Statistics with the GNU Scientific Library

Ben Klemens
klemens@hss.caltech.edu

January 10, 2004

This is a proposal for a book on doing statistical analysis using the GNU Scientific Library for C. It would be a much-needed supplement to any textbook on classical statistics, since every textbook I have ever seen (and I have seen many) focuses entirely on the mathematical issues, and either ignores implementation in code entirely or relegates it to a ten-page appendix. But statistics is all about lengthly computations on large sets of numbers: the mathematical results are useless to somebody who doesn't know which tools will implement those results.

The audience The primary audience for this book are people who have in mind a few mathematical results they want to apply to the data they have on their hard drive, but could find no help on how to do so. These are people who thought they knew statistics until they had to apply it to real data. Personally, I know *a lot* of people who fit this description, and have seen their panic firsthand.

Since stats textbooks are so depressingly lacking in teaching the implementation of the math they cover, this book would make a good supplementary text for a second-semester undergrad or a grad level statistics class.

This book is an animal with slightly different spots from O'Reilly's other publications because its goal is not comprehensive treatment of C or the GSL, but to teach the reader enough C to have a comprehensive ability to do stats. However, it is still a book about technical computing at the level which O'Reilly's books are known for, using the open source tools which O'Reilly has made a name documenting.

The competition I could find no books on stats with C—or anything else that isn't a stats package.

There are many books on doing stats in various stats packages: e.g., SPSS Corp publishes a book on data analysis with SPSS. These books inherently have the limitations of the stats package they teach/sell: notably a far too heavy focus on OLS and research-through-pictures, and little or no mention of techniques based on likelihood functions. Every geek I've spoken to on the topic is frustrated with these limitations, and is looking for an easier way to implement nontrivial statistical tests. That's whom I'm writing this book for.

About me I am currently a Guest Scholar at the Brookings Institution, a Washington think tank. Within that, I'm in the most academic division, the Center on Social and Economic Dynamics, which designs large-scale agent-based models to test policy. They write lots of Java, but have been moving toward C and C^{++}. After my undergrad days, I worked at a small midwestern brokerage firm as their risk control analyst, which was the sort of work I describe in this book: downloading large data sets and doing stats on them.

Education: I have a PhD in Social Sciences from Caltech, with time at Harvard University; and a BA in Econ from the University of Chicago, with time at the London School of Economics.

Works: I am one of those people who likes to write didactic essays to solidify his knowledge, and have a few online. One, 'The story of econometrics' is a favorite among Caltech students and a few of the research assistants here at Brookings. I have also written a few programs for public comsumption, including a game for the Palm Pilot.

The overview Not one to wait for a publisher's permission, I've already started writing; what follows is a large sample of what I have so far. You can think of it as a writing sample and the brief talking outline rolled into one. I've included a fraction of the code that will eventually be in the book, meaning that some code calls functions that aren't included in this overview.

The book will be about 150 pages, and will take about five months to complete. In the interest of keeping focused, I am not writing on the subject of data mining or Bayesian analysis, but could include those chapters on request.

Contents

1 Doing statistics in the modern day

In case it's not obvious to you, we can't do statistical analysis without computers. The mathematical explanation of a statistical procedure is really just pseudo-code, which we can make operational by translating it into real computer code.

I wrote this book to help you make that translation. The first focus is purely mathematical: we need to select techniques bearing in mind that they'll eventually be code. The second focus is about practical coding: your life is short, and you want spend as little of it coding as possible.

Doing math with a computer is unfettering. Instead of using regression techniques designed before computers, we can use techniques built around computing thousand-term likelihood functions, or taking millions of random samples. These techniques originally appeared in the textbooks as just theory, then eventually with a caveat that these techniques are possible but computationally intensive. Now computations are cheap, and we can use these techniques as we would their simplified brethren.

In my own pain-filled experience, the best way to operationalize statistical concepts is using C; the GNU Scientific Library, which facilitates the linear algebra and the Gaussian distribution work; and SQLite, a package which facilitates handling large data sets. This book will cover the basics of these components and the manner in which we can translate from the mathematical language to the language these libraries speak.

I assume that you've had a statistics class already, but may need some reminding here and there, and so I'll include sidebars to remind you of your Greek. I also assume that you're basically computer-literate, so I won't explain to you how to copy files or how to work a text editor.

1.1 The stats we'll cover

As powerful as we like to think modern statistics is, it has a limited set of tricks at its disposal. For the purposes of this book, I will divide them into three broad categories, which are broad enough to cover just about 100% of statistics (in fact, they overlap).

Projections. This includes the overused OLS linear regression (which is a projection of your data onto a line), and the underused factor analysis techniques. These are the techniques we humans use to reduce too many dimensions down to something we can comprehend and even draw a picture

of.

Gaussian distribution tricks. The Central Limit Theorem says that a very wide range of statistics will have a Normal distribution. Square that, and we have a Chi-squared distribution. Take the ratio of two Chi-squared distributions, and we have an F distribution. The sum of a random number of Normally distributed variables will have a Laplace distribution.

Likelihood function tricks. Once we know the probability of an event, probably thanks to a Gaussian distribution trick, we can then write down the likelihood that a given set of parameters would bring about the data we see, and then find the parameters that maximize this likelihood. MLEs have the pleasing property of meeting the Cramer-Rao lower bound, and the ratio of likelihood functions has the pleasant property embodied in the Neyman-Pearson lemma, making them the basis of hypotheses testing.

1.1.1 The goals of statistical analysis

First, let's settle an important fact about statistical analysis: it proves nothing, only persuades. A large part of this is that we are looking for causal stories about the world, but there is no statistical technique (and never will be) which proves causality. Further, there's always a way to rewrite a model or re-draw the data so that things will be different. But, of course, some results are more robust to tweaking than others, and some models are just plain more persuasive than others.

Within the overall goal of persuasion, we can subdivide the goals of statistical analysis into two parts. The first is to just say something interesting about a data set. This is often model-free; for example, you may just want to show that two variables are highly correlated, or that the data can mostly be described by three dimensions. This is often sufficient to support an argument, and if that's the case, then you should go no further in dazzling the reader with your statistical abilities.

The second part of statistical analysis is hypothesis testing, in which we calculate a parameter of the data and then make a claim about that parameter. Having observed in the last paragraph that the correlation coefficient of two variables is large, perhaps we'd like to prove that that correlation coefficient is almost certainly different from zero. More often, we have parameters in a model that we've written, and would like to make claims about those parameters.

The techniques used for the two goals above are entirely different. For

example, there are any of a number of distributions listed in the average stats textbook. Poisson or binomial distributions are used only for describing data culled from the real world. The Chi-squared distribution or the F distribution are used only for testing hypotheses about parameters we've written down ourselves (such as the correlation coefficient, which has an F distribution). Kmenta points out that nothing in nature has a Chi-squared distribution.

Often, your work will shift gears from describing the data (like fitting an OLS regression to the data) to testing a hypothesis (like the claim that the coefficients in your regression are significantly different from zero). The stats textbooks I've read tend to run these goals together at every opportunity; hopefully I'm doing better here, but it's up to you to know exactly which goal you're working on with each line of code or math, and to be certain that it's working toward your overall goal of saying something interesting and persuasive.

1.1.2 General theorems and their special cases

Here's another way to subdivide the theorems which underly statistics, this time into two classes. The first class, including the results about MLEs, applies almost universally, but requires a huge amount of computational power to arrive at a result. The second class consists of special cases of these general results, such as the theorems underlying OLS regressions (which are an application of the general theorems behind the likelihood ratio test); these results impose more assumptions, but are much easier to calculate, so that the results could be used a century before computers were invented.

The second class is what we spend most of our time learning in school, because a few decades ago, this was the only class of results which civilians had the computing power to use. This is when the stats packages that are so prevalent today came to the fore, automating the tedium of applying these special case results. The technology had an immediate influence on how people did research: they applied those darn special cases to everything, and you'd have a hard time finding an issue of an academic journal today that doesn't have at least one OLS regression. Everything in the world has become a linear process.

This book is about using OLS only when it's applicable. OLS was the only viable tool for a hundred years or so, but we're done with using it everywhere. The Central Limit Theorem tells us that yes, errors probably are Normally distributed; and it's often the case that yes, the dependent variable is more

or less a linear or log-linear function of several variables. If such descriptions do no violenece to the reality from which the data was culled, then OLS is the method to use, and using more general techniques will not be any more persuasive. But if these assumptions aren't true, then using OLS is at best unpersuasive, and at worst disingenuous. Before computing power was where it is today, OLS was as persuasive as we could get, and we all just had to accept that. But now it is possible (and as this book hopes to show, even easy) to write down exactly the right likelihood function, and to find exactly the best parameters, instead of settling for the unpersuasive—and often inapplicable—model which requires the least processor power.

1.2 The programming we'll cover

Using C is a mix of low-level and high-level work. You'll be allocating memory, telling the computer exactly where to shunt its electrons. But, thanks to the efforts of tens of thousands of programmers before you, you'll have the benefit of functions which will find the minimum of a function or calculate characteristics of a Gaussian distribution, without having to remember Newton's method or the equation for the Gaussian distribution.

The first few chapters of this book will cover the basics of C itself, in terms of its grammar and how to call those functions which you'll find so valuable. I am not shooting for completeness: just giving you enough that you'll understand the rest of the story. The remainder of the book will be at the higher level, describing how to glue together the functions in the GNU Scientific Library to get your research done.

After you've glued together a few functions the same way a few times, you'll see patterns in your code. After all, every analysis you do will read in data, do some math, and output something. Much of the fun of C comes from writing your own functions to do these things that you know you'll have to do again. You'll write them, so they'll work the way you think, reading data in your favorite format, encapsulating the things that annoy you so you never have to think about them again, and doing the types of analysis that are unique to your field.

1.2.1 Dealing with large data sets

More than anything, the problem with stats packages is that large data sets will be hard to deal with. Those languages which are designed around dealing

with large data sets tend to be even more draconian than C—for example, SAS's data input command is `card`, referring to the punch card it expects you to put in the hopper.

The best programs for large data sets are databases. They are designed from the ground up to do nothing but let you efficiently retrieve what you need from huge amounts of data. Of course, being so purpose-specific, there's no way to do statistics in a database (beyond calculating averages). Fortunately, we're working in C, so we can have the best of all worlds. The method I advocate in this book is to read all of your data into a database, and then query what you need to a matrix, as you need it. This requires learning a new syntax, SQL, which is decidedly neither Beautiful nor Perfect, and adds a level of complication on top of what we're already dealing with. But from my experience, it is very much worth it.

In this book, I will use the SQLite library, which will give you all the database functionality you need. Those of you who are already database gurus, or who are handed data which is already handled by another database engine, will easily be able to adapt the techniques used here, but will need to brush up on the details of how to access your site's database. Since we're using gcc instead of a stats package, you're guaranteed that there's an interface out there that you can download and incorporate into your programs.

1.2.2 Pretty pictures

One thing C is not really good for is drawing pretty pictures. This is not to be belittled, since those pretty pictures can be very persuasive. Consistent with the rest of this book, I will use Gnuplot, a program which is freely available and which will compile on the system you're using right now. Gnuplot is highly automatable, so once you've got a plot you like, you can have your C programs autogenerate them or manipulate them in cute ways, and can send your program to your colleague in Madras, and he'll have no problem reproducing and modifying your graphs.

If you love the way a certain program does its graphics, there's no reason to forsake that. Many stats packages are better than C for doing graphics, and I'll show you how to get the best of both worlds. The basic idea is to process the data in C and then write a text file specifying the data you want to plot. Gnuplot, Sigmaplot, or even Excel will be able to pick up that text file and plot its contents.

1.3 Why C?

There are so many other languages out there in which you could do statistics. Why use C instead of SAS, Stata, SPSS, S-PLUS, GAUSS, GAMS, MatLab, Mupad, Mathematica, Minitab, Limdep, Octave, R, or RATS? Since this is a pervasive question, I'm going to spend a few pages answering it as clearly as possible. Over the course of this, you should also get a better idea of why C is the way it is, and why things that seem annoying on the surface will pay off in the long run.

1.3.1 Reason #1: C will help you do better science.

As noted above, it's no longer OK to use OLS for everything. OLS, with all its assumptions, used to be the only technique that we had the computing power to actually implement. But my four-year old laptop regularly executes feats of computation that were entirely impossible fifty years ago. Similarly with your computer. *So why are we still using theorems written to facilitate computation?* More importantly, why are we using them in cases where their assumptions aren't true?

Unfortunately, the statistics packages are written around the specialized, assumption-heavy theorems, and because people do what the technology facilitates, people who use stats packages are very, very likely to assume OLS is valid. If OLS doesn't quite fit, they hit a brick wall, and don't have a simple way to go further. In the end, they'll take the path of least resistance and just gloss over OLS's assumptions.

There is nothing more embarassing than a presenter who answers a question about the assumptions or results of a model with 'that's just Stata's default'—or still worse, (and yes, I have heard this in a real live presentation by a real live researcher) 'I would have corrected this anomaly in my data, but Stata didn't have a function to do that.' This is beyond unpersuasive and into the realm of confidence-eroding.

Stats packages aren't designed around the general results, though it's technically possible to retrofit these packages to use them. Since they're Turing-complete,[1] you could write anything in them: maybe a word processor

[1] Alan Turing wrote down an imaginary machine which could execute a dozen types of instruction. All modern programming languages implement these instructions in one way or another, and are therefore equivalent to Turing's theoretical computer; by transitivity, they are all equivalent to each other. You could write a C compiler in MatLab if you were

or a painting program. But why? It's just as easy to write them in C using the packages I discuss here, and the resulting program will be more robust and orders of magnitude faster, as per Reason #2.

1.3.2 Reason #2: Stats packages will break your heart.

Stats packages are wonderful at first, making it easy to sit down and start working quickly. As you get better with the language, you'll grow to depend on the stats package for more and more things. And then, one day, you'll get to a problem that's too far out from what the language designers had in mind, or a problem that is too large-scale for the language to handle. Your favorite language just won't be able to do it, even after you spend hours trying to get around the language's assumptions about what you mean and endless attempts to optimize the code so it'll run a little bit faster. And after all that, you'll have nothing but a broken heart.

[I'd especially like to mention here that languages which don't require type declarations seem nice at first but are, in the long run, bad for you. Also, an interpreted language which needs to be parsed into a set of functions (written in C) before executing will be painfully slow for large data sets—like weeks slow.]

1.3.3 Reason #3: C is universal

The software I discuss in this book is available from www.gnu.org, for the system you're using, and for the system you will be using five years from now, for free. There is no other language I could say that about with such confidence.

This is not only important because you may find yourself in front of a different type of computer next year, but because we increasingly expect that the data and analysis behind a work be publically available. For example, it's a requirement for National Science Foundation funding. But if your analysis uses a stats package which isn't universally available, then you will break the hearts of all of your fellow researcher who want to work with your analysis but for whom it's logistically impossible to do so. Are you sure your colleague in Madras can afford a Stata license?

---

persistent enough.

1.4 Outline

Since I am assuming that you are computer-literate but not a C programmer, Chapter 2 will give you a crash course in C. It will not only get you familiar with the rules of the language, but how to best think about problems in C. Chapter 3 will then introduce you to the package of C functions most useful for doing stats: the GNU Scientific Library (GSL). Notably, it will cover how to do linear algebra using the GSL.

With that introduction, you'll be ready to dive in to the statistics. There's a chapter devoted to each of the three categories above: Chapter 4 handles projections of your data into various subspaces; Chapter 5 covers methods of comparing your data to various Gaussian distributions, such as the t test, F test, and chi-squared test; Chapter 7 will show you how to write down your likelihood functions and find their maxima, as in Probit or Normit estimations; and how to test hypotheses using a likelihood ratio test. There are some other topics which need covering: I'll throw a chapter in there (Chapter 6) about bootstrapping and jackknifing, which are dirty tricks which will one day save your life. Chapter 8 will give you a quick lesson in getting your data in and out of a database; this chapter is toward the end because you can live your life without it, but it makes things easier, and will be worth tackling before you have your next monolithic data set foisted upon you. Finally, Chapter 9 will cover creative uses of Gnuplot.

When you're done with all this—and this is no exaggeration—you'll have the tools to implement any technique in classical statistics in existence today, on any data set, no matter how large or exceptional.

2 C

This chapter will bring you up to speed on C. It is not a comprehensive tutorial: GNU C recognizes 33 key words and this book will only use 18 of them. We won't be using bit-shifting operators, so I'm not going to tell you what they are here. I'm going rapidly through the details because the book is filled to the brim with sample code, so you should have no problem finding an example which will show you the workings of any detail you may be unsure about.

My hope with this chapter is both to discuss where to put your semicolons and how things are generally done in C. Don't feel compelled to memorize

everything here (using `malloc` three times in a sentence, for example), since the manual is always there for you to read.

[Notice also, dear editor, that the order is not the traditional order. I think the place to discuss the compilation process is as soon as you have a working program, and the correct place to bring up the debugger is definitely right after introducing pointers. Again: it's not the best way to write a reference on C, but I think a good way to get our intended audience coding quickly.]

2.1 Variables have to be declared

Let's get this out in the open right from the start: you will need to declare every variable before you use it. This consists of listing the type of the variable and then the variable name, e.g.:

```
int i, j=0;
double stuff, k;
```

Notice that we can initialize a few variables of the same type on one line, and could initialize `j` to zero right when we declare it. The other variables (such as `i`) have unknown value right now. Assume nothing about what's contained in a declared but uninitialized variable.

Your basic options for variable types are `int, float, double, long double` and `char`. Your guess at `int` is right; `float` is a floating-point number, aka a real number. Sometimes you'll need more precision, and so you have `double`, which are double-precision real numbers, and `long double`, which you can think of as quadruple-precision reals. `char` are characters.

There are other types, which aren't worth caring about.

Declaring types You can define your own types. For example, these lines will declare three pairs of numbers: a, b, and c:

```
typedef double pair[2];
pair a, b, c;
```

This is generally useful in two contexts: sometimes you'll confuse yourself declaring arrays of arrays, and a nice typedef can save the day; and you'll often want more complex data types, in which case you'll want to name those types. Here:

```
typedef struct xxx{
double real;
double imaginary;
} complex;
complex a, b;
```

You can now use `a.real` or `b.imaginary` to refer to the appropriate constituents of these complex numbers. Notice that you'll need a unique name just before the curly braces which you will never use anywhere else in the program. This is your first hint that C is neither Beautiful nor Perfect; just follow the template there and nobody gets hurt.

[Hopefully, dear editor, this will give you an idea of the level and speed I have in mind when talking about a crash course in C. The remainder of this section will be more of an outline.]

2.2 C is functional

That is, every line of your programs will be either a declaration or a function. The declarations, as you saw, are sort of annoying but basically easy. Writing functions that are preeminently useful will be the focus of the rest of the book.

This section will show the format of a function, function declarations, and make mention of the uniqueness of `main`.

2.3 Compiling and running

Once we've written a complete program, it'd be nice to know how to run the thing. You do so by processing the text you've just written with a compiler and a linker. Throughout this book, I'll be referring to the GCC, the GNU Compiler Collection, which will compile C as well as a few other languages. I'll use `gcc` (herein lower case to refer to the command you'll be typing) because it's probably the most universally available program in the world today—and it's free.

This section will discuss the command line and the object-code creating and the linking phases of compilation. It will include some discussion of how to deal with the most common errors.

Including other files The two-stage process of creating libraries of functions and then gluing them together makes C a wonderful, powerful, do-all

language. All you need to do is find the right library.

This section will continue with:
–Instructions on headers, and the utility of function declarations
–some stuff about the std lib
–some stuff about the GSL
–some stuff about writing your own function library.

2.4 Assorted syntax

–Comments
–Conditions
–Loops
–If-then-else
–Scope

2.5 Dealing with pointers

Pointers are the thing that really distinguishes C from the stats packages. If you've never dealt with them before, you will spent some quantity of time puzzling over them, and then you'll wonder what all the fuss is about.
This section will cover:
–the pointer concept
–where to put the stars
–arrays and their indices
It will make no mention of the fact that `(j[3]==3[j])`.

2.6 The debugger

Which brings us to debugging. Sometimes you'll get lucky when your code refers to `j[3]`, and that part of memory will be holding something innocuous, like a zero. And sometimes that memory location will be something useful to another program. This is a 'segmentation fault', since you've just looked at a part of memory that isn't in the segment allocated to `j`. Since this could be anywhere, the safest thing to do is to immediately halt the program.

This is where the debugger comes in. The debugger will let you pause the program anywhere, look at where you are in the program, and see the

value of every variable declared at that spot. This beats inserting little `print` statements all over your code by a mile [1.6 km].

This section will cover:

–Some mention of visual debuggers.

–The most essential commands on the `gdb` command line: break, backtrace, frame, print, info args, info locals

–A few notes on debugging technique.

2.7 Dealing with strings

[Since C is so ornery with regard to strings, I feel that it's worth mentioning that `str="assign me"` will crash. This will be short, because statistical analysis doesn't require much string handling.]

2.8 Other auxiliary programs

I've already talked about the debugger; here are a few more programs that will make your life as a programmer easier.

2.8.1 Make

[I'll basically give the reader a makefile which should work for everything in the book.]

2.8.2 Revision control

[CVS is optional, but I'm a big fan; e.g., this proposal is under revision control. I'm just that kind of guy. This section will probably just point the reader to http://cvshome.org, suggest they learn revision control at their earliest convenience, and leave it at that.]

3 Linear algebra prerequisites

To tell you the truth, this will be the least fun chapter of the book, because it is nothing but prerequisites for the actual applications that will follow. But once you have this stuff down, the sky's the limit.

3.1 the GSL's matrices and vectors

As you saw in the last chapter, arrays can be directly implemented in C, but for the rest of the book, I'll be sticking to the GSL's matrix and vector objects. If you like using raw arrays better, it's easy to switch back and forth; see section 3.2.

Here's some sample code which will do a few useless things to a few sample objects:

```
#include <gsl/gsl_matrix.h>
#include <stdio.h>


int main(void){
   gsl_matrix   *m = gsl_matrix_alloc(10,10);
   gsl_vector   *v = gsl_vector_calloc(10);
   int  i;

   for (i=0;i< m->size1; i++){
      gsl_matrix_set(m, i, 0, i) ;
   }
   printf("Here's point (3,0): %g\n", gsl_matrix_get(m, 3,0));
   gsl_matrix_set_row(m, 3, v);
   printf("Here's point (3,0) again: %g", gsl_matrix_get(m, 3,0));
   gsl_matrix_free(m);
   gsl_vector_free(v);
}
```

A walk through the code Here's what just happened: we allocated a 10×10 matrix and a vector of length 10. For the sake of variety, we allocated the two differently. `gsl_matrix_alloc` simply set aside a block of memory for the matrix, and that block may have garbage in it. Meanwhile, `gsl_vector_calloc` set aside some space for the vector `v`, and set all the values of `v` to zero. We were able to do these allocations in the declaration itself.

That done, the `for` loop put some values in the first column of the matrix. The syntax should be familiar to you from subsection 2.4: we start at zero, not one, and increment up to the size of the matrix, which in this case is

`m->size1`. You'll recognize this as accessing a `struct`, which is exactly what we're doing: the declaration `gsl_matrix *m` means that `m` is a pointer to a `gsl_matrix`, which is a structure whose definition you can look up if you're so inclined. If you do so, you'll see that it includes elements `size1` and `size2`, for the row and column sizes of the matrix. [Row always comes first, then Column, just like the order in Roman Catholic, Randy Choirboy, or RC Cola.] Since the vector has only one dimension, the analagous element of the vector structure is `v->size`.

Next, we copied the vector to the third row of the matrix using `gsl_matrix_set_row(m, 3, v)`. Notice that in so doing, we overwrote the three at the point (3,0) of the matrix with a zero from the third element of `v`.

Finally, we freed the memory used for the vectors. This is not strictly necessary for a small program, since the GLS and the operating system will clean up some of your mess. But it's a good habit to get into for when you start getting the monolithic analyses, which you may not be able to run on your PC if you don't keep the memory clear of debris.

Naming conventions Notice the consistency of the GSL's naming scheme. Every function in the GSL library will begin with `gsl_`. Every function which affects a matrix will begin with `gsl_matrix_` and similarly with vectors and their functions, which all begin with `gsl_vector_`. Further, the first argument of all of these functions will be the object to be acted upon.

You'll see this form over and over again: the library gives us an interesting object, which we'll mostly treat as a black box, and it gives us functions which will allow us to do useful things to that black box. The consistency of the naming means that you'll have more to type, but less to memorize. Your text editor probably has some sort of name completion command, which you may want to look up. E.g., vim users, try <ctrl-n>.

Another alternative is to start writing your own functions. For example, you could write a file `my_convenience_fns.c`, which would include:

```
double mget(gsl_matrix *m, int row, int col){
   return gsl_matrix_get(m,row,col);
}

double vget(gsl_vector *v, int row){
   return gsl_vector_get(v,row);
}
```

You'll also want a header file, say `my_convenience_fns.h`:

```
double mget(gsl_matrix *m, int row, int col);
double vget(gsl_vector *v, int row);
```

After throwing an `#include "my_convenience_fns.h"` up at the top of your program, you'll be able to use your abbreviated syntax such as `vget(v,3)`. It's up to your aesthetics as to whether your code will be more or less legible after you make these changes.

3.2 Shunting data

Featuring such conversions as: matrix to vector, array to matrix, and reading data into an array from a text file.

3.3 The BLAS

Before there was the GSL, there was the BLAS—the basic linear algebra system. The GSL has a few functions to interact with the BLAS. In fact, it has 86. Here are the three that you'll actually use.

matrix · vector Here's the function you'll use to calculate the dot product of a matrix and a vector:

```
int gsl_blas_dgemv (CBLAS_TRANSPOSE_t TransX, float alpha,
          const gsl_matrix * X, const gsl_vector * x,
          float beta, gsl_vector * y)
```

This will put into the vector y the value $\alpha op(X)x + \beta y$. If `TransX` is "`CblasNoTrans`", then $op(X) = X$; if it is "`CblasTrans`" then $op(X) = X'$, the transpose of X.

To give a concrete example, assume you've already got some vectors and matrices which have the following declarations:

```
gsl_vector beta, gamma;
gsl_matrix x, y;
```

Then, to calculate $X \cdot \beta$, we'd need:

```
#include <gsl/gsl_blas.h>
gsl_vector *beta_dot_x      = gsl_vector_calloc(x->size1);
gsl_blas_dgemv (CblasNoTrans, 1.0, x, beta, 0.0, beta_dot_x);
```

Notice that we used `calloc`, instead of just `alloc`, because the system will add $x\beta$ to `beta_dot_x`, not just write it in, so `beta_dot_x` needs to start as all zeros.

vector · vector To find the dot product of two vectors, use this function:

`int gsl_blas_ddot (const gsl_vector * x, const gsl_vector * y, double * result);`

For example,

```
#include <gsl/gsl_blas.h>
gsl_vector *beta_dot_gamma      = gsl_vector_calloc(beta->size);
gsl_blas_ddot (beta, gamma, beta_dot_gamma);
```

matrix · matrix Finally, to take the dot product of two matrices, you'll need:

`int gsl_blas_dgemm (CBLAS_TRANSPOSE_t TransX, CBLAS_TRANSPOSE_t TransY, double` α`, const gsl_matrix * X, const gsl_matrix * Y, double` β`, gsl_matrix * dot_product)`

which will calculate `dot_product` $= \alpha op(X)op(Y) + \beta$ `dot_product`. $op(X)$ and $op(Y)$ will be either the matrix or its transpose, as above, depending on whether you choose `CblasTrans` or `CblasNoTrans`. For example, heres $X'Y$:

```
#include <gsl/gsl_blas.h>
gsl_matrix *x_dot_y      = gsl_matrix_calloc(x->size1, y->size2);
gsl_blas_dgemm (CblasTrans,CblasNoTrans, x, y, x_dot_y);
```

3.4 Matrix inversion and equation solving

Matrix inversion is one of the most computationally intensive problems around. In fact, some will tell you it is the problem for which computers were invented. The GSL discourages you from taking inverses directly, since you often don't need to. For example, we often write the OLS parameters as $\beta = (X'X)^{-1}(X'Y)$, but you could implement this as solving $(X'X)\beta = X'Y$, which involves no inversion.

[The GSL has functions for solving, but no functions for inverting general matrices, meaning that the user has to first decompose the matrix to be inverted into triangular matrices. I'll put some code in here to do that.]

4 Describing data with projections

A good part of statistical analysis is about projecting your N-dimensional data onto the best subspace of significantly less than N dimensions. This chapter will cover the best way to effect this projection given different definitions of 'best'. For example, the standard OLS regression consists of finding the one-dimensional line which minimizes the sum of squared distances between the data and that line. Factor analysis consists of finding the few dimensions where the data's variance is maximized, after being projected onto the subspace.

4.1 OLS

OLS is a projection onto a one-dimensional space. I'll give some code about dealing with it here; this is basically a gluing-together of everything in the last chapter.

4.2 Principal component analysis

This is also known as factor analysis or as spectral decomposition, depending upon your field.

This is a purely descriptive method. The idea is that we want a few dimensions that will capture the most variance possible—usually two, because we can plot two dimensions. That is, we will project the data onto the best plane, where 'best' means that it captures as much variance in the data as possible.

After plotting the data, perhaps with markers for certain observations, we may find intuitive descriptions for the dimensions that we had just plotted the data on. My favorite example of this is the work of Poole & Rosenthal, who did a principal component analysis[2] on all of the U.S. Congresses. They

[2]They actually did the analysis using an intriguing maximum likelihood method, rather than the eigenvector method here. Nonetheless, the end result and its interpretation is the same.

found that 90% of the variance in vote patterns could be explained by two dimensions. Studying the data points, they determined that one of these dimensions could be described as 'fiscal issues' and the other as 'social issues'. This method stands out because Poole & Rosenthal didn't have to look at bills and place them on either scale—the data placed itself, and they just had to name the scales.

It can be shown that the best n axes, in the sense above, are the n eigenvectors of the data's covariance matrix with the n largest associated eigenvalues.

4.2.1 Coding it

The only hard part is finding the eigenvalues of $(X'X)$; the GSL saw us coming, and gives us the `gsl_eigen_symm` functions to calculate the eigenvectors of a symmetric matrix.

I don't like the GSL's eigenvector syntax, which involves creating and freeing an eigenvector-finding workspace. In the grand tradition of writing convenience functions to work the way you do, I hid the workspace it in my own function, which allocates the workspace, does the math, and deallocates the workspace:

```
void find_eigens(gsl_matrix *subject, gsl_vector *eigenvals,
                            gsl_matrix *eigenvecs){
   gsl_eigen_symmv_workspace * w
                        = gsl_eigen_symmv_alloc(subject->size1);
   gsl_eigen_symmv(subject, eigenvals, eigenvecs, w);
   gsl_eigen_symmv_free (w);
   gsl_matrix_free(subject);
}
```

Notice that I free the matrix whose eigenvalues are being calculated at the end. This is because, for all intents and purposes, the matrix is destroyed in the calculations, and shouldn't be referred to again.

Here's how this routine is used. In the style of C, the code is mostly declarations, and in the last two lines, it will calculate the covariance matrix $X'X$ for the data set, and then find its eigenvalues and eigenvectors.

I will assume that you've already got a data matrix ready, named `data`, as per the last Chapter, and you've subtracted the means of each column.

```
#include <gsl/gsl_eigen.h>
int ds=data->size2;
gsl_matrix *cov_matrix  = gsl_matrix_calloc(ds, ds);
gsl_vector *eigenvals   = gsl_vector_alloc(ds);
gsl_matrix *eigenvecs   = gsl_matrix_alloc(ds, ds);

gsl_blas_dgemm(CblasTrans,CblasNoTrans, x, x, cov_matrix);
find_eigens(cov_matrix, eigenvals, eigenvecs);
```

Now we have the eigenvectors and their associated eigenvalues; we need only find the largest eigenvalues, and project the data onto their associated eigenvectors. The GSL helps us by giving us functions for finding the indices of the largest elements of a vector.

```
#include <gsl/gsl_sort_vector.h>
const int dimensions_we_want = 2;
gsl_matrix *pc_space
               = gsl_matrix_alloc(ds,dimensions_we_want);
gsl_vector *temp_vector = gsl_vector_alloc(ds);
int indexes[dimensions_we_want];
int i;

gsl_sort_vector_largest_index(indexes, dimensions_we_want, eigenvals);

for (i=0;i<dimensions_we_want; i++){
   gsl_matrix_get_col(temp_vector, eigenvecs, indexes[i]);
   gsl_matrix_set_col(pc_space, i, temp_vector);
}
```

All that's left to do is the projection. Notice the convention I used: the `pc_space` has eigenvectors on its columns, and as many columns as the dimensionality we want in the end. Below, I transpose that before premultiplying the data set by the principal component matrix.

```
gsl_matrix *projected
                = gsl_matrix_alloc(data->size1, dimensions_we_want);
gsl_blas_dgemm(CblasTrans,CblasNoTrans, pc_space, data, projected);
```

You'll probably want to plot the `projected` matrix; I'll continue this example in the chapter on plotting.

5 Hypothesis testing with Gaussian distributions

This subsection covers most of what we traditionally learn in first-year statistics. Most of the work will consist of taking a dot product, maybe inverting a matrix, and then looking up a number in a table.

Everything here depends on the Central Limit Theorem. If your data doesn't fit the CLT, then please don't use these techniques. Work out how your data is distributed, to the best of your abilities (try bootstrapping, Chapter 6), and then write down a likelihood function. If you're looking to estimate model parameters, do a maximum likelihood estimation; if you're looking to test a hypothesis, write down a likelihood ratio based on the distribution you've just calculated.

5.1 Meet the Gaussian family

[This subsection would cover the relations between the major distributions used for testing hypotheses about parameters: t, Gaussian, chi-squared, F. Yes, it's in the textbook, but nobody I talk to ever remembers this stuff, and I think a good explanation will keep the reader clear on exactly what we're testing when.]

5.2 Good ol' OLS

This section would discuss how to write your very own `regress` function, which, as noted above, just consists of solving for the β in $(X'X)\beta = X'Y$. The last chapter showed us the code to find the betas with the smallest squared error; this section will cover testing hypotheses about those betas.

5.2.1 GLS

Generalized least squares refers to any method that uses a variance-covariance matrix that isn't the identity matrix. Having written our `regress` function, it's almost trivial to generalize to GLS. But the fun of GSL is in working out what that matrix should be. This section would give examples of favorites such as AR-1 processes from time series analysis (and hey, why not AR-N?).

5.3 Hypotheses about the variance

Chi-squared tests and their utility.

5.4 F tests

You can test any hypothesis in the OLS world using an appropriate F test. Mathematically, it's a generalization of OLS, and we can implement it in the code as such.

5.5 Testing the assumptions

Errors have to be normally distributed, or else the whole OLS system here doesn't apply. Many stats package user's manuals suggest plotting the errors and then squinting at the picture. Me, I'm a fan of a slightly more scientific approach, which is based on the fact that a Normal distribution has only two parameters: the mean and the standard deviation. Everything else about the Normal distribution is defined therefrom. Notably, the skew is zero, and the kurtosis is $3\sigma^4$. We've already written everything we need to calculate all of these easily.

[I had some code here, but I don't like it. It just prints the first four moments of a data vector.]

You can either just eyeball this, and decide based on the scale of the variance whether the skew and the kurtosis look like they're far off from where they should be, or you could bootstrap the variance of the kurtosis, which would let you find a confidence interval around $3\sigma^4$ and state with a certain percentage of certainty that the kurtosis is or is not where it should be. Hint: the CLT applies to the kurtosis, so you know it has a Gaussian distribution.

[Oh, and by the way, the GSL gives you normalized kurtosis, not actual kurtosis. Will discuss that here, since I didn't notice that line in the documentation when I first started using the GSL and it really screwed me up for a while.]

6 Bootstrapping

Bootstrapping is probably one of the most eerily descriptive names I've ever seen. After you've gotten everything you can out of the data, then on top of

that you can bootstrap to find the variance of all of that.

At some point, I'll say more about it here.

An important caveat Bootstrapping from a sample will not fix the errors in your sample. If your sampling technique isn't perfect—and it isn't—then it won't capture the full variation in the data. That means that the variances you calculate using bootstrap will be less than the true variance. In a bind, it's all you've got, and you'll just have to state that and go on. But having smaller variances makes it easier to reject the null hypothesis, which is what your paper is probably trying to do, so bootstrapping works slightly in your favor, and against parsimony and skepticism. Therefore, if there is any way of getting information about the variance of your variable without bootstrapping, even if that estimate overstates the variance, then use that instead of bootstrapping. Otherwise, you dishonor your name, and bring shame to your research group.

6.1 Finding the variance of a parameter

[Odd that I felt the need to write that caveat in full, but I'm just leaving a placemarker for the content here. Anyway, here's the bootstrapping process:
–create random samples
–calculate the parameter you're interested in; place it in a vector
–find the variance of that vector
–Test your hypothesis using that variance, your parameter's value, and a Normal distribution

Naturally, the GSL has functions that will help you do all of the above.]

6.2 Finding a distribution

This section is about generating data from a model you've written yourself.

[There's even a GSL function to do this, which I stumbled upon in the documentation for the GSL one day—in the section on drawing histograms.]

7 Maximum likelihood estimation

Maximum likelihood estimators (MLEs) are the bread and butter of statistics. Most of the techniques we've handled so far are MLE techniques, except

mathematicians over the ages have found ways to hide that fact from you. But if there isn't a nice, convenient way to get around doing the maximization, you'll have to do it yourself. Fortunately, the GSL has the `gsl_multimin` family of objects, to help you find the optimal parameters.

7.1 Why likelihood functions are great

There are two reasons, with four names.

MLEs achieve the Cramer-Rao lower bound The CRLB of variance is the inverse of the derivative of the derivative of the log of the likelihood function. [I'll get around to writing it nicely later.] It's called a 'lower bound' because Mr.s Cramer and Rao proved that any estimator of β must have a variance greater than or equal to the CRLB. Your favorite statistics textbook will also prove that for the MLE, the variance is actually equal to the CRLB, meaning that we can not get an estimator of β which will have any smaller a variance.

The Neyman-Pearson lemma There are two types of error we could have with a hypothesis test: Type I is that we reject the null when it's true; Type II is that we accept the null when it's false. The first type is the one we focus on, because it's what we mean when we say that our test has $\alpha = 95\%$ confidence. What about Type II errors? Well, Neyman and Pearson showed that a likelihood ratio test will have the minimum possible Type II error of any test with the α that we selected. After establishing this fact, we just ignore Type II errors.

7.2 Writing your likelihood function

This section is about writing down the likelihood function for those situations where a linear least squares function is not appropriate. Writing down your function is pretty darn straightforward, but there are details you'll need to take into account.
–Take logs. This is important from a practical standpoint because the product of a thousand probabilities $\in (0, 1)$ will quickly underflow your likelihood function.
–Follow the `gsl_multimin` template.

[I'll eventually write this out, but for now, here's an example:]

```
#include "gsl_convenience_fns.h"
double probit_likelihood(const gsl_vector *beta, void *d){
int i;
long double n, total_prob = 0;
gsl_matrix *data  = d; //just type casting.

   dot(beta,data);    //a wrapper for gsl_blas_dgemm.
   for(i=0;i< data->size1; i++){
      n =gsl_cdf_gaussian_P(gsl_vector_get(beta_dot_x,i),1);
      if (gsl_matrix_get(data, i, 0)==0)  total_prob += log(n);
      else  total_prob += log((1 - n));
   }
   return -total_prob;
}
```

7.3 Description: Maximum likelihood estimators

If you've written down the function correctly in the last section, you'll have no problem getting the GSL to find the optimal parameters given your data. It helps if you know the derivative of your data, which I'll also discuss a little further.

7.3.1 The GSL's multimin functions

The process of finding a minimum consists of trying a value, picking a direction to move in, and checking whether the change is a good enough one. The GSL gives you a function to do all of these things, which you'll have to put together to do a complete minimization.

7.4 Hypothesis testing: Likelihood ratio tests

Every test in the last chapter was a likelihood ratio test—I just didn't tell you what the likelihood function was. Those functions are easy because they've been carefully studied and methods have been found to let you calculate them without explicitly writing down a likelihood function and calculating its value at various locations.

But if your data is at all interesting, then you'll need to write down the likelihood function yourself. Fortunately, we have a computer to do the tedious math, unlike poor Mr. Gauss, who had no such conveniences.

What if you don't know the variance or distribution? The main convenience of the canned methods of the last chapter is that we have mathematical proofs telling us exactly what the distribution looks like. What if our data doesn't fit the assumptions of any of the proofs in the textbook?

Then we bootstrap to find the distribution! I'll put an example here which puts together the techniques from the last chapter and this one, constructing a likelihood function like the one above using the bootstrap.

8 Databases

If you've been reading sequentially, you've already got all the techniques you'll need to do statistics. This chapter is basically a convenience, although as conveniences go, it's a great one.

One especially nice thing about keeping your data in an SQL database is that it'll give your data names again. Here's some valid SQL: `select age, gender, year from survey`. Why, that's proper English. It goes downhill from there in terms of properness, but at its worst, it's still pretty easy to look at a query and know what's actually going on.

8.1 Basic queries

[This section will discuss the syntax of selection, with focus on the joys of `group by`, which is one of the things that SQL does easily and matrix-oriented programs do poorly. Joins will be covered, but not comprehensively, since they're not particularly essential for the work we're doing here.]

8.2 Using SQLite

[Most of the hard part of dealing with SQLite is in writing good callback functions. I've written a few functions to get matrices in and out of the GSL's matrices which are a good example of doing that, and probably all the reader needs. I'll describe them here.

And did you know how to use in-memory databases? Hint: it's supported, but isn't in the documentation; you have to read the source code to find the trick.]

8.2.1 Getting data in and out

We now have three different ways to represent a matrix of data: as a `gsl_matrix`, as a text file, or as a database table. This section will show you how to best shunt your data between a database and the other two formats.

8.3 An example: dummy variables

Here's a neat trick: using SQL's `case` a few dozen times, we can turn a variable which is discrete but not ordered (such as district numbers in the following example) into a series of dummy variables. It requires writing down a separate `case` statement for each value the variable could take, but that's what `for` loops are for.

[Note to editor: I just cut and pasted this from my hard drive. Will clean it up later. The gist is that we first query out the list of districts; then we write a select statement with a line `case district when district_no then 1 else 0` for each and every district_no. You can then run your regression on the output of the query without any further manipulation.

Notice that the for loop goes from i=1, not i=0; this is because when including dummy variables, you always have to exclude one value, which will be the baseline; using i=1 means district[0] will be the baseline.]

```
gsl_matrix *districts, *data_set;
query_to_matrix(&districts,
        "select distinct district from survey \
        where   age!=-1 and gender !=-1 and race !=-1 and \
        year !=-1 and district !=-1 and \
        (votedverified ==1 or votedverified ==3)");

query=malloc(sizeof(char)*10000);
strcpy(query,"select \
        case votedverified when 3 then 1 else 0 end,  \
        round(age/10) as age_group, gender, ");
for(i=1;i< districts->size1; i++){
```

```
        query=realloc(query, sizeof(char)*(strlen(query)+500));
        sprintf(query,
                 "%s case district when %i then 1 else 0 end, \n",
                 query, (int) gsl_matrix_get(districts,i,0));
        }
query=realloc(query, sizeof(char)*(strlen(query)+5000));
strcat(query, " year-1950 \
        from survey \
        where   age!=-1 and gender !=-1 and race !=-1 and \
        year !=-1 and district !=-1 and \
        (votedverified ==1 or votedverified ==3)");
query_to_matrix(&data_set, query);
```

8.4 An example: the easiest t-test you'll ever run.

Say we have a set of observations of our sample's years of education, and their annual income. We want to know if getting that grad school education is *really* worth it. The null hypothesis is: (Income for people with education less than 16 years) $\leq$ (income for people with greater than or equal to 16 years of education).

That first data set is:

```
#include <gsl/gsl.h>
#include <gsl/gsl_matrix.h>
#include "sqlite_wrappers.h"
gsl_vector *undereducated;
   query_to_vector(&undereducated,
      "select income from survey \
      where education <16");
```

while the second group is:

```
gsl_vector *overeducated;
   query_to_vector(&overeducated,
      "select income from survey \
      where education >=16");
```

Here's a factoid for you: incomes are usually distributed log-normally, so we should do a t-test on the log of income:

```
#include <gsl/gsl_sf_log.h>
int i;
for(i=0;i< overeducated->size; i++)
   gsl_vector_set(overeducated, i,
                gsl_sf_log(gsl_vector_get(overeducated, i)));
for(i=0;i< undereducated->size; i++)
   gsl_vector_set(undereducated, i,
                gsl_sf_log(gsl_vector_get(undereducated, i)));
```

We've already written functions to find the mean and variance of a vector [though I've omitted them from this overview]:

```
#include "gsl_wrappers.h"
double mean_over, mean_under, var_over, var_under;
mean_over  = mean(overeducated);
mean_under = mean(undereducated);
var_over   = variance(overeducated);
var_under  = variance(undereducated);
```

The other factoid you'll need is that the difference of two normal distributions is also normal, and the variance of the difference is the sum of the two original variances.[3] That is, we can write down:

```
#include <gsl/gsl_cdf.h>
double test_me = gsl_cdf_gaussian_P(mean_over-mean_under,
                                    var_over+var_under);
```

and `test_me` is the probability that the difference between the means is less than or equal to zero. If `test_me` turns out to be greater than your preferred confidence interval, (e.g., 95%), then reject the null and go to grad school. Else, there isn't enough information to distinguish between the two with confidence.

[3]Your stats textbook will tell you that the sum of two Normals is normal: $\mathcal{N}(\mu_1, \sigma_1) + \mathcal{N}(\mu_2, \sigma_2) \sim \mathcal{N}(\mu_1 + \mu_2, \sigma_1 + \sigma_2)$. Now, subtracting a $\mathcal{N}(\mu_2, \sigma_2)$ is exactly equivalent to adding a $\mathcal{N}(-\mu_2, \sigma_2)$, so we get the result in the text.

9 Gnuplot

This chapter will give you some tips on plotting using Gnuplot, which is a graphing package which is as open and freely available as gcc. [It is unrelated to the GNU, by the way. The name is a compromise between the names the two main authors preferred: nplot and llamaplot.]

Gnuplot does nothing but plot points. It has some interface with the GSL to plot functions for you, but you mostly won't be concerned with those: you'll just want to plot your data.

9.1 Dumping output to an agreeable format

9.1.1 Histograms

Gnuplot, refusing to do calculations for you, makes making histograms a pain. Fortunately, the GSL has you covered, with the `gsl_histogram` object.

9.2 Autogenerating Gnuplot scripts

Since there are so many ways to tweak Gnuplot, I like to include a subprocedure to write Gnuplot scripts in the main program. I'll give a few examples here.