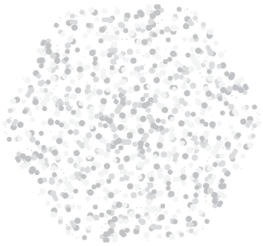
4



MATRICES AND MODELS

> My freedom thus consists in moving about within the narrow frame that I have assigned myself for each one of my undertakings.... Whatever diminishes constraint diminishes strength. The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.
>
> —Stravinsky (1942, p 65)

Recall that the C language provides only the most basic of basics, such as addition and division, and everything else is provided by a library. So before you can do data-oriented mathematics, you will need a library to handle matrices and vectors.

There are many available; this book uses the GNU Scientific Library (*GSL*). The GSL is recommended because it is actively supported and will work on about as many platforms as C itself. Beyond functions useful for statistics, it also includes a few hundred functions useful in engineering and physics, which this book will not mention. The full reference documentation is readily available online or in book form (Gough, 2003). Also, this book co-evolved with the Apophenia library, which builds upon the GSL for more statistics-oriented work.

This chapter goes over the basics of dealing with the GSL's matrices and vectors. Although insisting that matrices and vectors take on a specific, rigid form can be a constraint, it is the constraint that makes productive work possible. The predictable form of the various structures makes it is easy to write functions that allocate and fill them, multiply and invert them, and convert between them.

4.1 THE GSL'S MATRICES AND VECTORS

Quick—what's 14 times 17? Thanks to calculators, we are all a bit rusty on our multiplication, so Listing 4.1 produces a multiplication table.

```
1  #include <apop.h>
2
3  int main(){
4    gsl_matrix *m = gsl_matrix_alloc(20,15);
5      gsl_matrix_set_all(m, 1);
6      for (int i=0; i< m->size1; i++){
7          Apop_matrix_row(m, i, one_row);
8          gsl_vector_scale(one_row, i+1);
9      }
10     for (int i=0; i< m->size2; i++){
11         Apop_matrix_col(m, i, one_col);
12         gsl_vector_scale(one_col, i+1);
13     }
14     apop_matrix_show(m);
15     gsl_matrix_free(m);
16 }
```

Listing 4.1 Allocate a matrix, then multiply each row and each column by a different value to produce a multiplication table. Online source: `multiplicationtable.c`.

- The matrix is allocated in the introductory section, on line four. It is no surprise that it has `alloc` in the name, giving indication that memory is being allocated for the matrix. In this case, the matrix has 20 rows and 15 columns. Row always comes first, then Column, just like the order in Roman Catholic, Randy Choirboy, or RC Cola.
- Line five is the first matrix-level operation: set every element in the matrix to one.
- The rest of the file works one row or column at a time. The first loop, from lines six to nine, begins with the `Apop_matrix_row` macro to pull a single row, which it puts into a vector named `one_row`.
- Given the vector `one_row`, line eight multiplies every element by `i+1`. When this happens again by columns on line 12, we have a multiplication table.
- Line 14 displays the constructed matrix to the screen.
- Line 15 frees the matrix.[1] The system automatically frees all matrices at the end of the program. Some consider it good style to free matrices and other allocated memory anyway; others consider freeing at the end of `main` to be a waste of time.

[1] Due to magic discussed below, vectors allocated by `Apop_matrix_row` and `_col` do not really exist and do not need to be freed.

Naming conventions Every function in the GSL library will begin with `gsl_`, and the first argument of all of these functions will be the object to be acted upon. Most GSL functions that affect a matrix will begin with `gsl_matrix_` and most that operate on vectors begin with `gsl_vector_`. The other libraries used in this book stick to such a standard as well: 100% of Apophenia's functions begin with `apop_` and a great majority of them begin with a data type such as `apop_data_` or `apop_model_`, and GLib's functions all begin with `g_`*`object`*: `g_tree_`, `g_list_`, et cetera.[2]

This custom is important because C is a general-purpose language, and the designers of any one library have no idea what other libraries authors may be calling in the same program. If two libraries both have a function named `data_alloc`, then one will break.

C's library-loaded matrix and vector operations are clearly more verbose and redundant than comparable operations in languages that are purpose-built for matrix manipulation. But C's syntax does provide a few advantages—notably that it is verbose and redundant. As per the discussion of debugging strategy on page 46, spacing out the operations can make debugging numerical algorithms less painful. When there is a type name in the function name, there is one more clue in the function call itself whether you are using the function correctly.

The authors of the Mathematica package chose not to use abbreviations; here is their answer to the question of why, which applies here as well:

> The answer... is consistency. There is a general convention ... that all function names are spelled out as full English words, unless there is a standard mathematical abbreviation for them. The great advantage of this scheme is that it is *predictable*. Once you know what a function does, you will usually be able to guess exactly what its name is. If the names were abbreviated, you would always have to remember which shortening of the standard English words was used. (Wolfram, 2003, p 35)

The naming convention also makes indices very helpful. For example, the index of the GSL's online reference gives a complete list of functions that operate on vectors alphabetized under `gsl_vector_...`, and the index of this book gives a partial list of the most useful functions.

[2]There is one awkward detail to the naming scheme: some functions in the Apophenia library act on `gsl_matrix`es and `gsl_vector`s. Those have names beginning with `apop_matrix` and `apop_vector`, compromising between the library name and the name of the main input.

Q4.1 Don't delay—have a look at the `gsl_vector_...` and `gsl_matrix_...` sections of the index to this book or the GSL's online reference and skim over the sort of operations you can do. The Apophenia package has a number of higher-level operations that are also worth getting to know, so have a look at the `apop_vector_...`, `apop_matrix_...`, and `apop_data_...` sections as well.

If you find the naming scheme to be too verbose, you can write your own wrapper functions that require less typing. For example, you could write a file `my_convenience_fns.c`, which could include:

```
void mset(gsl_matrix *m, int row, int col, double data){
    gsl_matrix_set(m, row, col, data);
}

void vset(gsl_vector *v, int row, double data){
    gsl_vector_set(v, row, data);
}
```

You would also need a header file, `my_convenience_fns.h`:

```
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_vector.h>
void mset(gsl_matrix *m, int row, int col, double data);
void vset(gsl_vector *v, int row, double data);

#define VECTOR_ALLOC(vname, length) gsl_vector *vname = gsl_vector_alloc(length);

// For simple functions, you can rename them via #define; see page 212:
#define vget(v, row) gsl_vector_get(v, row)
#define mget(m, row, col) gsl_matrix_get(m, row, col)
```

After throwing an `#include "my_convenience_fns.h"` at the top of your program, you will be able to use your abbreviated syntax such as `vget(v,3)`. It's up to your æsthetic as to whether your code will be more or less legible after you make these changes. But the option is always there: if you find a function's name or form annoying, just write a more pleasant wrapper function for your personal library that hides the annoying parts.

BASIC MATRIX AND VECTOR OPERATIONS The simplest operations on matrices and vectors are element-by-element operations such as adding the elements of one matrix to those of another. The GSL provides the functions you would expect to do such things. Each modifies its first argument.

```
gsl_matrix_add (a,b);              // a_ij ← a_ij + b_ij, ∀ i, j
gsl_matrix_sub (a,b);              // a_ij ← a_ij − b_ij, ∀ i, j
gsl_matrix_mul_elements (a,b);     // a_ij ← a_ij · b_ij, ∀ i, j
gsl_matrix_div_elements (a,b);     // a_ij ← a_ij / b_ij, ∀ i, j
gsl_matrix_scale (a,x);            // a_ij ← a_ij · x, ∀ i, j ∈ ℕ, x ∈ ℝ
gsl_matrix_add_constant (a,x);     // a_ij ← a_ij + x, ∀ i, j ∈ ℕ, x ∈ ℝ

gsl_vector_add (a,b);              // a_i ← a_i + b_i, ∀ i
gsl_vector_sub (a,b);              // a_i ← a_i − b_ij, ∀ i
gsl_vector_mul (a,b);              // a_i ← a_i · b_i, ∀ i
gsl_vector_div (a,b);              // a_i ← a_i / b_i, ∀ i
gsl_vector_scale (a,x);            // a_i ← a_i · x, ∀ i ∈ ℕ, x ∈ ℝ
gsl_vector_add_constant (a,x);     // a_i ← a_i + x, ∀ i ∈ ℕ, x ∈ ℝ
apop_vector_log(a);                // a_i ← ln(a_i), ∀ i
apop_vector_log10(a);              // a_i ← log_10(a_i), ∀ i
apop_vector_exp(a);                // a_i ← e^{a_i}, ∀ i
```

The functions to multiply and divide matrix elements are given slightly lengthier names to minimize the potential that they will be confused with the process of multiplying a matrix with another matrix, $\mathbf{AB}$, or its inverse, $\mathbf{AB}^{-1}$. Those operations require functions with more computational firepower, introduced below.

$\mathbb{Q}_{4.2}$

Rewrite the structured birthday paradox program from page 35 using a `gsl_matrix` instead of the `struct` that it currently uses.

- `alloc` or `calloc` the matrix in `main`; pass it to both functions.
- Replace the `#include` directives to call in `apop.h`.
- Replace everything after the title-printing line in `print_days` with `apop_matrix_show(`*`data_matrix`*`)`.
- Put three `gsl_matrix_set` commands in the `for` loop of `calculate_days` to set the number of people, likelihood of matching the first, and likelihood of any match (as opposed to one minus that likelihood, as in `bdayfns.c`).

Apply and map Beyond the simple operations above, you will no doubt want to transform your data in more creative ways. For example, the function in Listing 4.2 will take in a `double` indicating taxable income and will return US income taxes owed, assuming a head of household with two dependents taking the standard deduction (as of 2006; see Internal Revenue Service (2007)). This function can be applied to a vector of incomes to produce a vector of taxes owed.

```
1  #include <apop.h>
2
3  double calc_taxes(double income){
4      double cutoffs[] = {0, 11200, 42650, 110100, 178350, 349700, INFINITY};
5      double rates[] = {0, 0.10, .15, .25, .28, .33, .35};
6      double tax = 0;
7      int bracket = 1;
8          income −= 7850; //Head of household standard deduction
9          income −= 3400∗3; //exemption: self plus two dependents.
10         while (income > 0){
11             tax += rates[bracket] ∗ GSL_MIN(income, cutoffs[bracket]−cutoffs[bracket−1]);
12             income −= cutoffs[bracket];
13             bracket ++;
14         }
15         return tax;
16 }
17
18 int main(){
19         apop_db_open("data−census.db");
20         strncpy(apop_opts.db_name_column, "geo_name", 100);
21         apop_data ∗d = apop_query_to_data("select geo_name, Household_median_in as income\
22                                     from income where sumlevel = '040'\
23                                     order by household_median_in desc");
24         Apop_col_t(d, "income", income_vector);
25         d−>vector = apop_vector_map(income_vector, calc_taxes);
26         apop_name_add(d−>names, "tax owed", 'v');
27         apop_data_show(d);
28 }
```

Listing 4.2 Read in the median income for each US state and find the taxes a family at the median would owe. Online source: `taxes.c`.

- Lines 24–27 of Listing 4.2 demonstrate the use of the `apop_data` structure, and will be explained in detail below. For now, it suffices to know that line 24 produces a `gsl_vector` named `income_vector`, holding the median household income for each state.
- The bulk of the program is the specification of the tax rates in the `calc_taxes` function. In the exercise on page 192, you will plot this function.
- The program does not bother to find out the length of the arrays declared in lines four and five. The `cutoffs` array has a final value that guarantees that the `while` loop on lines 10–14 will exit at some point. Similarly, you can always add a final value like `NULL` or `NAN`[3] to the end of a list and rewrite your `for` loop's header to `for (int i=0; data[i] != NAN; i++)`. This means you have to remember to put the sentinel value at the end of the list, but do not need to remember to fix a counter every time you fix the array.

[3] `NAN` is read as not-a-number, and will be introduced introduced on page 135.

You could write a `for` loop to apply the `calc_tax` function to each element of the income vector in turn. But the `apop_vector_map` function will do this for you. Let $c()$ be the `calc_taxes` function, and $\mathbf{i}$ be the `income_vector`; then the call to `apop_vector_map` on line 25 returns $c(\mathbf{i})$, which is then assigned to the vector element of the data set, `d->vector`. Line 27 displays the output.

Threading

Even low-end laptops ship with processors that are capable of simultaneously operating on two or more stacks of frames, so the `map` and `apply` functions can split their work among multiple processors. Set `apop_opts.thread_count` to the desired number of threads (probably the number of processor *cores* in your system), and these functions apportion work to processors appropriately.

When threading, be careful writing to global variables: if a thousand threads could be modifying a global variable in any order, the outcome is likely undefined. When writing functions for threading, your best bet is to take all variables that were not passed in explicitly as read-only.

But `apop_vector_map` is just the beginning: Apophenia provides a small family of functions to map and apply a function to a data set. The full index of functions is relegated to the manual pages, but here is a list of examples to give you an idea.

- You saw that `apop_vector_map(income_vector, calc_taxes)` will take in a `gsl_vector` and returns another vector. Or, `apop_vector_apply(income_vector, calc_taxes)` would replace every element of `income_vector` with `calc_taxes(`*`element`*`)`.
- One often sees functions with a header like `double log_likelihood(gsl_vector indata)`, which takes in a data vector and returns a log likelihood. Then if every row of *`dataset`* is a vector representing a separate observation, then `apop_matrix_map(`*`dataset`*`, log_likelihood)` would return the vector of log likelihoods of each observation.
- Functions with `..._map_..._sum`, like `apop_matrix_map_all_sum`, will return the sum of $f(\text{item})$ for every item in the matrix or vector. For example, `apop_matrix_map_all_sum(`*`m`*`, gsl_isnan)` will return the total number of elements of *`m`* that are `NAN`. Continuing the log likelihood example from above, `apop_matrix_map_sum(`*`dataset`*`, log_likelihood)` would be the total log likelihood of all rows.
- Another example from the family appeared earlier: Listing 3.4 (page 100) used `apop_matrix_apply` to generate a vector of GDP per capita from a matrix with GDP and population.

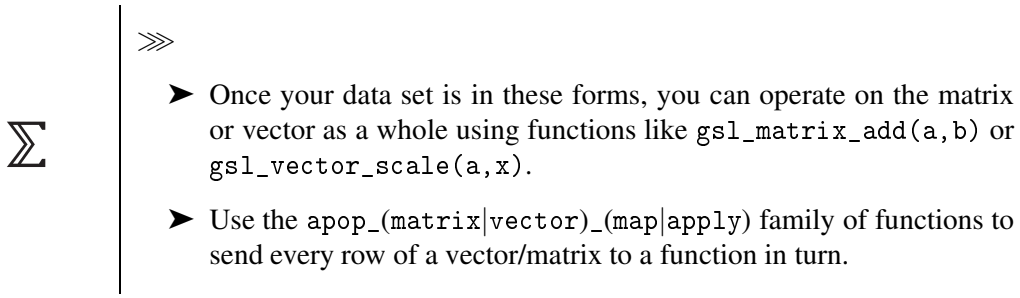
$\sum$

➤ You can express matrices and vectors via `gsl_matrix` and `gsl_vector` structures.

➤ Refer to elements using `gsl_matrix_set` and `gsl_matrix_get` (and similarly for `apop_data` sets and `gsl_vectors`). ⋙

	"v"	"c0"	"c1"	"c2"	"t0"	"t1"	"t2"
"r0"	(0, −1)	(0, 0)	(0, 1)	(0, 2)	(0, 0)	(0, 1)	(0, 2)
"r1"	(1, −1)	(1, 0)	(1, 1)	(1, 2)	(1, 0)	(1, 1)	(1, 2)
"r2"	(2, −1)	(2, 0)	(2, 1)	(2, 2)	(2, 0)	(2, 2)	(2, 2)

Figure 4.3 The vector is column −1 of the matrix, while the text gets its own numbering system. Row names are shared by all three elements.



- Once your data set is in these forms, you can operate on the matrix or vector as a whole using functions like `gsl_matrix_add(a,b)` or `gsl_vector_scale(a,x)`.
- Use the `apop_(matrix|vector)_(map|apply)` family of functions to send every row of a vector/matrix to a function in turn.

4.2 `apop_data` The `apop_data` structure is the joining-together of four data types: the `gsl_vector`, `gsl_matrix`, a table of strings, and an `apop_name` structure.

The conceptual layout is given in Figure 4.3. The vector, columns of the matrix, and columns of text are all named. Also, all rows are named, but there is only one set of row names, because the presumption is that each row of the structure holds information about a single observation.

- Think of the vector as the -1st element of the matrix, and the text elements as having their own addresses.
- There are various means of creating an `apop_data` set, including `apop_query_to_data`, `apop_matrix_to_data`, `apop_vector_to_data`, or creating a blank slate with `apop_data_alloc`; see below.
- For example, Listing 4.2 used `apop_query_to_data` to read the table into an `apop_data` set, and by setting the `apop_opts.db_name_column` option to a column name on line 20, the query set row names for the data. Line 25 sets the `vector` element of the data set, and line 26 adds an element to the `vector` slot of the `names` element of the set.
- You can easily operate on the subelements of the structure. If your *matrix_manipulate* function requires a `gsl_matrix`, but *your_data* is an `apop_data` structure, then you can call *matrix_manipulate*(*your_data*->`matrix`). Similarly,

you can manipulate the names and the table of text data directly. The size of the text data is stored in the `textsize` element. Sample usage:

```
apop_data *set = apop_query_to_text(...);
for (int r=0; r< set->textsize[0]; r++){
    for (int c=0; c< set->textsize[1]; c++)
        printf("%s\t", set->text[r][c]);
    printf("\n");
}
```

- There is no consistency-checking to make sure the number of row names, the `vector->size`, or `matrix->size1` are equal. If you want to put a vector of fifteen elements and a 10×10 matrix in the same structure, and name only the first two columns, you are free to do so. In fact, the typical case is that not all elements are assigned values at all. If *vector_size* is zero, then

```
apop_data *newdata_m = apop_data_alloc(vector_size, n_rows, n_cols);
```

will initialize most elements of `newdata` to `NULL`, but produce a `n_rows`×`n_cols` matrix with an empty set of names. Alternatively, if `n_rows == 0` but `vector_size` is positive, then the vector element is initialized and the matrix set to `NULL`.[4]

Get, set, and point You can use any of the GSL tools above to dissect the `gsl_matrix` element of the `apop_data` struct, and similarly for the `vector` element. In addition, there is a suite of functions for setting and getting an element from an `apop_data` set using the names. Let t be a title and i be a numeric index; then you may refer to the row–column coordinate using the (i, i), (t, i), (i, t), or (t, t) form:

```
apop_data_get(your_data, i, j);
apop_data_get_ti(your_data, "rowname", j);
apop_data_get_it(your_data, i, "colname");
apop_data_get_tt(your_data, "rowname", "colname");
apop_data_set(your_data, i, j, new_value);
apop_data_set_ti(your_data, "rowname", j, new_value);
...
apop_data_ptr(your_data, i, j);
apop_data_ptr_ti(your_data, "rowname", j);
...
```

[4] Seasoned C programmers will recognize such usage as similar to a `union` between a `gsl_vector`, a `gsl_matrix`, and a `char` array, though the `apop_data` set can hold both simultaneously. C++ programmers will observe that the structure allows a form of polymorphism, because you can write one function that takes an `apop_data` as input, but operates on one or both of a `gsl_vector` or a `gsl_matrix`, depending on which is not `NULL` in the input.

- The `apop_data_ptr...` form returns a pointer to the given data point, which you may read from, write to, increment, et cetera. It mimics the `gsl_matrix_ptr` and `gsl_vector_ptr` functions, which do the same thing for their respective data structures.
- As above, you can think about the vector as the -1st element of the matrix, so for example, `apop_data_set_ti(your_data, "rowname", -1)` will operates on the `apop_data` structure's vector rather than the matrix. This facilitates forms like `for (int i=-1; i< data->matrix->size2; i++)`, that runs across an entire row, including both vector and matrix.
- These functions use case-insensitive regular-expression matching to find the right name, so you can even be imprecise in your column request. Appendix B discusses regular expressions in greater detail; for now it suffices to know that you can be approximate about the name: `"p.val.*"` will match `P value`, `p-val` and `p.values`.

For an example, flip back to `ttest.c`, listed on page 111. Line ten showed the full output of the t test, which was a list of named elements, meaning that the output used the set's rownames and `vector` elements. Line eleven pulled a single named element from the vector.

※ *Forming partitioned matrices* You can copy the entire data set, stack two data matrices one on top of the other (stack rows), stack two data matrices one to the right of the other (stack columns), or stack two data vectors:

```
apop_data *newcopy = apop_data_copy(oldset);
apop_data *newcopy_tall = apop_data_stack(oldset_one, oldset_two, 'r');
apop_data *newcopy_wide = apop_data_stack(oldset_one, oldset_two, 'c');
apop_data *newcopy_vector = apop_data_stack(oldset_one, oldset_two, 'v');
```

Again, you are generally better off doing data manipulation in the database. If the tables are in the database instead of `apop_data` sets the vertical and horizontal stacking commands above are equivalent to

```
select * from oldset_one
union
select * from oldset_two

/* and */

select t1.*, t2.*
   from oldset_one t1, oldset_two t2
```

$\mathbb{Q}_{4.3}$

The output of the exercise on page 105 is a table with tattoos, piercings, and political affiliation. Run a Probit regression to determine whether political affiliation affects the count of piercings.

- The function `apop_data_to_dummies` will produce a new data matrix with a column for all but the first category.
- Stack that matrix to the right of the original table.
- Send the augmented table to the `apop_probit.estimate` function. The output for the categorical variables indicates the effect relative to the omitted category.
- Encapsulate the routine in a function: using the code you just wrote, put together a function that takes in data and a text column name or number and returns an augmented data set with dummy variables.

$\sum$

- ➤ The `apop_data` structure combines a vector, matrix, text array, and names for all of these elements.
- ➤ You can pull named items from a data set (such as an estimation output) using `apop_data_get_ti` and family.

4.3 SHUNTING DATA

Igor Stravinsky, who advocated constraints at the head of this chapter, also points out that "Rigidity that slightly yields, like Justice swayed by mercy, is all the beauty of earth."[5] None of function to this point would make any sense if they did not operate on a specific structure like the `gsl_matrix` or `gsl_vector`, but coding is much easier when there is the flexibility of easily switching among the various constrained forms. To that end, this section presents suggestions for converting among the various data formats used in this book. It is not an exciting read (to say the least); you may prefer to take this section as a reference for use as necessary.

Table 4.4 provides the key to the method most appropriate for each given conversion. From/to pairs marked with a dot are left as an exercise for the reader; none are particularly difficult, but may require going through another format; for example, you can go from a `double[ ]` to an `apop_data` set via `double[ ]` $\Rightarrow$ `gsl_matrix` $\Rightarrow$ `apop_data`. As will be proven below, it is only two steps from any format to any other.

[5] Stravinsky (1942), p 54, citing GK Chesterton, "The furrows," in *Alarms and discursions*.

From \ To	Text file	Db table	double[ ]	gsl_vector	gsl_matrix	apop_data
Text file	C	F	·	·	·	F
Db table	·	Q	·	Q	Q	Q
double[ ]	·	·	C	F	F	·
gsl_vector	P	P	F	C	F	F
gsl_matrix	P	P	F	V	C	F
apop_data	P	P	F	S	S	C

Table 4.4 A key to methods of conversion.

※ *Copying structures* The computer can very quickly copy blocks without bothering to comprehend what that data contains; the function to do this is `memmove`, which is a safe variant of `memcpy`. For example, borrowing the `complex` structure from Chapter 2:

```
complex first = {.real = 3, .imaginary = -1};
complex second;
memmove(&second, &first, sizeof(complex));
```

The computer will go to the location of `first` and blindly copy what it finds to the location of `second`, up to the size of one `complex` struct. Since `first` and `second` now have identical data, their constituent parts are guaranteed to also be identical.[6]

But there is one small caveat: if one element of the `struct` is a pointer, then it is the pointer that is copied, not the data itself (which is elsewhere in memory). For example, the `gsl_vector` includes a `data` pointer, so using `memmove` would result in two identical structs that both point to the same data. If you want this, use a view, as per Method V below; if you want a copy, then you need to `memmove` both the base `gsl_vector` and the `data` array. This sets the stage for the series of functions below with `memcpy` in the name that are modeled on C's basic `memmove`/`memcpy` functions but handle internal pointers correctly.

Method C: Copying The `gsl_..._memcpy` functions assume that the destination to which you are copying has already been allocated; this allows you to reuse the same space and otherwise carefully oversee memory. The

[6] How to remember the order of arguments: computer scientists think in terms of data flowing from left to right: in C, `dest = source`; in R, `dest <- source`; in pseudocode, `dest` ← `source`. Similarly, most copying functions have the data flow from end of line to beginning: `memmove(dest, source)`.

`apop_..._copy` functions allocate and copy in one step, so you can declare and copy on the same line, and more easily embed a copy into a filtering operation.

```
//Text file ⇒ Text file
//Just use the system's file copy command. The apop_system function acts like
//the standard C system command, but accepts printf-style arguments:
    apop_system("cp %s %s", from_file_name, to_file_name);
//gsl_vector ⇒ gsl_vector
    gsl_vector *copy = gsl_vector_alloc(original->size);
    gsl_vector_memcpy(copy, original);
    gsl_vector *copy2 = apop_vector_copy(original);
//double[ ] ⇒ double[ ]
//Let original_size be the length of the original array.[7]
    double *copy1 = malloc(sizeof(double) * original_size);
    memmove(copy1, original, sizeof(double) * original_size);
    double copy2[original_size];
    memmove(&copy2, original, sizeof(original));
//gsl_matrix ⇒ gsl_matrix
    gsl_matrix *copy = gsl_matrix_alloc(original->size1, original->size2);
    gsl_matrix_memcpy(copy, original);
    gsl_matrix *copy2 = apop_matrix_copy(original);
//apop_data ⇒ apop_data
    apop_data *copy1 = apop_data_alloc(original->vector->size, original->matrix->size1,
          original->matrix->size2);
    apop_data_memcpy(copy1, original);
    apop_data *copy2 = apop_data_copy(original);
```

Method F: Function call These are functions designed to convert one format to another.

There are two ways to express a matrix of `double`s. The analog to using a pointer is to declare a list of pointers-to-pointers, and the analog to an automatically allocated array is to use double-subscripts:

```
double **method_one = malloc(sizeof(double*)*size_1);
for (int i=0; i< size_1; i++)
    method_one[i] = malloc(sizeof(double) * size_2);
double method_two[size_1][size_2] = {{2,3,4},{5,6,7}};
```

The first method is rather inconvenient. The second method seems convenient, because it lets you allocate the matrix at once. But due to minutiæ that will not be

[7]The `sizeof` function is not just for types: you can also send an array or other element to `sizeof`. If `original` is an array of 100 `double`s, then `sizeof(original)=100*sizeof(double)`, while `sizeof(*original)=sizeof(double)`, and so you could use `sizeof(original)` as the third argument for `memmove`. However, this is incredibly error prone, because this is one of the few places in C where you could send either an object or a pointer to an object to the same function without a warning or error. In cases with modest complexity, the difference between an array and its first element can be easy to confuse and hard to debug.

discussed here (see Kernighan & Ritchie (1988, p 113)), that method is too much of a hassle to be worth anything.

Instead, declare your data as a single line, listing the entire first row, then the second, et cetera, with no intervening brackets. Then, use the `apop_line...` functions to convert to a matrix. For another example, see page 9.

```
//text ⇒ db table
//The first number states whether the file has row names; the second
//whether it has column names. Finally, if no colnames are present,
//you can provide them in the last argument as a char **
    apop_text_to_db("original.txt", "tablename", 0 , 1, NULL);
//text ⇒ apop_data
    apop_data *copyd = apop_text_to_data("original.txt", 0 , 1);
//double[ ][ ] ⇒ gsl_vector,gsl_matrix
    double original[] = {{2,3,4}, {5,6,7}};
    gsl_vector *copv = apop_array_to_vector(original, original_size);
    gsl_matrix *copm = apop_array_to_matrix(original, original_size1, original_size2);

//double[ ] ⇒ gsl_matrix
    double original[] = {2,3,4,5,6,7};
    int orig_vsize = 0, orig_size1 = 2, orig_size2 = 3;
    gsl_matrix *copym = apop_line_to_matrix(original, orig_size1, orig_size2);
//double[ ] ⇒ apop_data
    apop_data *copyd = apop_line_to_data(original, orig_vsize, orig_size1, orig_size2);

//gsl_vector ⇒ double[ ]
    double *copyd = apop_vector_to_array(original_vec);
//gsl_vector ⇒ n × 1 gsl_matrix
    gsl_matrix *copym = apop_vector_to_matrix(original_vec);
//gsl_vector, gsl_matrix ⇒ apop_data
    apop_data *copydv = apop_vector_to_data(original_vec);
    apop_data *copydm = apop_matrix_to_data(original_matrix);
```

Method P: Printing Apophenia's printing functions are actually four-in-one functions: you can dump your data to either the screen, a file, a database, or a system pipe [see Appendix B for an overview of pipes]. Early in putting together an analysis, you will want to print all of your results to screen, and then later, you will want to save temporary results to the database, and then next month, a colleague will ask for a text file of the output; you can make all of these major changes in output by changing one character in your code.

The four choices for the `apop_opts.output_type` variable are

```
apop_opts.output_type = 's'; //default: print to screen.
apop_opts.output_type = 'f'; //print to file.
apop_opts.output_type = 'd'; //store in a database table.
apop_opts.output_type = 'p'; //write to the pipe in apop_opts.output_pipe.
```

- The screen output will generally be human-readable, meaning different column sizes and other notes and conveniences for you at the terminal to understand what is going on. The file output will generally be oriented toward allowing a machine to read the output, meaning stricter formatting.
- The second argument to the output functions is a string. Output to screen or pipe ignores this; if outputting to file, this is the file name; if writing to the database, then this will be the table name.[8]

```
//gsl_vector, gsl_matrix, apop_data ⇒ text file
    apop_opts.output_type = 't'
    apop_vector_print(original_vector, "text_file_copy");
    apop_matrix_print(original_matrix, "text_file_copy");
    apop_data_print(original_data, "text_file_copy");
//gsl_vector, gsl_matrix, apop_data ⇒ db table
    apop_opts.output_type = 'd'
    apop_vector_print(original_vector, "db_copy");
    apop_matrix_print(original_matrix, "db_copy");
    apop_data_print(original_data, "db_copy");
```

Method Q: Querying The only way to get data out of a database is to query it out.

```
//db table ⇒ db table
    apop_query("create table copy as \
    select * from original");
//db table ⇒ double, gsl_vector, gsl_matrix, or apop_data
    double d = apop_query_to_float("select value from original");
    gsl_vector *v = apop_query_to_vector("select * from original");
    gsl_matrix *m = apop_query_to_matrix("select * from original");
    apop_data *d = apop_query_to_data("select * from original");
```

Method S: Subelements Sometimes, even a function is just overkill; you can just pull a subelement from the main data item.

Notice, by the way, that the `data` subelement of a `gsl_vector` can not necessarily be copied directly to a `double[ ]`—the *stride* may be wrong; see Section 4.6 for details. Instead, use the copying functions from Method F above.

```
//apop_data ⇒ gsl_matrix, gsl_vector
    my_data_set -> matrix
    my_data_set -> vector
```

[8] File names tend to have periods in them, but periods in table names produce difficulties. When printing to a database, the file name thus has its dots stripped: `out.put.csv` becomes the table name `out_put`.

Method V: Views Pointers make it reasonably easy and natural to look at subsets of a matrix. Do you want a matrix that represents $\mathbf{X}$ with the first row lopped off? Then just set up a matrix whose `data` pointer points to the second row. Since the new matrix is pointing to the same data as the original, any changes will affect both matrices, which is often what you want; if not, then you can copy the submatrix's data to a new location.

However, it is not quite as easy as just finding the second row and pointing to it, since a `gsl_matrix` includes information about your data (i.e., *metadata*), such as the number of rows and columns. Thus, there are a few macros to help you pull a row, column, or submatrix from a larger matrix. For example, say that `m` is a `gsl_matrix*`, then

```
Apop_matrix_row(m, 3, row_v);
Apop_matrix_col(m, 5, col_v);
Apop_submatrix(m, 2, 4, 6, 8, submatrix);
```

will produce a `gsl_vector*` named `row_v` holding the third row, another named `col_v` holding the fifth column, and a 6×8 `gsl_matrix*` named `submatrix` whose (0, 0)th element is at (2, 4) in the original.

For an `apop_data` set, we have the names at our disposal, and so you could use either `Apop_row(m, 3, row_v)` and `Apop_col(m, 5, col_v)` to pull the given vectors from the matrix element of an `apop_data` structure using row/column number, or `Apop_row_t(m, "fourth row", row_v)` and `Apop_col_t(m, "sixth column", col_v)` to pull these rows and columns by their titles.

The macros work a bit of magic: they internally declare an automatically-allocated `gsl_matrix` or `vector` with the requisite metadata, and then declare a pointer with the name you selected, that can be used like any other pointer to a matrix or vector. However, because these macros used only automatically allocated memory, you do not need to free the matrix or vector generated by the macro. Thus, they provide a quick, disposable view of a portion of the matrix.[9] If you need a more permanent record, them copy the view to a regular vector or matrix using any of the methods from prior pages (e.g., `gsl_vector *permanent_copy = apop_vector_copy(temp_view);`).

[9]These macros are based on GSL functions that are slightly less convenient. For example:

```
gsl_vector v = gsl_matrix_col(a_matrix, 4).vector;
apop_vector_show(&v);
```

If the macro seems to be misbehaving, as macros sometimes do, you can fall back on this form.

4.4 LINEAR ALGEBRA

Say that we have a transition matrix, showing whether the system can go from a row state to a column state. For example, Figure 4.4 was such a transition matrix, showing which formats can be converted to which other formats.

Omitting the labels and marking each transition with a one and each dot in Figure 4.4 with a zero, we get the following transition matrix:

```
1 1 0 0 0 1
0 1 0 1 1 1
0 0 1 1 1 0
1 1 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 1
```

Listing 4.5 shows a brief program to read the data set from a text file, take the dot product of `t` with itself, and display the result.

```
#include <apop.h>
int main(){
    apop_data *t = apop_text_to_data("data-markov", 0, 0);
    apop_data *out = apop_dot(t, t, 0, 0);
    apop_data_show(out);
}
```

Listing 4.5 Two transitions along a transition matrix. Online source: `markov.c`.

Before discussing the syntax of `apop_dot` in detail, here is the program's output:

```
    2 3 1 2 2 3
    3 4 3 4 4 4
    2 2 3 3 3 2
    4 5 4 5 5 5
    4 5 4 5 5 5
    4 5 4 5 5 5
```

This tells us, for example, that there are three ways to transition from the first state to the second in two steps (you can verify that they are: $1 \Rightarrow 1 \Rightarrow 2$, $1 \Rightarrow 2 \Rightarrow 2$, and $1 \Rightarrow 6 \Rightarrow 2$).

The `apop_dot` function takes up to four arguments: two `apop_data` structures, and one flag for each matrix indicating what to do with it (`'t'`=transpose the matrix, `'v'`=use the vector element, `0`=use the matrix as-is). For example, if `X` is a matrix, then

```
apop_dot(X, X, 't', 0);
```

will find $\mathbf{X}'\mathbf{X}$: the function takes the dot product of `X` with itself, and the first version is transposed and the second is not.

- If a data set has a `matrix` component, then it will be used for the dot product, and if the `matrix` element is `NULL` then the `vector` component is used.
- There should be exactly as many transposition flags as matrices. If the first element is a vector, it is always taken to be a row; if the second element is a vector, it is always a column. In both cases, if the other element is a matrix, you will need one flag to indicate whether to use the `apop_data` set's `vector` element (`'v'`), use the transposed matrix (`'t'`), or use the matrix as written (any other character).[10]
- If both elements are vectors, then you are probably better off just using `gsl_blas_ddot`, below, but if you use `apop_dot`, the output will be an `apop_data` set that has a vector element of length one.

$\mathbb{Q}_{4.4}$ The quadratic form $\mathbf{X}'\mathbf{Y}\mathbf{X}$ appears very frequently in statistical work. Write a function with the header `apop_data *quadratic_form(apop_data *x, apop_data *y);` that takes two `gsl_matrix`es and returns the quadratic form as above. Be sure to check that `y` is square and has the same dimension as `x->size1`.

Vector · vector Given two vectors $\mathbf{x}$ and $\mathbf{y}$, `gsl_blas_ddot` returns $x_1y_1 + x_2y_2 + \cdots + x_ny_n$. Rather than outputting the value of $\mathbf{x}\cdot\mathbf{y}$ as the function's return value, it takes the location of a `double`, and places the output there. E.g., if `x` and `y` are `gsl_vector*`s, use

```
double dotproduct;
gsl_blas_ddot (x, y, &dotproduct);
```

[10] Why do you have to tell a computer whether to transpose or not? Some feel that if you send a `1` to indicate transposition when you meant `0` (or vice versa), the system should be able to determine this. Say that you have a 1×10 vector that you will multiply against three data sets, where the first is 8×10, the second is 15×10, and the third is 10×10. You write a simple `for` loop:

```
for(int i=0; i<3; i++)
      out[i] = apop_dot(data[i], v, 1);
```

At `i=0`, a 'smart' system realizes that you committed a faux pas: an 8×10 matrix dot a 10×1 column vector works without transposition. So it corrects you without telling you, and does the same with `data[1]`. With `data[2]`, the transposition works, since there are both ten rows and ten columns. So `out[0]` and `out[1]` are correct and `out[2]` is not. Good luck catching and debugging that.

Write a table displaying the sum of squares $1^2 + 2^2 + 3^2 + \cdots + n^2$ for $n = 1$ through 10.

- Write a function that takes in n and
  - allocates a `gsl_vector*` of size n,
  - fills the vector with $1, \ldots, n$,
  - calculates and returns $\mathbf{v} \cdot \mathbf{v}$ using `gsl_blas_ddot`,
  - and finally frees `v`.
- Write a loop for $n = 1$ through 10 that calls the above function and then prints n and the returned value.
- Verify your work, by printing $n(n+1)(2n+1)/6$ alongside your calculation of the sum of squares up to n.

Q4.5

An example: Cook's distance Cook's distance is an estimate of how much each data point affects a regression (Cook, 1977). The formula is

$$C_i = \frac{\sum_j (\hat{y}_j^{\mathrm{r}} - \hat{y}_j^{\mathrm{ri}})^2}{p \cdot MSE}, \tag{4.4.1}$$

where p is the number of parameters, MSE is mean squared error for the overall regression, $\hat{y}_j^{\mathrm{r}}$ is the jth element of the predicted value of $\hat{\mathbf{y}}$ based on the overall regression, and $\hat{y}_j^{\mathrm{ri}}$ is the jth element of the predicted value of $\hat{\mathbf{y}}$ based on a regression excluding data point i. That is, to find Cook's distance for 3,000 data points, we would need to do a separate regression on 3,000 data sets, each excluding a different data point. The formula simply quantifies whether the predictions made by the main regression change significantly when excluding a given data point.

The procedure provides us a good opportunity to do some matrix-shunting and linear algebra, since we will need functions to produce the subsets, functions to calculate $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$, and to find the squared differences and MSE.

The first function is in Listing 4.6. It produces the series of data sets, each with one row missing. The function is named after the *jackknife* procedure, which uses the same delete-one loop for calculating covariances or correcting bias.[11]

- Lines 9–10 use a submatrix to produce a view of the main matrix starting at the

[11] The Jackknife is not discussed in this book; see the online documentation for Apophenia's `apop_jackknife_cov`.

```
1  #include <apop.h>
2
3  typedef double (*math_fn)(apop_data *);
4
5  gsl_vector *jack_iteration(gsl_matrix *m, math_fn do_math){
6    int height = m->size1;
7    gsl_vector *out = gsl_vector_alloc(height);
8    apop_data *reduced = apop_data_alloc(0, height - 1, m->size2);
9      APOP_SUBMATRIX(m, 1, 0, height - 1, m->size2, mv);
10     gsl_matrix_memcpy(reduced->matrix, mv);
11     for (int i=0; i< height; i++){
12           gsl_vector_set(out, i, do_math(reduced));
13           if (i < height - 1){
14                 APOP_MATRIX_ROW(m, i, onerow);
15                 gsl_matrix_set_row(reduced->matrix, i, onerow);
16           }
17     }
18     return out;
19 }
```

Listing 4.6 Iteratively produce `in->size1` submatrices, each with one omitted row of data. Online source: `jackiteration.c`.

position (1,0), and with size (`m->size1 - 1, m->size2`)—that is, the original matrix with the first row missing—and then uses `gsl_matrix_memcpy` to copy that to a new matrix.

- The `for` loop then repeats the view-and-copy procedure row by row. It begins with row zero, which was omitted before, and overwrites row zero in the copy, aka row one of the original. It then copies over original row one, overwriting the copy of original row two, and so on to the end of the matrix.
- Line 12 calls the function which was sent in as an argument. See page 190 for notes on writing functions that take functions as inputs, including the meaning of the `typedef` on line 3.

Now that we have the matrix-shunting out of the way, Listing 4.7 provides additional functions to do the linear alegbra.

- The `sum_squared_diff` function calculates $\sum_i (L_i - R_i)^2$. The first line finds $L - R$, and the second line applies the function `gsl_pow_2` to each element of $L - R$ (that is, it squares each element) and returns the post-squaring sum.[12]
- The `project` function is taking the dot product $\mathbf{y}_{\text{est}} = \mathbf{X}\boldsymbol{\beta}$. By giving this single

[12]The GSL provides efficient power calculators from `gsl_pow_2` up to `gsl_pow_9`, and the catch-all function `gsl_pow_int(value, exponent)`, that will raise `value` to any integer exponent in a more efficient manner than the general-purpose `pow`.

```
#include <apop.h>

typedef double (*math_fn)(apop_data *);
gsl_vector *jack_iteration(gsl_matrix *, math_fn);
apop_data *ols_data;
gsl_vector * predicted;
double p_dot_mse;

double sum_squared_diff(gsl_vector *left, gsl_vector *right){
    gsl_vector_sub(left, right); //destroys the left vector
    return apop_vector_map_sum(left, gsl_pow_2);
}

gsl_vector *project(apop_data *d, apop_model *m){
    return apop_dot(d, m->parameters, 0, 'v')->vector;
}

double cook_math(apop_data *reduced){
    apop_model *r = apop_estimate(reduced, apop_ols);
    double out =sum_squared_diff(project(ols_data, r), predicted)/p_dot_mse;
    apop_model_free(r);
    return out;
}

gsl_vector *cooks_distance(apop_model *in){
  apop_data *c = apop_data_copy(in->data);
  apop_ols.prep(in->data, in);
  ols_data = in->data;
  predicted = project(in->data, in);
  p_dot_mse = c->matrix->size2 * sum_squared_diff(in->data->vector, predicted);
  return jack_iteration(c->matrix, cook_math);
}

int main(){
    apop_data *dataset = apop_text_to_data("data-regressme", 0, 1);
    apop_model *est = apop_estimate(dataset, apop_ols);
    printf("plot '-'\n");
    strcpy(apop_opts.output_delimiter, "\n");
    apop_vector_show(cooks_distance(est));
}
```

Listing 4.7 Calcluate the Cook's distance, by running 3,200 regressions. Compile with `jackiteration.c`. Online source: `cooks.c`.

line a function of its own, we hide some of the details and get a self-documenting indication of the code's intent.

- The `cook_math` function calculates Equation 4.4.1 for each value of `i`. It is not called directly, but is passed to `jack_iteration`, which will apply the function to

each of its 3,200 submatrices.

- The `cooks_distance` function produces two copies of the data set: an untainted copy `c`, and a regression-style version with the dependent variable in the `vector` element of the data set, and the first column of the `matrix` all ones.
- After `main` calls `cooks_distance`, which calls the various linear algebra procedures and `jack_iteration`, which calls `cook_math` for each submatrix, we have a list of the Cook's distance for every point in the set, which we can use to search for outliers.
- The `main` function produces Gnuplot-ready output, so run this using, e.g., `./cook | gnuplot -persist`. Some researchers prefer to sort the data points before plotting; i.e., try sending the output vector to `gsl_vector_sort` before plotting.

Add some bad data points to the `data-regressme` file, like `1|1|1`, to simulate outliers or erroneous data. Does the Cook's distance of the bad data stand out as especially large?

MATRIX INVERSION AND EQUATION SOLVING

Inverting a matrix requires significantly more computation than the element-by-element operations above. But here in the modern day, it is not such a big deal: my old laptop will invert a $1,000 \times 1,000$ matrix in about ten seconds, and does the inversion step for the typical OLS regression, around a 10×10 matrix at the most, in well under the blink of an eye.

Apophenia provides functions to find determinants and inverses (via the GSL and BLAS's triangular decomposition functions), named `apop_matrix_inverse`, `apop_matrix_determinant`, and for both at once, `apop_det_and_inv`. Examples for using this function are located throughout the book; e.g., see the calculation of OLS coefficients on page 280.

Sometimes, you do not have to bother with inversion. For example, we often write the OLS parameters as $\boldsymbol{\beta} = (\mathbf{X}'\mathbf{X})^{-1}(\mathbf{X}'\mathbf{Y})$, but you could implement this as solving $(\mathbf{X}'\mathbf{X})\boldsymbol{\beta} = \mathbf{X}'\mathbf{Y}$, which involves no inversion. If `xpx` is the matrix $\mathbf{X}'\mathbf{X}$ and `xpy` is the vector $\mathbf{X}'\mathbf{Y}$, then `gsl_linalg_HH_solve(xpx, xpy, betav)` will return the vector $\boldsymbol{\beta}$.

- ➤ `apop_data` · `apop_data`: `apop_dot`.
- ➤ Vector · vector: `gsl_blas_ddot`.
- ➤ Inversion: `apop_matrix_inverse`, `apop_matrix_determinant`, or `apop_det_and_inv`.

4.5 NUMBERS Floating-point numbers can take several special values, the most important of which are `INFINITY`, `-INFINITY`, and `NAN`.[13] Data-oriented readers will mostly be interested in `NAN` (read: not a number), which is an appropriate way to represent missing data. Listing 4.8 shows you the necessary vocabulary. All four `if` statements will be true, and print their associated statements.

```
#include <math.h> //NaN handlers
#include <stdio.h> //printf

int main(){
  double missing_data = NAN;
  double big_number = INFINITY;
  double negative_big_number = -INFINITY;
    if (isnan(missing_data))
        printf("missing_data is missing a data point.\n");
    if (isfinite(big_number)== 0)
        printf("big_number is not finite.\n");
    if (isfinite(missing_data)== 0)
        printf("missing_data isn't finite either.\n");
    if (isinf(negative_big_number)== -1)
        printf("negative_big_number is negative infinity.\n");
}
```

Listing 4.8 Some functions to test for infinity or NaNs. Online source: `notanumber.c`.

Because floating-point numbers can take these values, division by zero won't crash your program. Assigning `double d = 1.0/0.0` will result in `d == INFINITY`, and `d = 0.0/0.0` will result in `d` being set to `NAN`. However, integers have none of these luxuries: try `int i = 1/0` and you will get something in the way of `Arithmetic exception (core dumped)`.

Comparison to an `NAN` value always fails:

```
double blank = NAN;
blank == NAN;    // This evaluates to false.
blank == blank;  // This evaluates to false. (!)
isnan(blank);    // Returns 1: the correct way to check for an NaN value.
```

[13] Pedantic note on standards: These values are defined in the C99 standard (§7.12) only on machines that support the IEEE 754/IEC 60559 floating-point standards, but since those standards are from 1985 and 1989, respectively, they may be taken as given: to the best of my knowledge, all current hardware supports `INFINITY` and `NAN`. Recall that `gcc` requires `-std=gnu99` for C99 features; otherwise, the GSL provides `GSL_POSINF`, `GSL_NEGINF`, and `GSL_NAN` that work in non-C99 and non-IEEE 754 systems.

	GSL constant	approx.		GSL constant	approx.
e	`M_E`	2.71828	π	`M_PI`	3.14159
$\log_2 e$	`M_LOG2E`	1.44270	$\pi/2$	`M_PI_2`	1.57080
$\log_{10} e$	`M_LOG10E`	0.43429	$\pi/4$	`M_PI_4`	0.78540
$\ln 2$	`M_LN2`	0.69315	$1/\pi$	`M_1_PI`	0.31831
$\ln 10$	`M_LN10`	2.30259	$2/\pi$	`M_2_PI`	0.63662
$\sqrt{2}$	`M_SQRT2`	1.41421	$\sqrt{\pi}$	`M_SQRTPI`	1.77245
$\sqrt{1/2} = 1/\sqrt{2}$	`M_SQRT1_2`	0.70711	$2/\sqrt{\pi}$	`M_2_SQRTPI`	1.12838
$\sqrt{3}$	`M_SQRT3`	1.73205	$\ln \pi$	`M_LNPI`	1.14473
Euler constant (γ)	`M_EULER`	0.57722			

Table 4.9 The GSL defines a number of useful constants.

※ *Predefined constants* There are a number of useful constants that are defined by the GSL (via preprocessor `#defines`); they are listed in Table 4.9.[14] It is generally better form to use these constants, because they are more descriptive than raw decimal values, and they are defined to greater precision than you will want to type in yourself.

PRECISION As with any finite means of writing real numbers, there is a roundoff error to the computer's internal representation of numbers. The computer basically stores non-integer numbers using scientific notation. For those who have forgotten this notation, π is written as 3.14159×10^0, or `3.14159e0`, and 100π as 3.14159×10^2, or `3.14159e2`. Numbers always have exactly one digit before the decimal point, and the exponent is chosen to ensure that this is the case.

Your computer works in binary, so floating-point numbers (of type `float` and `double`) are of the form $d \times 2^n$, where d is a string of ones and zeros and n is an exponent.[15]

The *scale* of a number is its overall magnitude, and is expressed by the exponent n. The floating-point system can express a wide range of scales with equal ease: it is as easy to express three picometers (3e$-$12) as three million kilometers (3e9). The *precision* is how many significant digits of information are held in d: 3.14e$-$12 and 3.14e9 both have three significant decimal digits of information.

There is a fixed space for d, and when that space is exceeded, n is adjusted to suit, but that change probably means a loss in precision for d. To give a small base-ten

[14] The GSL gets most of these constants from BSD and UNIX, so you may be able to find them even when the GSL is not available. The exceptions are `M_SQRT3` and `M_SQRTPI`, which are GSL-specific.

[15] This oversimplifies some details that are basically irrelevant for users. For example, the first digit of d is always one, so the computer normally doesn't bother storing it.

i	2^i	2^{-i}
100	1.26765e+30	7.88861e−31
200	1.60694e+60	6.22302e−61
300	2.03704e+90	4.90909e−91
400	2.58225e+120	3.87259e−121
500	3.27339e+150	3.05494e−151
600	4.14952e+180	2.40992e−181
700	5.26014e+210	1.90109e−211
800	6.66801e+240	1.4997e−241
900	8.45271e+270	1.18305e−271
1000	1.07151e+301	9.33264e−302
1100	inf	0
1200	inf	0

Table 4.10 Multiplying together large columns of numbers will eventually fail.

example, say that the space for d is only three digits; then $5.89\text{e}0 \times 892\text{e}0 = 525\text{e}1$, though $5.89 \times 892 = 5,254$. The final four was truncated to zero to fit d into its given space.

Precision can easily be lost in this manner, and once lost can never be regained. One general rule of thumb implied by this is that if you are writing a precision-sensitive function to act on a `float`, use `double` variables internally, and if you are taking in `doubles`, use `long double` internally.

The loss of precision becomes especially acute when multiplying together a long list of numbers. This will be discussed further in the chapter on maximum likelihood estimation (page 330), because the likelihood function involves exactly such multiplication. Say that we have a column of a thousand values each around a half. Then the product of the thousand elements is about 2^{-1000}, which strains what a `double` can represent. Table 4.10 shows a table of powers of two as represented by a `double`. For $i > 1,000$—a modest number of data points—a `double` throws in the towel and calls $2^i = \infty$ and $2^{-i} = 0$. These are referred to as an *overflow error* and *underflow error*, respectively.

For those who would like to try this at home, Listing 4.11 shows the code used to produce this table, and also repeats the experiment using a `long double`, which doesn't give up until over 16,000 doublings and halvings.

```
#include <math.h>
#include <stdio.h>

int main(){
    printf("Powers of two held in a double:\n");
    for(int i=0; i< 1400; i+=100)
        printf("%i\t %g \t %g\n", i, ldexp(1,i), ldexp(1,-i));
    printf("Powers of two held in a long double:\n");
    for(int i=0; i< 18000; i+=1000)
        printf("%i\t %Lg \t %Lg\n", i, ldexpl(1,i), ldexpl(1,-i));
}
```

Listing 4.11 Find the computer's representation of 2^i and 2^{-i} for large i. Online source: `powersoftwo.c`.

- The program uses the `ldexp` family of functions, which manipulate the floating-point representation of a number directly (and are thus probably bad form).
- The `printf` format specifier for the `long double` type is `%Lg`.[16]

The solution to the problem of finding the product of a large number of elements is to calculate the log of the product rather than the product itself; see page 330.

If you need to calculate π to a million decimal points, you will need to find a library that can work with numbers to arbitrary precision. Such libraries typically work by representing all numbers as a data structure listing the ones place, tens place, hundreds place, ..., and then extending the list in either direction as necessary. Another alternative is rational arithmetic, which leaves all numbers in `(int)/(int)` form for as long as possible. Either system will need to provide its own add/subtract/multiply/divide routines to act on its data structures, rather than using C's built-in operators. Unfortunately, the added layer of complexity means that the arithmetic operations that had been fast procedures (often implemented via special-purpose registers on the processor hardware itself) are now a long series of library calls.

So to do math efficiently on large matrices, we are stuck with finite precision, and therefore must not rely too heavily on numbers after around maybe four significant digits. For the purposes of estimating and testing the parameters of a model using real-world data, this is OK. If two numbers differ only after eight significant digits (say, 3.14159265 versus 3.14159268), there is rarely any reason to take these numbers as significantly different. Even if the hypothesis test indicates that they are different, it will be difficult to convince a referee of this.

[16] The `%g` and `%Lg` format specifiers round off large values, so change them to to `%f` and `%Lf` to see the precise value of the calculations without exponential notation.

CONDITIONING Most matrix routines do badly when the determinant is near zero, or when eigenvalues are different orders of magnitude. One way to cause such problems with your own data is to have one column that is of the order of 1×10^{10} and another that is on the order of 1×10^{-10}. In finite-precision arithmetic on two numbers of such wide range, the smaller number is often simply swallowed: `3.14e10 + 5.92e-10 = 3.14e10`.

Thus, try to ensure that each column of the data is approximately of the same order of magnitude before doing calculations. Say that you have a theory that mean fingernail thickness is influenced by a location's population. You could modify the scale when pulling data from the database,

```
select population/1000., nail_thickness*1000.
from health_data;
```

or you could modify it in the `gsl_matrix`:

```
APOP_COL(data, 0, pop)
gsl_vector_scale(pop, 1/1000.);
APOP_COL(data, 1, nails)
gsl_vector_scale(nails, 1000.);
```

These notes about conditioning are not C-specific. Any mathematics package that hopes to work efficiently with large matrices must use finite-precision arithmetic, and will therefore have the same problems with ill-conditioned data matrices. For much more about precision issues and the standard machine representation of numbers, see Goldberg (1991).

COMPARISON Floating-point numbers are exact representations of a real number with probability zero. Simply put, there is a bit of fuzz, so expect every number to be a little bit off from where it should be.

It is normally not a problem that $4 + 1\text{e}{-20} \neq 4$, and such fuzz can be safely ignored, but Polhill *et al.* (2005) point out that the fuzziness of numbers can be a problem for comparisons, and those problems can create odd effects in simulations or agent-based models. After a long series of floating-point operations, a comparison of the form `(t == 0)` will probably not work. For example, Listing 4.12 calculates $1.2 - 3 \cdot 0.4$ using standard IEEE arithmetic, and finds that it is less than zero.[17]

There are labor-intensive solutions to the problem, like always using `long ints`

[17]This example is from a web site affiliated with the authors of the above paper, at `http://www.macaulay.ac.uk/fearlus/floating-point/`.

```
#include <stdio.h>

int main(){
   double t = 1.2;
      t -= 0.4;
      t -= 0.4;
      t -= 0.4;
      if (t<0)
            printf ("By the IEEE floating-point standard, 1.2 - 3*.04 < 0.\n");
}
```

Listing 4.12 The IEEE standard really does imply that $1.2 - 3 \cdot 0.4 < 0$. Online source: `fuzz.c`.

for everything,[18] but the most sensible solution is to just bear in mind that no comparison is precise so, for example, agents should not die when their wealth is zero, but when it is less than maybe $1e{-}6$. Otherwise, the model should be robust to agents who have an iota of negative wealth.

- Floating-point numbers can take on values of `-INFINITY`, `INFINITY`, and `NAN`.
- Multiplying together a column of a thousand numbers will break, so get the log of the product by summing the logs of the column's elements.
- Reporting results based on the fifth significant digit (or so) is spurious.
- Try to keep the scale of your variables within a factor of about a thousand of each other.
- Exact comparisons of floating-point numbers can fail, so do not test `f == 0`, but `fabs(f) < 1e-6` (or so).

4.6 ※ `gsl_matrix` AND `gsl_vector` INTERNALS

First, a warning: the intent of this section is not to show you how to circumvent the GSL's access functions such as `gsl_matrix_get` and `gsl_vector_set`. Doing so is bad form, inviting errors and making code more difficult to read.[19] Instead, these notes will be useful to you for

[18] Recall that `(a/b)*b + a%b` is exactly `a` for `long int a, b`, $b \neq 0$.

[19] If you are really concerned about the overhead from these functions, then `#define GSL_RANGE_CHECK_OFF`, either via that preprocessor directive or adding -DGSL_RANGE_CHECK_OFF to your compilation com-

understanding why the data structures are the way they are, and giving you a handle on what operations are easy and what operations are difficult.

Here is the relevant section of the declaration of the `gsl_matrix` structure:

```
typedef struct {
  size_t size1;
  size_t size2;
  size_t tda;
  double * data;
  [...]
  int owner;
} gsl_matrix;
```

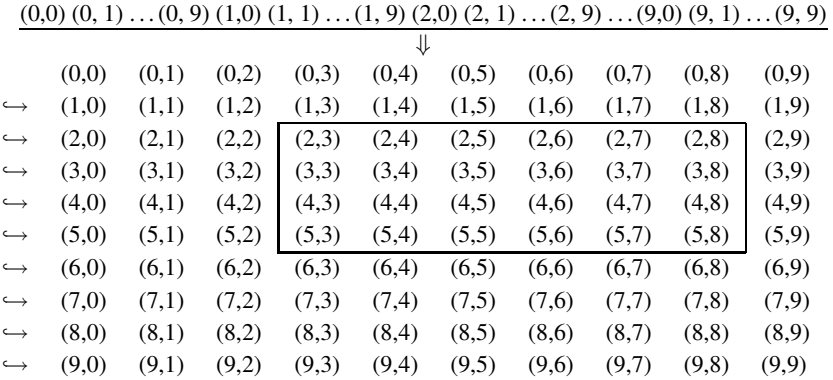


Figure 4.13 Within a submatrix, the (3,4) element is still one step from the (3,3) element, and ten steps from the (2,4) element.

As you know, `size1` and `size2` are simply the count of rows and columns. The `data` pointer is a single pointer to a stream of numbers. Since memory addresses are linear, the top of Figure 4.13 is closer to what is actually in memory: the first row of data, followed immediately afterward by the second row, then the third row, and so on, forming one long row of data. By adding line breaks, we humans can think of this one long row of data as actually being a grid, like the second half of Figure 4.13.

Stepping along the row means simply stepping along by `sizeof(double)` units, and stepping down a matrix column means stepping by `sizeof(double)* size2` steps from the current element. For example, to reach the (3,5) element of a ten by ten matrix, the processor must skip three rows and then skip five items, so it would jump `sizeof(double)*35` steps from the base element.

Modern computers are proactive about data gathering. When they read data from slower types of memory, they also check the neighbors as well. If the code is relatively predictable, the system can gather the next bit of data at the same time as

mand line. Between this and the compiler's optimization routines, the function call will reduce to the appropriate array operation.

it is crunching the current data element. The `gsl_matrix` structure works wonderfully with such a system, because steps are predictable and of fixed size, so the processor has a good chance of correctly guessing what data to put into its faster caches.

Now say that we have a 100×10 matrix, which would have the following information:

```
size1 = 100;
size2 = 10;
tda = 10;
data = [location of (0,0)];
owner =1;
```

With `tda` equal to `size2`, jumping down a column would require a jump of `sizeof(double)*tda`.[20]

If we wanted to pull out the 4×6 submatrix that begins at $(3, 2)$, then the resulting submatrix data would look like this:

```
size1 = 4;
size2 = 6;
tda = 10;
data = [location of (3,2) in the original matrix];
owner =0;
```

We can use this matrix exactly as with the full matrix: For example, to get the third elements in the first row, we would step `sizeof(double) * 2` forward from the base element pointed to by `data`, and to get to the beginning of the next column, we would jump `sizeof(double)*tda`. Thus, the process of pulling a subset of the data merely required finding the first point and writing down arbitrary limits for `size1` and `size2`. No actual data was copied. This is how `gsl_matrix_row`, `APOP_ROW`, `APOP_COL`, `APOP_SUBMATRIX`, and other such routines work.

The `owner` variable now becomes important, because there could be multiple submatrices all pointing to the same data. Since the submatrix is not the owner of its data, the GSL will not allow it to free the data set to which it points.

The `gsl_vector` has a similar structure, including a starting point and a stride to indicate how far to jump to the next element. Thus, taking a row or column subset of a matrix also merely requires writing down the correct coordinates and lengths.

The GSL's structures are good for fast access, because the next element is always

[20] The abbreviation `tda` stands for *trailing dimension of array.* For a `gsl_vector`, the analogous element is the *stride*.

a fixed jump relative to the current element, whether you are traversing by rows or columns. They are exceptionally good for describing submatrices and subvectors, because doing so merely requires writing down new coordinates. They handle a $1 \times 10,000$ matrix just as easily as a $10,000 \times 1$ matrix or a 100×100 matrix.

They are not good for non-contiguous subsets like the first, second, and fifth columns of a data set, since the relative jumps from one column to the next are not identical. Similarly, they are not good for holding various types of data, where some jumps could be `sizeof(int)` and others could be `sizeof(double)`. Also, since space is always allocated for every element, there is no way to efficiently represent sparse matrices.

Systems that deal well with variable-sized jumps have the pros and cons reversed from the above. For example, one solution is to let the overall table be a list of column vectors, where each column has its own type. But traversing along a row could involve jumping all over memory, so common operations like finding the sum for each row becomes a significantly slower operation.

Other designers with different goals have used different means of representing a data matrix, and the `gsl_matrix` is by no means the best for all needs. But it does very well for the goal of allowing the hardware to process rows and columns of homogeneous data with maximal efficiency.

- The GSL's matrix and vector structures are very good for efficient computation, because each element has a fixed size and is a fixed distance from the neighboring elements.
- It is very easy to take contiguous subvectors or submatrices of these structures. Doing so requires copying only a few bits of metadata, but not the data itself.
- There is no simple way to take non-contiguous subsets of `gsl_matrices` or `gsl_vectors`. You will either need to copy the data manually (i.e., using a `for` loop), or do the manipulations in the database before your data is in `gsl_matrix` form.

4.7 MODELS

Recall the one-sentence summary of statistical analysis from the first page of the introduction: estimate the parameters of a model using data. The Apophenia library provides functions and data structures at exactly this level of abstraction, in the form of the `apop_model` and `apop_data` structures and the functions that operate on them.

You have already met the `apop_data` structure, which lent a hand to operations on the matrix algebra layer of abstraction; the remainder of the chapter introduces the `apop_model` structure, which provides similar forms of strength through constraint: it encapsulates model information in a uniform manner, allows models to be used interchangeably in functions that can take any model as an input, and allows sensible defaults to be filled in as necessary.

A great deal of statistical work consists of converting or combining existing models to form new ones. That is, models can be filtered to produce models just as data can be filtered to provide new information. We can read estimation as filtering an un-parameterized model into a parameterized one. Bayesian updating (discussed more thoroughly on page 258) takes in a prior model, a likelihood function, and data, and outputs a new model—which can then be used as the input to another round of filtering when new data comes in.

Another example discussed below, is the imposition of a constraint: begin by estimating a general model, then generate a new model with a constraint imposed on some of the parameters, and re-estimate. The difference in log likelihoods of the constrained and unconstrained models can then be used for hypothesis testing.

The structure of the model struct In the usage of this book, a model intermediates between data and parameters. From there, the model can go in three directions:

i) $\mathbf{X} \Rightarrow \boldsymbol{\beta}$: Given data, estimate parameters.

ii) $\boldsymbol{\beta} \Rightarrow \mathbf{X}$: Given parameters, generate artificial data (e.g., make random draws from the model, or find the expected value).

iii) $(\mathbf{X}, \boldsymbol{\beta}) \Rightarrow p$: Given both data and parameters, estimate their likelihood or probability.

To give a few examples, form (i) is the descriptive problem, such as estimating a covariance or OLS parameters. Monte Carlo methods use form (ii): producing a few million draws from the model given fixed parameters. Bayesian estimation is based on form (iii), describing a posterior probability given both data and parameters, as are the likelihoods in a maximum likelihood estimation.

For many common models, there are `apop_models` already written, including distributions like the Normal, Multivariate Normal, Gamma, Zipf, et cetera, and generalized linear models like OLS, WLS, probit, and logit. Because they are in a standardized form, they can be sent to model-handling functions, and be applied to data in sequence. For example, you can fit the data to a Gamma, a Lognormal, and an Exponential distribution and compare the outcomes (as in the exercise on page 257).

Every model can be estimated via a form such as

```
apop_model *est = apop_estimate(data, apop_normal);
```

Examples of this form appear throughout the book—have a look at the code later in this section, or on pages 133, 289, 352, or 361, for example.

Discussion of the other directions—making random draws of data given parameters and finding likelihoods given data and parameters—will be delayed until the chapters on Monte Carlo methods and maximum likelihood estimation, respectively.

Changing the defaults A complete model includes both the model's functions and the environment in which those functions are evaluated (Gentleman & Ihaka, 2000). The `apop_model` thus includes both the outputs, the functions, and everything one would need to replicate one from the other. For the purposes of Apophenia's model estimations, an $\mathcal{N}(0,1)$ is a separate model from an $\mathcal{N}(1,2)$, and a maximum likelihood model whose optimization step is done via a conjugate gradient method is separate from an otherwise identical model estimated via a simplex algorithm.

But a generic `struct` indended to hold settings for all models faces the complication that different methods of estimation require different settings. The choice of conjugate gradient or simplex algorithm is meaningless for an instrumental variable regression, while a list of instrumental variables makes no sense to a maximum likelihood search.

Apohenia standard `apop_model` struct thus has an open space for attaching different groups of settings as needed. If the model's defaults need tweaking, then you can first add an MLE, OLS, histogram, or other settings group, and then change whatever details need changing within that group. Again, examples of the usage and syntax of this two-step processs abound, both in online documentation and throughout the book, such as on pages 153, 339, or 352.

Writing your own It would be nice if we could specify a model in a single form and leave the computer to work out the best way to implement all three of the directions at the head of this section, but we are rather far from such computational nirvana. Recall the example of OLS from the first pages of Chapter 1. The first form of the model—find the value of β such that $(\mathbf{y} - \mathbf{X}\beta)^2$ is minimized—gave no hint that the correct form in the other direction would be $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$. Other models, such as the probit model elaborated in Chapter 10, begin with similar $\mathbf{X}\beta$-type forms, but have no closed form solution.

Thus, writing down a model for a computable form requires writing down a procedure for one, two, or all three of the above directions, such as an `estimate` function, a `log_likelihood` and `p` (probability) function, and a `draw` function to make random draws. You can fill in those that you can solve in closed form, and can leave Apophenia to fill in computationally-intensive default procedures for the rest.

```
1  #include <apop.h>
2
3  apop_model new_OLS;
4
5  static apop_model *new_ols_estimate(apop_data *d, apop_model *params){
6      APOP_COL(d, 0, v);
7      apop_data *ydata = apop_data_alloc(d->matrix->size1, 0, 0);
8      gsl_vector_memcpy(ydata->vector, v);
9      gsl_vector_set_all(v, 1); //affine: first column is ones.
10     apop_data *xpx = apop_dot(d, d, 't', 0);
11     apop_data *inv = apop_matrix_to_data(apop_matrix_inverse(xpx->matrix));
12     apop_model *out = apop_model_copy(new_OLS);
13     out->data = d;
14     out->parameters = apop_dot(inv, apop_dot(d, ydata, 1), 0);
15     return out;
16 }
17
18 apop_model new_OLS = {.name ="A simple OLS implementation",
19                       .estimate = new_ols_estimate};
20
21 int main(){
22     apop_data *dataset = apop_text_to_data("data-regressme", 0, 1);
23     apop_model *est = apop_estimate(dataset, new_OLS);
24     apop_model_show(est);
25 }
```

Listing 4.14 A new implementation of the OLS model. Online source: `newols.c`.

For example, listing 4.14 shows a new implementation of the OLS model. The math behind OLS is covered in detail on page 274.

- In this case, only the `estimate` function is specified.
- The procedure itself is simply a matter of pulling out the first column of data and replacing it with ones, and calculating $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$.
- Lines 12–14 allocate an `apop_model` and set its `parameter` element to the correct value. Line 13 keeps a pointer to the original data set, which is not used in this short program, but often comes in handy.
- The allocation of the output on line 12 needs the model, but we have not yet declared it. The solution to such circularity is to simply give a declaration of the

model on line 3.

- Given the function, line 18 initializes the model itself, a process which is rather simple thanks to designated initializers (see p 32).
- Although the `main` function is at the bottom of this file, the typical model deserves its own file. By using the `static` keyword in line five, the function name will only be known to this file, so you can name it what you wish without worrying about cluttering up the global name space. But with is a pointer to the function in the model object itself, a routine in another file could use the `apop_new_OLS.estimate(...)` form to call this function (which is what `apop_estimate` on line 23 will do internally).
- Lines 1–3 provide the complete header an external file would need to use the new model, since the structure of the `apop_model` is provided in `apop.h`.

$\mathbb{Q}_{4.7}$

Compare the `new_OLS` model with the `apop_ols` model.

- Modify Listing 4.14 to declare an array of `apop_models`. Declare the first element to be `apop_ols` and the second to be `new_OLS`. [The reverse won't work, because `new_OLS` destroys the input data.]
- Write a `for` loop to fill a second array of pointers-to-models with the estimate from the two models.
- Calculate and display the difference between `estimate[0]->parameters->vector` and `estimate[1]->parameters->vector`.

※ **AN EXAMPLE: NETWORK DATA** The default, for models such as `apop_ols` or `apop_probit`, is that each row of the data is assumed to be one observation, the first column of the data is the dependent variable, and the remaining columns are the independent variable.

For the models that are merely a distribution, the rule that one row equals one observation is not necessary, so the data matrix can have any form: $1 \times 10{,}000$, or $10{,}000 \times 1$, or 100×100. This provides maximum flexibility in how you produce the data.

But for data describing ranks (score of first place, second place, ...) things get more interesting, because such data often appears in multiple forms. For example, say that we have a classroom where every student wrote down the ID number his or her best friend, and we tallied this list of student numbers:
1 1 2 2 2 2 3 4 4 4 6 7 7 7.
First, we would need to count how often each student appeared:

id_no count
2 4
4 3
7 3
1 2
3 1
6 1.

In SQL:

```
select id_no, count(*) as ct
    from surveys
    group by id_no
    order by ct desc
```

If we were talking about city sizes (another favorite for rank-type analysis), we would list the size of the largest city, the second largest, et cetera. The labels are not relevant to the analysis; you would simply send the row of counts for most popular, second most popular, et cetera:
4 3 3 2 1 1.
Each row of the data set would be one classroom like the above, and the column number represents the ranking being tallied.

As mentioned above, you can add groups of settings to a model to tweak its behavior. In the case of the models commonly used for rank analysis, you can signal to the model that it will be getting rank-ordered data. For example:

```
apop_model *rank_version = apop_model_copy(apop_zipf);
Apop_settings_add_group (rank_version, apop_rank, NULL);
apop_model_show(apop_estimate(ranked_draws, rank_version));
```

Alternatively, some data sets are provided with one entry listing the rank for each observation. There would be four 1's, three 2's, three 3's, et cetera:
1 1 1 1 2 2 2 3 3 3 4 4 5 5.
In the city-size example, imagine drawing people uniformly at random from all cities, and then writing down whether each person drawn is from the largest city, the second largest, et cetera. Here, order of the written-down data does not matter. You can pass this data directly to the various estimation routines without adding a group of settings; e.g., `apop_estimate(`*ranked_draws*`, apop_gamma)`.

$\mathbb{Q}_{4.8}$

The `nominee` column in the file `data-classroom` is exactly the sort of data on which one would run a network density analysis.

- Read the data into a database.
- Query the vector of ranks to an `apop_data` set, using a query like the one above.
- Transpose the matrix (*hint*: `gsl_matrix_transpose_memcpy`), because `apop_zipf`'s `estimate` function requires each classroom to be a row, with the nth-ranked in the nth column. This data set includes only one classroom, so you will have only one row of data.
- Call `apop_estimate(`*`yourdata`*`, apop_zipf)`; show the resulting estimate. How well does the Zipf model fit the data?

※ **MLE MODELS** To give some examples from a different style of model, here are some notes on writing models based on a maximum likelihood estimation.

- Write a likelihood function. Its header will look like this:

```
static double apop_new_log_likelihood(gsl_vector *beta, apop_data *d)
```

  Here, `beta` holds the parameters to be maximized and `d` is the fixed parameters—the data. This function will return the value of the log likelihood function at the given parameters and data.
  In some cases, it is more natural to express probabilities in log form, and sometimes in terms of a plain probability; use the one that works best, and most functions will calculate the other as needed.
- Declare the model itself:

```
apop_model new_model = {"The Me distribution", 2, 0, 0,
                .log_likelihood = new_log_likelihood };
```

  - If you are using a probability instead of a log likelihood, hook it into your model with `.p = `*`new_p`*.
  - The three numbers after the name are the size of the parameter structure, using the same format as `apop_data_alloc`: size of vector, rows in matrix, then columns in matrix. If any of these is `-1`, then the `-1` will be replaced with the number of columns in the input data set's matrix (i.e., *`your_`*`-`

data->matrix->size2).[21] This is what you would use for an OLS regression, for example, where there is one parameter per independent variable.

With this little, a call like apop_estimate(*your_data*, new_model) will work, because apop_estimate defaults to doing a maximum likelihood search if there is no explicitly-specified new_model.estimate function.

- For better estimations, write a gradient for the log likelihood function. If you do not provide a closed-form gradient function, then the system will fill in the blank by numerically estimating gradients, which is slower and has less precision. Calculating the closed-form gradient is usually not all that hard anyway, typically requiring just a few derivatives. See Listing 4.17 (or any existing apop_model) for an example showing the details of syntax.

SETTING CONSTRAINTS A constraint could either be imposed because the author of the model declared an arbitrary cutoff ('we can't spend more than $1,000.') or because evaluating the likelihood function fails ($\ln(-1)$). Thus, the system needs to search near the border, without ever going past it, and it needs to be able to arbitrarily impose a constraint on an otherwise unconstrained function.

Apophenia's solution is to add a constraint function that gets checked before the actual function is evaluated. It does two things if the constraint is violated: it nudges the point to be evaluated into the valid area, and it imposes a penalty to be subtracted from the final likelihood, so the system will know it is not yet at an optimum. The unconstrained maximization routines will then have a continuous function to search but will never find an optimum beyond the parameter limits.[22]

To give a concrete example, Listing 4.15 adds to the apop_normal model a constraint function that will ensure that both parameters of a two-dimensional input are greater than given values.

Observe how the constraint function manages all of the requisite steps. First, it checks the constraints and quickly returns zero if none of them binds. Then, if they do bind, it sets the return vector to just inside the constrained region. Finally, it returns the distance (on the *Manhattan metric*) between the input point and the point returned.[23] The unconstrained evaluation system should repeatedly try points closer and closer to the zero-penalty point, and the penalty will continuously decline as we approach that point.

[21] if your model has a more exotic parameter count that needs to be determined at run-time, use the prep method of the apop_model to do the allocation.

[22] This is akin to the common penalty function methods of turning a constrained problem into an unconstrained one, as in Avriel (2003), but the formal technique as commonly explained involves a series of optimizations where the penalty approaches zero as the series progresses. It is hard to get a computer to find the limit of a sequence; the best you could expect would be a series of estimations with decreasing penalties; apop_estimate_restart can help with the process.

[23] If you need to calculate the distance to a point in your own constraint functions, see either apop_vector_distance or apop_vector_grid_distance.

```
1  #include <apop.h>
2
3  double linear_constraint(apop_data * d, apop_model *m){
4      double limit0 = 2.5,
5             limit1 = 0,
6             tolerance = 1e-3; // or try GSL_EPSILON_DOUBLE
7      double beta0 = apop_data_get(m->parameters, 0, -1),
8             beta1 = apop_data_get(m->parameters, 1, -1);
9      if (beta0 > limit0 && beta1 > limit1)
10         return 0;
11     //else create a valid return vector and return a penalty.
12     apop_data_set(m->parameters, 0, -1,GSL_MAX(limit0 + tolerance, beta0));
13     apop_data_set(m->parameters, 1, -1, GSL_MAX(limit1 + tolerance, beta1));
14     return GSL_MAX(limit0 + tolerance - beta0, 0)
15                     + GSL_MAX(limit1 + tolerance - beta1, 0);
16 }
17
18 int main(){
19     apop_model *constrained = apop_model_copy(apop_normal);
20     constrained->estimate = NULL;
21     constrained->constraint = linear_constraint;
22     apop_db_open("data-climate.db");
23     apop_data *dataset = apop_query_to_data("select pcp from precip");
24     apop_model *free = apop_estimate(dataset, apop_normal);
25     apop_model *constr = apop_estimate(dataset, *constrained);
26     apop_model_show(free);
27     apop_model_show(constr);
28     double test_stat = 2 * (free->llikelihood - constr->llikelihood);
29     printf("Reject the null (constraint has no effect) with %g%% confidence\n",
           gsl_cdf_chisq_P(test_stat, 1)*100);
30 }
```

Listing 4.15 An optimization, a constrained optimization, and a likelihood ratio test comparing the two. Online source: `normallr.c`.

In the real world, set linear constraints using the `apop_linear_constraint` function, which takes in a set of contrasts (as in the form of the F test) and does all the requisite work from there. For an example of its use, have a look at the budget constraint in Listing 4.17.

The `main` portion of the program does a likelihood ratio test comparing constrained and unconstrained versions of the Normal distribution.

- Lines 19–21 copy off the basic Normal model, and add the constraint function to the copy. The `estimate` routine for the Normal doesn't use any constraint, so it is invalid in the unconstrained case, so line 20 erases it.
- Lacking an explicit `estimate` routine in the model, line 25 resorts to maximum likelihood estimation (MLE). The MLE routine takes the model's constraint into

account.

- Lines 28–29 are the hypothesis test. Basically, twice the difference in log likelihoods has a χ^2 distribution; page 350 covers the details.

AN EXAMPLE: UTILITY MAXIMIZATION The process of maximizing a function subject to constraints is used extensively outside of statistical applications, such as for economic agents maximizing their welfare or physical systems maximizing their entropy. With little abuse of the optimization routines, you could use them to solve any model involving maximization subject to constraints. This section gives an extended example that numerically solves an Econ 101-style utility maximization problem.

The consumer's utility from a consumption pair (x_1, x_2) is $U = x_1^\alpha x_2^\beta$. Given prices P_1 and P_2 and B dollars in cash, she has a budget constraint that requires $P_1 x_1 + P_2 x_2 \leq B$. Her goal is to maximize utility subject to her budget constraint.

The data (i.e., the `apop_data` set of fixed elements) will have a one-element vector and a 2×2 matrix, structured like this: $\left[\begin{array}{c|cc} \text{budget} & \text{price}_1 & \alpha \\ & \text{price}_2 & \beta \end{array}\right]$.

Once the models are written down, the estimation is one function call, and calculating the marginal values is one more. Overall, the program is overkill for a problem that can be solved via two derivatives, but the same framework can be used for problems with no analytic solutions (such as for consumers with a stochastic utility function or dynamic optimizations with no graceful closed form).

Because the estimation finds the slopes at the optimum, it gives us comparative statics, answering questions about the change in the final decision given a marginal rise in price P_1 or P_2 (or both).

Listing 4.16 shows the model via numerical optimization, and because the model is so simple, Listing 4.17 shows the analytic version of the model.

- The `econ101_estimate` routine just sets some optimization settings and calls `apop_maximum_likelihood`.
- The budget constraint, in turn, is a shell for `apop_linear_constraint`. That function requires a constraint matrix, which will look much like the matrix of equations sent in to the F tests on page 310. In this case, the equations are

$$\begin{bmatrix} -\text{budget} & < & -p_1\beta_1 & - & p_2\beta_2 \\ 0 & < & \beta_1 & & \\ 0 & < & & & \beta_2 \end{bmatrix}.$$

All inequalities are in less-than form, meaning that the first—the cost of goods is

```
#include <apop.h>
apop_model econ_101;

static apop_model * econ101_estimate(apop_data *choice, apop_model *p){
    Apop_settings_add_group(p, apop_mle, p);
    Apop_settings_add(p, apop_mle, tolerance, 1e-4);
    Apop_settings_add(p, apop_mle, step_size, 1e-2);
    return apop_maximum_likelihood(choice, *p);
}

static double budget(apop_data *beta, apop_model* m){
  double price0 = apop_data_get(m->data, 0, 0),
         price1 = apop_data_get(m->data, 1, 0),
         cash = apop_data_get(m->data, 0, -1);
  apop_data *constraint = apop_data_alloc(3, 3, 2);
    apop_data_fill(constraint,
                   -cash, -price0, -price1,
                   0., 1., 0.,
                   0., 0., 1.);
    return apop_linear_constraint(m->parameters->vector, constraint, 0);
}

static double econ101_p(apop_data *d, apop_model *m){
  double alpha = apop_data_get(d, 0, 1),
         beta = apop_data_get(d, 1, 1),
         qty0 = apop_data_get(m->parameters, 0, -1),
         qty1 = apop_data_get(m->parameters, 1, -1);
    return pow(qty0, alpha) * pow(qty1, beta);
}

apop_model econ_101 = {"Max Cobb-Douglass subject to a budget constraint", 2, 0, 0,
    .estimate = econ101_estimate, .p = econ101_p, .constraint= budget};
```

Listing 4.16 An agent maximizes its utility. Online source: `econ101.c`.

less than the budget—had to be negated. However, the next two statemets, β_1 is positive and β_2 is positive, are natural and easy to express in these terms. Converting this system of inequalities into the familiar vector/matrix pair gives

$$\left[\begin{array}{c|cc} -\text{budget} & -p_1 & -p_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right].$$

- The budget consraint as listed has a memory leak that you won't notice at this scale: it re-specifies the constraint every time. For larger projects, you can ensure that `constraint` is only allocated once by declaring it as `static` and initially setting it to `NULL`, then allocating it only if it is not `NULL`.
- The analytic version of the model in Listing 4.17 is a straightforward translation

```
#include <apop.h>
apop_model econ_101_analytic;
apop_model econ_101;

#define fget(r, c) apop_data_get(fixed_params, (r), (c))

static apop_model * econ101_analytic_est(apop_data * fixed_params, apop_model *pin){
  apop_model *est = apop_model_copy(econ_101_analytic);
  double budget = fget(0, -1), p1 = fget(0, 0), p2 = fget(1, 0),
              alpha = fget(0, 1), beta = fget(1, 1);
  double x2 = budget/(alpha/beta + 1)/p2,
         x1 = (budget - p2*x2)/p1;
    est->data = fixed_params;
    est->parameters = apop_data_alloc(2,0,0);
    apop_data_fill(est->parameters, x1, x2);
    est->llikelihood = log(econ_101.p(fixed_params, est));
    return est;
}

static void econ101_analytic_score(apop_data *fixed_params, gsl_vector *gradient,
      apop_model *m){
  double x1 = apop_data_get(m->parameters, 0, -1);
  double x2 = apop_data_get(m->parameters, 1, -1);
  double alpha = fget(0, 1), beta = fget(1, 1);
    gsl_vector_set(gradient, 0, alpha*pow(x1,alpha-1)*pow(x2,beta));
    gsl_vector_set(gradient, 1, beta*pow(x2,beta-1)*pow(x1,alpha));
}

apop_model econ_101_analytic = {"Analytically solve Cobb-Douglass maximization subject
      to a budget constraint",
    .vbase =2, .estimate=econ101_analytic_est, .score = econ101_analytic_score};
```

Listing 4.17 The analytic version. Online source: `econ101.analytic.c`.

of the solution to the constrained optimization. If you are familiar with Lagrange multipliers you should have little difficulty in verifying the equations expressed by the routines. The routines are in the natural slots for estimating parameters and estimating the vector of parameter derivatives.

- The term *score* is defined in Chapter 10, at which point you will notice that its use here is something of an abuse of notation, because the score is defined as the derivative of the log-utility function, while the function here is the derivative of the utility function.
- The preprocessor can sometimes provide quick conveniences; here it abbreviates the long function call to pull parameters to `fget`. Section 6.4 (page 211) gives the details of the preprocessor's use and many caveats.
- The process of wrapping library functions in standardized model routines, and oth-

```
#include <apop.h>
apop_model econ_101, econ_101_analytic;

void est_and_score(apop_model m, apop_data *params){
  gsl_vector *marginals = gsl_vector_alloc(2);
  apop_model *e = apop_estimate(params, m);
    apop_model_show(e);
    printf("\nThe marginal values:\n");
    apop_score(params, marginals, e);
    apop_vector_show(marginals);
    printf("\nThe maximized utility: %g\n", exp(e->llikelihood));
}

int main(){
  double param_array[] = {8.4, 1, 0.4,
                          0, 3, 0.6}; //0 is just a dummy.
  apop_data *params = apop_line_to_data(param_array, 2,2,2);
    sprintf(apop_opts.output_delimiter, "\n");
    est_and_score(econ_101, params);
    est_and_score(econ_101_analytic, params);
}
```

Listing 4.18 Given the models, the `main` program is but a series of calls. Online source: `econ101.main.c`.

erwise putting everything in its place, pays off in the `main` function in Listing 4.18. Notably, the `est_and_score` function can run without knowing anything about model internals, and can thus run on both the closed-form and numeric-search versions of the model. It also displays the maximized utility, because—continuing the metaphor that likelihood=utility—the `econ101` model's maximization returned the log utility in the `llikelihood` element of the output model.

- The `econ101.analytic` model calculates the parameters without calculating utility, but there is no need to write a separate calculation to fill it in—just call the utility calculation from the `econ101` model. Thanks to such re-calling of other models' functions, it is easy to produce variants of existing models.
- The only public parts of `econ101.c` and `econ101.analytic.c` are the models, so we don't have to bother with a header file, and can instead simply declare the models themselves at the top of `econ101.main.c`.
- The model files will be compiled separately, and all linked together, using a Makefile as per Appendix A, with an `OBJECTS` line listing all three `.o` files.

Use one of the models to produce a plot of marginal change in x_0 and x_1 as α expands.

- The `apop_model` aggregates methods of estimating parameters from data, drawing data given parameters, and estimating likelihoods given both.
- Most `apop_models` take in a data structure with an observation on each row and a variable on each column. If the model includes a dependent variable, it should be the first column.
- Given a prepackaged model, you can estimate the parameters of the model by putting your data into an appropriate `apop_data` structure, and then using `apop_estimate(data, your_model)`. This will produce an `apop_model` that you can interrogate or display on screen.
- If closed-form calculations for any of the model elements are available, then by all means write them in to the model. But the various functions that take `apop_models` as input make their best effort to fill in the missing methods. For example, the score function is not mandatory to use gradient-based optimization methods.