

Notes on Apopenia's design

B Klemens

January 30, 2008

This note describes some of the considerations that went into the design of Apopenia. To make sure you don't miss it, it starts with the main new concept—models as objects—and then moves on to the details and environment in which those models live. This note will definitely be of interest to those who are writing new models but may also give casual users of the library a better idea of what to expect.

Models as objects

The goal of probability and statistics, from the perspective of a stats library, is to build simple mathematical models that reflect reality. The `apop_model` is intended to encapsulate that view of the world, in a form that embodies the numerical calculations one would want from a model.

Why a model with a standard form? Why not let users write estimate or log likelihood functions without putting them into a type-imposing parent object?¹

First, a few quick, small benefits. Calling functions can fill in blanks with default methods. No `log_likelihood` function? Then use the log of the `p` function. No `dlog_likelihood`? Then build one via numerical gradients. This facilitates laziness (see below), since a user writing a new model can start with just a `p` function, then add other functions as the need for analytic solutions (rather than numerically computed defaults) arises.

The model is also intended to be an easy means of communication among researchers. If the model is written well, a researcher should have no problem sending it to colleagues to try on their own data. As such, it makes a good atom for users to contribute to Apopenia, like the package concept in most stats packages.

Modularity and filtering But the primary benefit is that it is much easier to write functions that take models as inputs, which is one of the key elements that makes the model feel like a self-encapsulated object. For example, it is easier to swap objects around: you can test the fit of your data against four different models by putting the models in an array and writing a for loop to fit each to the data in turn.

Or perhaps you would like to try a slightly more subtle form of model searching, such as testing several variants of a model, which means applying a transformation on

¹The word 'object' is ill-defined, and I see no benefit in putting great effort into defining it. I generally think of any struct that includes functions as an object, and in this essay, I will use 'object' and 'structure' interchangeably. See also this essay on objects in C².

one model to produce a new model. For example, one could see testing a constraint as a series of filtered models. Here is some pseudocode describing the process:

```
apop_model *base, *free_est, *constr, *constr_est;
base      = *apop_normal;
free_est  = apop_estimate(data, *base);
constr    = apop_model_fix_params(base, ...);
constr_est = apop_estimate(data, *constr);
double test_stat = 2*(free_est->llikelihood - constr_est->llikelihood);
```

Each line (but the last) is a filter, transforming an input model into a new model. Line three transforms a base model with no estimated parameters into one whose parameters, log likelihood, &c are estimated; note well that in the terminology of the system, an estimated and un-estimated model are two distinct models. Line four filters the base model into a model with constrained elements, which is then filtered into a fully-estimated model in line five. Line six produces the likelihood ratio test statistic.

Let us say that you decide that the Normal approximation is not quite right, and you would prefer to use a binomial distribution. Then simply change `base = apop_normal` to `base = apop_binomial` and re-run. [You will also have to change the parameter to which the model is fixed, represented by `...` in the code above.]

Bayesians are already familiar with model transformations or filters, because they are in the habit of updating from a prior model to a posterior model, using a likelihood model as an intermediary.

A tour of the `apop_model` structure There are three inputs we could have: parameters, data, or parameters and data together. From parameters, we can get artificial data—a random draw—or an expected value. From data, we can estimate the parameters. From data and parameters, we can get the probability of the data set, or the log of that probability or the derivative of the log of that probability, et cetera.

And this is what the `apop_model` represents: the various means of going between parameters, data, and probabilities.

Finally, there is a list of model settings. [By ‘parameters’, I mean model parameters like mean and variance; by ‘settings’ I means function parameters like step size or tolerance.] Including the settings for every possible model, such as standard MLEs, simulated annealing, Bayesian updating, et cetera, would make for a (still more) bloated and not-extensible structure. The `more` element is in there just in case, but model authors are encouraged to create a model-specific struct to hold the settings and other model-specific information, and then add it to the model’s `settings` array. Have a look at how the MLE and OLS models set parameters for examples [and perhaps you can recycle `apop_OLS_params` or `apop_MLE_params` for your own model].

Next, the model includes the outputs, including estimated parameters, expected values, and so on. The list is slightly arbitrary, and is based on my (BK) personal experience about what users expect and stats packages deliver. It’s easy to add more via one of the hooks in the prior paragraph, and nothing prevents adding more to the structure in the future.

Why include the outputs with the inputs? The output parameters are often inputs into the next step. You may estimate a distribution fitted to your data, and then make random draws from that parameterized distribution. Having one sometimes-NULL structure for parameters and one for the model itself is stupendously messy and simply requires carrying around two structures instead of one. [I know because I tried this for a while.]

Designated initializers to the rescue! Even though the model structure has so many elements, declaration of a new model is simple, thanks to designated initializers. Designated initializers are not well known, but are a superbly useful part of the C99 standard. They allow a structure to be globally defined at startup in a compact manner. For example, if the appropriate functions are defined, you can define a model as `apop_model newmodel = {"A brand new model", .p=new_probability, .constraint=stay_away_from_zero_fn}`. You can ignore what you don't need and are guaranteed that it will be initialized to zero/NULL (even if the struct is a local variable). By declaring the model as global, the user has a host of models immediately available for use without additional initialization calls. Because designated initializers allow you to only declare the structure elements you want, the `apop_model` structure can throw parsimony to the wind and have as many functions as are needed.

Data

Now that the model-as-object discussion is out of the way, we can move on to the rest of statistical work, which is primarily the manipulation of data sets. In fact, as exciting as I find Apophenia's models to be, I answer most of my questions without using them at all.

The other two structures introduced by Apophenia (after the `apop_model`), are the `apop_data` structure and its support, the `apop_name` structure. The great majority of Apophenia's functions act on either the entire `apop_data` object or on one of its constituents.

To facilitate the exposition, here is the `apop_data` set:

```
typedef struct {
    gsl_vector *vector;
    gsl_matrix *matrix;
    apop_name *names;
    char ***text;
    int textsize[2];
    gsl_vector *weights;
    char title[101];
} apop_data;
```

The data set is, more than anything, a pairing of a `gsl_vector` and a `gsl_matrix`. The pairing is surprisingly versatile. First, it allows some functions like `apop_dot` to operate on either vector or matrix as appropriate. But more importantly, vector-matrix pairs come up often, e.g., the mean vector plus covariance matrix of a

set of parameters, or the vector of one dependent variable plus the matrix of many independent variables in a linear model.

The data/name pair facilitate flexibility by allowing users to not worry much about whether their data is a vector or matrix or what-have-you. The names have obvious benefit, and are to a great extent what distinguishes a data matrix for statistics from the matrices used in pure math. But because the `apop_data` set is as lightweight as it is, you could use it as just a `gsl_matrix` at the cost of only a few bytes wasted in NULL pointers to other elements.

Why not use a do-anything structure like R's data frame, that mixes text, integers, and reals? That would be crossing the line between lightweight and heavyweight objects. We would not be able to use the `gsl_matrix` and `_vector` objects (and all their supporting code), and data would be unevenly spaced in memory, which would lead to an unacceptable, order n or more loss in speed. Also, automatic type-casting (which goes hand in hand with variable-type columns) never works predictably for all data sets, and is often more trouble than it's worth.

Modern C

The library takes a relatively modern approach to C coding, oriented toward facilitating quick statistical queries without hampering those who need to be more careful in management of data and memory.

Reduce, reuse, recycle One motivation for the Apophenia project is that no C library existed on the level of general models and data sets. For a number of reasons, the authors of stats software feel the need to go straight to an *ad hoc* language and an interpreter, and the library underlying the final stats package is not necessarily applicable elsewhere.

In short, Apophenia is intended to provide a vocabulary for statistics (nouns like the above structures and verbs to manipulate them) without imposing a new grammar.

One author has compiled a list of about 2,500 languages³. The wisdom of the ages has shown that every language with pretensions to being the One True Language is eventually superseded by another language with the same pretensions. There are certainly aspects of statistics that would benefit from a specialized syntax, but the end result has been a maze of proprietary walled gardens. We all hoped R would be the One True Package, but it too falls flat in a number of ways, including its lack of a call-by-reference mechanism and the interpreter/Rmalloc's way of slowing down C code.

This contrasts with virtually all other types of computing, where libraries have evolved over time, and new libraries build upon older libraries. Apophenia is an attempt to get back to that, which is why the `apop_data` set includes `gsl_matrix` and `gsl_vector` elements to do all the real work.

So it's back to C, but we don't have to write C the way K&R wrote it on their 64kb computers. We enjoy the fast-and-easy coding style typical of stats packages and shell languages, and there's no reason why we can't use that style in C.

³<http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>

Facilitate laziness First, default values are oriented toward the lazy. MLEs without a delta, step size, et cetera, assume reasonable defaults, and Apophenia's default is modestly verbose unless users demand quiet. This makes it easy for novices to start, those who want quick results to get their results, and those who want control to specify what they need.

There are a many functions that basically overlap, such as pairs of the form `apop_data_dosomething` and `apop_matrix_dosomething`. Textbooks on proper coding insist that such redundancy (sorry, non-orthogonality) can be hard to maintain, and needlessly inflates the function index. But it sure does save time wasted converting from one format to another.

Lazy memory usage Pointers are good for you, but their management can be an annoyance. Simply by being in C, Apophenia requires that the user understand pointers. Many coders who know pointers insist that everybody else is too dumb to understand them, but if a user can understand the concept of projecting a data matrix onto an arbitrary basis space, he or she probably has enough grey matter to work out that data and the address of data can be manipulated separately.

However, there are ways to facilitate the process of dealing with memory allocation and deallocation for the lazy.

The GSL is careful to never allocate large spans of memory internally, and does not return large blocks of memory outside of routines with `alloc` in the name. This imposes work on the user, however, because so many functions must be preceded with several lines of memory allocation. Under the *facilitate laziness* rule, Apophenia takes the opposite approach: it freely allocates memory behind your back, although this should be documented for every function that does so, and many functions, such as the estimation routines, return potentially very large blocks of memory. That means that the declaration and the estimation can be on one line, or the returned value can be nested in other functions:

```
apop_model *e = apop_estimate(data, apop_ols);  
(or)  
apop_model_show(apop_estimate(data, apop_ols));
```

Or to give an example that is compact to the point of bad form, here is the code in the constraint test example above, reduced to a single line:

```
double test_stat = 2*(apop_estimate(data, apop_normal)->llikelihood  
- apop_estimate(data, *(apop_model_fix_params(apop_normal, ...)))->llikeliho
```

Copy commands and tests also return a pointer to newly-allocated data, again allowing the user to declare and assign on the same line.

Stats scripts are often short The reader will note that the `apop_model_show` line in the example above produces a memory leak, since the `apop_model` returned by the estimation can't be deallocated. But so what? The typical computer today has over a gigabyte of memory, while many data sets are only a few thousand observations.

If every single allocation in the analysis resulted in a memory leak, the memory lost would still be marginal.

Users who are dealing with the more monolithic analyses can of course take greater effort to manage memory carefully.

Conclusion

Apophenia is intended to be comparable to existing statistics packages, but without an *ad hoc* grammar attached. If you have learned C from any of the hundreds of books and tutorials, then then only new vocabulary you will need to learn are the nouns `apop_model` and `apop_data` and the various verbs that operate on those nouns.

Conceptually, the library does little that is particularly new, except that models are treated as an object that can be transformed in a variety of manners to produce new models. This perspective unifies several different threads of statistics, because the Bayesian, frequentist, maximum likelihood, Monte Carlo, &c approaches can all be put into a standardized, interchangeable form, so models can be transformed and estimated using any and all techniques at once.