Mapping functions

Ben Klemens

21 August 2013

This is another note on useful things Apophenia does; today's entry is about applying a function to every element of a matrix, vector, or data set.

This form helps in thinking about program flow, and makes for clearer writing. It is also a good hook for threading, and for some situations gives you threaded programs almost for free. This is the map/reduce paradigm: get the log likelihood of every observation separately, perhaps on different cores, then sum up the likelihoods to get the total log likelihood for the data set.

In some interpreted languages there's some sort of nontrivial difference between a simple `for` loop over all the rows in a data set and a vectorized function, perhaps because the interpreter has to drop down to C on every line of the `for` loop, but consolidates those calls via the vectorizer. In C, there is exactly zero overhead to calling C functions, so having a function-applying function is thus largely a convenience to help with writing readable code and to make threading easier.

You would first write a *callback function*, a function whose sole intent is to be sent to the map or apply function; for finding log likelihood of a data set, this would be the one-observation log likelihood. Then the mapping function has the task of applying the function to every element of the data set.

Here's a simple example, which fills a 5×5 matrix with random draws by calling the `draw` callback for every element, then produces a new matrix that takes the log of those elements by applying the `ln` callback to every element. The `apply` function changes the input matrix in place; the `map` function uses the callback to map the input matrix to a new output matrix.

```
#include <apop.h>

gsl_rng *r;

void draw(double *in){*in = gsl_rng_uniform(r);}
double ln(double in){return log(in);}

int main(){
    r = apop_rng_alloc(123);
    apop_data *d = apop_data_alloc(5, 5);
    apop_matrix_apply_all(d->matrix, draw);
    gsl_matrix *log_d = apop_matrix_map_all(d->matrix, ln);
    apop_data_show(d);
    printf("\n");
```

```
    apop_matrix_show(log_d);
}
```

This was my first draft of a mapping setup, with several functions (which you can look at in the appropriate segments of the Apophenia documentation[1]). Notice how the random number generator gets passed in to the `draw` function: it's just global to the file. This is universally deemed to be a lazy way to do it, but I never removed it from the Apophenia interface because I still use in my scripts all the time. There isn't really a 'global scope' in C, just scope from one point to the end of the file, and sometimes that's not fatal to readability or maintainability. However, if the global variable isn't read-only (and the RNG isn't), then you can't use this for threading.

A replication example The next extended example is taken from the blog of Nathan Lemoine[2], who used a replication experiment as a timing test between R and Python.

[I'm out of the timing-test business, because I've come to realize that the people who use slow platforms are the people who don't care about speed and evidently never face situations where speed matters. The C code here is still faster than the R code a commenter on that page offered using lm.fit; I imagine that it took me about as long to write the sample code here as it would take an R partisan to write the sped-up R version; and it's up to you to decide whether C fits into your workflow.]

In the setup, create a fake data set, where the first dependent column is an index and the second is a draw from $\mathcal{N}(0, 100)$, and the dependent variable is 2*index + the draw. Then run a regression, producing an expected value of the dependent variable and a residual. Then for the 10,000 replications, build a fake dependent variable by adding the shuffled-up residuals to the same expected values, and re-run the regression. Mark down the 10,000 coefficients on the second term. Report the mean of those coefficients. I'll have more notes on the use of the mapping functions below, and as usual, it helps to read the bottom function first.

- In the non-lazy C tradition, parameters get passed in to the callback via a single `void*` pointer. The textbooks on C programming strongly advise that you generate an *ad hoc* struct for passing in parameters, which I know because I wrote one of those textbooks. Also, because C doesn't allow nested functions (though it's a GNU extension and Apple relies heavily on it), the function has to be outside of the function where the `apop_map` call is placed. Other languages bring the callback and the calling function closer together.
- You can write a callback that takes in an `apop_data` set, in which case the callback will receive a single-row view of the main data set. Therefore, the row index in the callback is always zero. Because the inputs to `apop_data_get` and `_set` and `_ptr` all default to zero, that means you can just omit the `.row=0` part.
- The `apop_map` and `apop_map_sum` functions use the named-input mechanism to determine what sort of mapping you want to do: the name specifies whether the callback takes in a `double` or a `gsl_vector`, a `void *` parameter, the index of the row being worked on. See the documentation for details,

[1] `http://apophenia.info/group__mapply.html`

[2] `http://climateecology.wordpress.com/2013/08/19/r-vs-python-speed-comparison-for-bootstrapping/`

```
#include <gsl/gsl_permute_vector.h>
#include <apop.h>

typedef struct {
    gsl_rng *r;
    apop_model *model;
} rowset_s;

double set_a_row(apop_data *in, void *in_struct, int index){
    rowset_s *s = in_struct;
    double draw;
    apop_draw(&draw, s->r, s->model);
    apop_data_fill(in, 2*index + draw, index, draw);
    return 0;
}


typedef struct {
    apop_data *data;
    gsl_vector *resid, *yhat;
} rep_s;

double one_replication(apop_data *ignore, void *in, int index){
    gsl_rng *r = apop_rng_alloc(index);
    rep_s *rs = in;
    gsl_permutation * p = gsl_permutation_alloc (rs->resid->size);
    gsl_permutation_init(p);
    gsl_ran_shuffle(r, p->data, rs->resid->size, sizeof(size_t));

    //Can't write to the input, so build a new one.
    //point to the original matrix of independents,
    //but generate a new dependent var in alt_data->vector.
    apop_data *alt_data = apop_data_alloc();
    alt_data->matrix = rs->data->matrix;
    alt_data->vector = apop_vector_copy(rs->resid);
    gsl_permute_vector(p, alt_data->vector);
    gsl_permutation_free(p);
    gsl_vector_add(alt_data->vector, rs->yhat);

    apop_model *mb = apop_estimate(alt_data, apop_ols);
    double out = apop_data_get(mb->parameters, .row=1, .col=-1);

    //Clean up. Even at this scale, this is optional.
    alt_data->matrix = NULL;
    apop_data_free(alt_data);
    apop_model_free(mb);
    gsl_rng_free(r);
    return out;
}

int main(){
    apop_opts.thread_count = 4;
    gsl_rng *r = apop_rng_alloc(123);
    apop_data *d = apop_data_alloc(101, 3);
    apop_map(d, .fn_rpi=set_a_row,
                    .param=&((rowset_s){.model=apop_model_set_parameters(apop_normal, 0,
                            10), .r=r}));
    apop_model *m = apop_estimate(d, apop_ols);
    apop_model_show(m);
```

but `.fn_rpi` means the function takes in an `apop_data` row, a `void*` parameter, and the index. This use of named inputs makes the documentation look daunting, but seems to work OK in practice, and gives us type-checking.

- The parameter structs can be built in place, using compound literals and designated initializers. I also count this as textbook stuff, for the same reason as above.
- The original author draws without replacement from the residuals. I wouldn't count this as a bootstrap, which is done with replacement, but nomenclature aside, I used the GSL's permutation struct to shuffle the vector of residuals. That adds four lines of code. Does anybody feel a strong need for an `apop_data_shuffle` function?
- The `one_replication` function initializes a new random number generator and permutation struct for every row of data. In fact, all of the inputs are treated as read-only, and only the internal elements and the final return value are changed. It is thus thread-safe.
- Apophenia's threading functions look to the `apop_opts.thread_count` variable for the number of threads they should use. They use POSIX threads (aka pthreads), and just split the full list of elements into evenly-sized chunks, and each thread runs a `for` loop over its chunk. That is, all you have to do to thread code written using `apop_map` and `apop_map_sum` is to set `apop_opts.thread_count`.
- This is demo code; I probably wouldn't thread this in practice. There is a small overhead to threading, and you can see that things that aren't read-only need to be re-produced in every thread. Also, the unthreaded version of the demo I wrote before this one is several lines shorter. I'm open to suggestions on how we could improve the mapping and threading interface as described here.