lowercase letters and dots
- be descriptive
- be longer than three letters long, unless it is an iterator

For example,

### 6.1.5 Private Functions and Variables

Functions and variables that are intended to be hidden from the user follow all the rules of their visible counterparts, except they must begin with a leading dot. This is so that the export command will ignore them.

### 6.1.6 Operator Overloading

While in many circumstances operator overloading is a useful tool, it is highly discouraged in Zelig packages. Please store all overloaded operators in a file named `ZELIG_zzz_`*`model_class`*`_operators.R`, where *model_class* is the model's class. Note that, because operators do not begin with an alphabet character, they will be ignored by the standard construction of a `NAMESPACE` file. As a result, operators must be explicitly exported in the `NAMESPACE` file.

## 6.2 Suggested Conventions

### 6.2.1 S3 Object Orientation

While S4 objects offer a stricter style of object-oriented programming, it is still standard to write functions as S3 objects due to better support and consistency with core packages (all of which as S3 objects). Zelig can interface with S4 objects, however the results at times are unpredictable, and might only be suitable for truly savvy R developers.

### 6.2.2 Package Naming

It is suggested that packages depending on the Zelig library have ".zelig" appended to them. This will make clear that the syntax and structure of the package is compatible with that of other Zelig objects.

## 6.3 Synopsis

**function** verb, camel-case, leading capital letter

**method** *same as function*

**generic function** verb, alphanumeric, lowercase leading letter, preferably single word

**variable** noun with lowercase letters, numbers, and dots only

**class object** alphanumeric characters only. name must match constructor name and filename

**hidden variable or function** typical naming conventions but with leading dot

**overloaded operators** stored in `ZELIG_zzz_`*`model_name`*`_operators.R`