1 Purpose of this document

This is not an introduction to SAS. Nor is it a manual on how to do specific things that a finance PhD might be interested in doing: that has been done very adequately by [Boehmer, Broussard, and Kallunki (2002)].

Rather, it is a description of a set of tools and constructs that I use frequently in my data work; a set which is likely to be of help to researchers and PhD students doing similar data work.

Should you find this useful, or have comments that I might find useful, please email me: mailto: adesouza@stern.nyu.edu.

2 Where to get help for SAS problems

SAS-L, the SAS users’ mailing list, is archived here: [http://www.listserv.uga.edu/archives/sas-l.html](http://www.listserv.uga.edu/archives/sas-l.html). I find it tedious to search. A better solution is to use Google. When you search for any SAS-related term, you’re very likely to find research notes written for SAS USer Group (SUG) meetings. SAS User Groups meet frequently all over the US and the world, and publish notes which describe specific features of SAS. Many of the references of this document are such notes.

The first place to go when you want to learn how to use a new feature of SAS (as opposed to being stuck on a particular problem) is an SUG publication (there may be several) on that topic. For instance, to learn about arrays in SAS, you could read “Arrays Made Easy: An Introduction to Arrays and Array Processing” [http://www2.sas.com/proceedings/sugi30/242-30.pdf](http://www2.sas.com/proceedings/sugi30/242-30.pdf). Almost all these notes are easy to understand, and have detailed examples.

3 Some preliminaries you should know

3.1 SAS basics

I assume you know how to read data into SAS, create a dataset, and run a data step. I assume familiarity with code that says

```sas
data first(rename=(ret=unsquared_ret));
  set crsp.msf(keep=permno date ret);
  squared_ret=ret**2;
run;
```
You should also know how to sort datasets and run, say, proc means:

```plaintext
proc sort data=first;
  by date;
run;
proc means data=first;
  by date;
var ret;
run;
```

You should know how to redirect output:

```plaintext
proc sort data=first;
  by date;
run;
proc means data=first noprint;
  by date;
var ret;
output out=sumrets sum(ret)=sum_ret;
run;
```

### 3.2 The logic of the DATA step

There are subtleties to the DATA step that it would help to understand.

The DATA step is nothing but an implicit loop. The standard DATA step looks like this:

```plaintext
data a;
  set b;
  <do something>
  run;
```

What this does is to read from dataset b, one observation at a time, do something to that observation, and then output that observation to dataset a. This is an implicit loop, which in pseudocode, might be made explicit by saying:

```plaintext
DO I = 1 TO ROWS(B);
  CURRENT_OBSERVATION=B[I,.;]
  <do something to CURRENT_OBSERVATION>;
OUTPUT A;
END;
```

treating A and B as matrices so ROWS(B) and B[I,.] (by which I mean the \(I^{th}\) row of the matrix B – the \(I^{th}\) observation) – have meaning.

There are two things that it is important to know. One is the role of the OUTPUT statement. The other is the role of the counter \(N\).
3.2.1 The OUTPUT statement

The output statement in a DATA step outputs the observation in its current state to the dataset named on the data line (or all datasets on the data line if no one is specified). Consider the example below:

```r
data first first_positive;
    set crsp.msf(keep=permno date ret);
output first;
if ret>0 then output first_positive;
run;
```

The dataset first contains all observations read from crsp.msf; the dataset first_positive contains those for which the returns were positive. There are two things to note about output statements. First, every data step contains an implicit output statement. If there is no output statement in a dataset, then the observation is output as-is at the end of the data step. The two data steps below are identical:

```r
data first;
    set crsp.msf(keep=permno date ret);
output first;
run;
```

```r
data first;
    set crsp.msf(keep=permno date ret);
run;
```

Second, when there is an explicit output statement in a data step, the implicit output statement is turned off.

```r
data first first_positive;
    set crsp.msf(keep=permno date ret);
if ret>0 then output first_positive;
run;
```

If you run this code, the dataset first has zero observations, since (a) there is an explicit output statement in the data step thus turning the implicit output statement off and (b) no output statement references the dataset first.

3.2.2 The counter _N_

The counter _N_ counts the iterations through the implicit loop of the DATA step. It increments when the top of the loop is encountered.

Suppose dataset B consists of one variable, called INDEX, with values 1-10, in order.
If I run this code

```sas
data b;
  set b;
  counter=_N_;  
  run;
```

I get

<table>
<thead>
<tr>
<th>Obs</th>
<th>INDEX</th>
<th>counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Suppose instead I were to run:

```sas
data b;
  set b;
  counter=_N_;  
  _N_=1;  
  run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>INDEX</th>
<th>counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
one would imagine that counter would always have the value 2, since \( N \) is being reset to 1 at every iteration of the data step loop. But this is not the case:

<table>
<thead>
<tr>
<th>Obs</th>
<th>INDEX</th>
<th>counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

This means that \( N \) is only a place that SAS puts the value of an internal counter which we never see and have no way of changing. Whatever you do to \( N \) in the data step, when the loop comes to the head of the DATA step, SAS will place the incremented value of the internal counter in \( N \).

One useful construct is:

```sas
data b;
    set b;
    if _n_ =1 then do;
        <some stuff>
    end;
run;
```

which does “some stuff” at the first value read from b and nowhere else. This is useful for initializing variables and hashes (see section 12).

### 3.3 Some CRSP variable names

Since this is intended for finance PhDs, the examples often use CRSP data\(^1\). If you are unfamiliar with CRSP, all you need to know is here.

Stocks may be identified by CUSIP, ticker, or name. Stocks in CRSP are identified by PERMNO, which is very useful because it does not change when the company changes superficially, in contrast with, say, CUSIPs, which may change after companies change their names. The monthly stock file on WRDS, crsp.msf or /wrds/crsp/sasdata/msf.sas7bdat has a host of monthly stock return information, but the variables of most interest to us are

\(^1\) Also see the section on WRDS datasets.
For the stock identified by PERMNO, RET is the holding period return from the beginning of the month to the end of the month.

### 3.4 Macro basics

Much of the text relies on a knowledge of SAS's macro language. I assume that you know what a macro variable is, how to create one (via `call symput` or explicit `%let` statements), and how to create and run macros. If you do not know this, your life will probably be immeasurably improved by learning it. A good place to start is the SUG paper by Delwiche and Slaughter (whom you may remember from The Little SAS Book): Delwiche and Slaughter (2004).

### 4 Move to SAS on Unix

If you have not done so already, I strongly urge you to move to SAS on Unix, rather than using SAS on your desktop. Using SAS on your desktop means that:

- You can run only one SAS program at a time
- You are limited by the amount of space and memory your PC has
- When you run programs that use lots of resources your computer hangs
- You can’t access your programs from anywhere (unless you have Remote Desktop set up)
- You are compelled to use SAS's editor, which is almost as bad as Notepad.

None of these limits apply to SAS on Unix. Of course, in the first two points above, these limits are replaced by larger limits.

Moreover, WRDS is much simpler to access via SAS on the WRDS server than it is via the web interface. Your control over the data you extract from WRDS' datasets is greatly increased².

#### 4.1 Cygwin and X-Windows

While you can get by using PuTTY (available from [http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html](http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)), I recommend that you go the whole hog and download Cygwin ([http://www.cygwin.com](http://www.cygwin.com)), because it effectively turns your Windows PC into a Unix server. It gives you X-Windows when you connect to servers, which means, among other things, that you can get font highlighting in Emacs in colour.

I also recommend you learn to use Emacs. The learning curve is somewhat steep, but you will find it well worth the effort. Emacs is available with Cygwin, and is installed on virtually all Unix machines, so even when you use PuTTY to connect to a server, you can always say

    emacs -nw myfile.sas

²In some instances, you may actually prefer to use the Web interface to extract some data, which would be complicated to merge in using the Unix interface, but these instances are not many.
at the command line to start up Emacs. The “-nw” (no window) tells Emacs that you are not running X-windows.

One disadvantage of using SAS on Windows is that you are compelled to use SAS’s editor, which does not have one-tenth of the features you will come to expect from editors after using Emacs.

5 Reading in datasets

5.1 Quick and dirty: convert proc import log output to an input statement

Reading in CSV or otherwise delimited text files using proc import does not give you the control you want over how the variables are defined, but using an input statement means you have to type out all the (possibly tens of) variable names and format definitions. It also means that you have to know all your variable names and what exactly SAS’s name for the delimiter is and a whole host of other things.

A useful way to have control while not doing too much work is to use proc import on a small subsample first, and then use what proc import writes to the log to construct your own input statement, changing variable names or formats as you like.

For instance, I have the MSF dataset stored as a CSV file: text.csv. The first two lines are

\[
\begin{align*}
\text{PERMNO, DATE, RET} \\
10000, 19860228, -0.257143
\end{align*}
\]

A proc import would read:

```
proc import file='text.csv' out=out dbms=csv;
run;
```

and produce the following in the log file:

```
data WORK.OUT ;
%let _EFIERR_ = 0; /* set the ERROR detection macro variable */
infile 'text' delimiter = ',' MISSOVER DSD lrecl=32767 firstobs=2 ;
informat PERMNO best32. ;
informat DATE best32. ;
informat RET best32. ;
format PERMNO best12. ;
format DATE best12. ;
format RET best12. ;
input
PERMNO
DATE
RET
```

\[3\text{This is not, strictly speaking, true. You can run SAS in batch mode at Cygwin’s command line, but I am talking about SAS installed on Windows without Cygwin, as opposed to SAS on a machine with Cygwin installed.}\]
24 ;
25 if _ERROR_ then call symputx('_EFIERR_',1); /* set ERROR detection macro variable */
26 run;

Notice that the date variable is stored as a number, and it would have been a hassle to get it into the correct format.

But working from this log, it is trivial to produce the following clean input data step:

data WORK.OUT;
infile 'text' delimiter = ',' MISSOVER DSD lrecl=32767 firstobs=2;
informat PERMNO best32. ;
informat DATE YYMMDD8. ;
informat RET best32. ;
format PERMNO best12. ;
format DATE date9. ;
format RET best12. ;
input
  PERMNO
  DATE
  RET
; run;

Notice the change in the informat and format for date, so that the output dataset has date correctly coded.

5.2 guessingrows

Proc Import uses the first 20 rows of your data file to decide what kind of data each variable is. If your data is such that the first 20 rows are not enough, you can use the guessingrows option in proc import:

    proc import file='text.csv' out=out dbms=csv;
guessingrows=300;
run;

This is useful if you have mixed character and numeric variables for which the first 20 observations are all numeric.

5.3 Why does proc import hang on Unix?

In some circumstances, when you run a SAS program on Unix with proc import (or proc export), the program reaches the import step and just sits there, not moving forward nor dying with an error. What may be happening is that proc import is trying to open an X-window telling you that it is running, and you are not running an X-windowing system.

To get around this, at the command prompt type
sas -noterminal myfile.sas &

instead of

sas myfile.sas &

This prevents SAS from trying to open an X-window to tell you about the progress of proc import. This also works with other procs that try to open windows: for instance, proc gplot.

6 Working with WRDS datasets

6.1 CRSP: DSF/MSF

- The date in the MSF file is not the end-of-month, but the last trading day of each month. To avoid errors, I always reset the date when I open crsp.msf:

```sas
data msf;
   set crsp.msf(keep=permno date ret);
   date=intnx('month', date, 1)-1;
run;
```

- Closing prices in the MSF and DSF files are negative if the price is not the price from an actual trade, but the average of bid-ask spreads. I always set prc=abs(prc) before I do anything else with prices.

6.2 Names files in CRSP: stocknames and fundnames

Aside from the DSF and MSF files (and analogous files for mutual funds), data that changes infrequently and irregularly for each permno or crsp_fundno (like company name or ticker or exchange) is stored in names files. These files have the form, for instance

```
PERMNO NAME BEGINNING_DATE ENDING_DATE
```

where BEGINNING_DATE is the first date on which that PERMNO has that NAME (or ticker, or ncusip, or whatever), and ENDING_DATE is the last date on which that PERMNO has that NAME.

For stocks, this file is crsp.stocknames, for funds, it is crsp.fundnames. Other files stored in this format include crsp.fund_fees and crsp.fund_style.

Consider stocknames. The BEGINNING_DATE and ENDING_DATE variables here are named NAMEDT and NAMEENNDT. To change stocknames into a dataset that can be merged with, say, msf, I can do
data stocknames(keep=permno date ticker);
    set crsp.stocknames(keep=namedt nameenddt permno ticker);
where namedt is not missing;
date=intnx('month', namedt, 1)-1;
do while(date<=nameenddt);
    output;
    date=intnx('month', date+1, 1)-1;
end;
run;
data msf;
    set crsp.msf(keep=permno date ret);
date=intnx('month', date, 1)-1;
run;

and then merge the two by permno and date.
The statement

    date=intnx('month', namedt, 1)-1;

moves the date to the end of the month in which namedt lies.
If you do not include the “where namedt is not missing”, missing namedts will result in an infinite loop, since SAS treats the missing value as equal to negative infinity in numerical comparisons.
There are special missing values for namedt and nameenddt given by “.B” and “.E”, to indicate the “Beginning” of time and the “End” of time respectively. If you want to include these observations, you must say:

    data stocknames(keep=permno date ticker);
    set crsp.stocknames(keep=namedt nameenddt permno ticker);
if namedt=.B then namedt='01Jan1900'd;
if nameenddt=.E then namedt='31Dec2020'd;
if not(missing(namedt));
date=intnx('month', namedt, 1)-1;
do while(date<=nameenddt);
    output;
    date=intnx('month', date+1, 1)-1;
end;
run;

That is, you allow SAS to read in missing values, change them appropriately, and delete observations where namedt is still missing. Notice that the statement says “if namedt=.B”, without any quotes around “.B”. Also notice that the “namedt is not missing” has become “not(missing(namedt))”. SAS does not support the use of the “is not missing” construct outside where statements.
6.3 CRSP: ncusip and cusip

The CUSIP that CRSP reports is the so-called “header CUSIP”, which is the last CUSIP belonging to that PERMNO. A PERMNO may have had several different CUSIPs in the past. These are called the “names CUSIPs” (NCUSIPs) by CRSP.

Suppose I have data about stocks that only has historical CUSIPs as identifiers. Examples include holdings data, governance data, and mutual fund voting data. To convert these into PERMNOs, I could use the stocknames file, doing as suggested by Subsection 6.2. Alternatively, I could use the fact that CUSIPS are not reused in the data:

```
proc sort data=crsp.stocknames(keep=permno cusip) out=permnocusip(rename=(cusip=ncusip)) nodupkey;
where cusip is not missing;
by permno cusip;
run;
```

```
proc sort data=crsp.stocknames(keep=permno ncusip) out=permnoncusip nodupkey;
where ncusip is not missing;
by permno ncusip;
run;
```

```
proc append base=permnoncusip data=permnocusip;
run;
```

```
proc sort data=permnoncusip nodupkey;
by permno ncusip;
run;
```

The last file is a permno-ncusip lookup table, which I can merge with my CUSIP data to obtain permnos.

Why did I include CUSIPs as well? One would expect that the NCUSIP history perfectly accounted for even the last CUSIP held - the header CUSIP. But it does not always.

6.4 Adjusting shares in holdings data

A constant problem in using holdings data is adjusting the shares held for stock splits between the reported date and any date in the future.

For instance, if you are calculating the change in shares of Adobe held by a mutual fund from January 1, 2005 to December 31, 2005, you need to adjust the shares held on January 1 for splits over the year before you can subtract them from the shares held in December.

The way to do this is to use CRSP’s CFACSHR variable, available in the MSF file. To adjust shares reported on \textit{reported date} to another date, \textit{desired date}, I say

\[
\text{shares}_{\text{reported, adjusted to desired date}} = \frac{\text{shares}_{\text{reported}} \times \text{cfacshr}_{\text{reported date}}}{\text{cfacshr}_{\text{desired date}}}
\]

Suppose the fund held 2000 shares of Adobe (PERMNO 75510) on January 1, 2005 and 4500 shares on December 31, 2005.

If I say
proc print data=crsp.msf;
    where permno=75510 and date>='01Dec2004'd and date<=’31Dec2005’d;
    var permno date ret prc cfacshr;
runit;

SAS produces:

<table>
<thead>
<tr>
<th>PERMNO</th>
<th>DATE</th>
<th>RET</th>
<th>PRC</th>
<th>CFACSHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>75510</td>
<td>20041231</td>
<td>0.036204</td>
<td>62.74000</td>
<td>2</td>
</tr>
<tr>
<td>75510</td>
<td>20050131</td>
<td>-0.093083</td>
<td>56.90000</td>
<td>2</td>
</tr>
<tr>
<td>75510</td>
<td>20050228</td>
<td>0.085237</td>
<td>61.75000</td>
<td>2</td>
</tr>
<tr>
<td>75510</td>
<td>20050331</td>
<td>0.087976</td>
<td>67.17000</td>
<td>2</td>
</tr>
<tr>
<td>75510</td>
<td>20050429</td>
<td>-0.114634</td>
<td>59.47000</td>
<td>2</td>
</tr>
<tr>
<td>75510</td>
<td>20050531</td>
<td>0.113839</td>
<td>33.12000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20050630</td>
<td>-0.136171</td>
<td>28.61000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20050729</td>
<td>0.036001</td>
<td>29.64000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20050831</td>
<td>-0.087719</td>
<td>27.04000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20050930</td>
<td>0.103920</td>
<td>29.85000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20051031</td>
<td>0.080402</td>
<td>32.25000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20051130</td>
<td>0.011163</td>
<td>32.61000</td>
<td>1</td>
</tr>
<tr>
<td>75510</td>
<td>20051230</td>
<td>0.133395</td>
<td>36.96000</td>
<td>1</td>
</tr>
</tbody>
</table>

Apparently, there was a 2-for-1 stock split between the end of April and the end of May. The 2000 shares in January would have become 4000 shares in December. The formula above yields:

shares in January adjusted to December=shares in January*January cfacshr/December cfacshr

Or

shares in January adjusted to December=shares in January*2/1

Or

shares in January adjusted to December=2000*2/1=4000

Therefore the increase in shares is 500 “December shares”. The value of this trade, assuming it happened in December, is 500*36.96, which is the December price.

6.5 Merging CRSP and Compustat

7 Working with time-series data and dates

7.1 Moving dates around

A simplistic way to work with dates with monthly data in SAS is to convert all dates to a year and month, and then use those. For instance

    data msf;
    set crsp.msf(keep=permno date ret);
    year=year(date);
    month=month(date);
run;
proc sort data=msf;
    by permno year month;
run;

proc means data=msf;
    by permno year;
var ret;
run;

This is appealing when, say, you have to calculate within-year averages, but it can quickly get tedious if you have to do other simple tasks. For instance, suppose you want to create a series of *daily* dates from January 1, 1961 to December 31, 1962.

data dates;
do year=1961 to 1962;
do month=1 to 12;
    lengthofmonth=31;
        if month=4 or month=6 or month=9 or month=11 then lengthofmonth=30;
        if month=2 then lengthofmonth=28;
    do day=1 to lengthofmonth;
        output;
    end;
end;
end;
run;

This is terribly tedious - not to mention fraught with the danger that you'll make errors in leap years. The alternative is to use SAS' intnx function, which increments dates by specified intervals.

data dates;
    date='01Jan1961'd;
do while(date<='31Dec1962'd);
    output;
    date=intnx('day', date, 1);
end;
run;

The intnx function increments date (its second argument), by 1 unit (the third argument) of 'day' (its first argument).

The more astute among you will observe that I get the same effect by saying

data dates;
    date='01Jan1961'd;
do while(date<='31Dec1962’d);
    output;
    date=date+1;
end;
run;

because dates are treated as numbers in SAS. This is well and good if I want to increment by single days. Suppose instead I want to increment by weekdays. That is, I want the list of all weekdays in 1961 and 1962. An easy way to do this is:

    data dates;
        date='01Jan1961’d;
    do while(date<='31Dec1962’d);
        output;
        date=intnx(’weekday’, date, 1);
    end;
    run;

which is a useful construct in working with the DSF file, and horrendously complicated to do any other way.

The intnx function, as already said, increments dates, but it does so in a nonintuitive way. Suppose you are incrementing by year.

    date=intnx(’year’, date, 1);

Intnx moves the date to the first day of the subsequent year, not forward by one twelve-month period. If you say ’month’ instead of ’year’, intnx moves the date forward to the first day of the next month however many times you specify. Saying

    date=intnx(’month’, date, 0);

moves the date to the beginning of the month - the first of the month- in which date falls. Similarly for ’qtr’.

I commonly use the intnx function to do the following:

• Move the date to the end of the month/quarter/year

    date=intnx(’month’, date, 1)-1;
    date=intnx(’qtr’, date, 1)-1;
    date=intnx(’year’, date, 1)-1;

• Move the date to the beginning of the month/quarter/year
Move the date to the end of the previous month

\[ \text{date}=\text{intnx('month', date, 0)}-1; \]

Move the date (which is an end-of-month) to the end of the next month

\[ \text{date}=\text{intnx('month', date+1, 1)}-1; \]

Observe here that I have to say date+1. Since the date is currently of the form 31Jan1999, saying

\[ \text{date}=\text{intnx('month', date, 1)}-1; \]

will leave us with 01Feb1999-1, or 31Jan1999.

Move the date to the end of month 12 months ago

\[ \text{date}=\text{intnx('month', date, -11)}-1; \]

Move the date (which is an end-of-month) to the end of the 12 months ahead

\[ \text{date}=\text{intnx('month', date+1, 12)}-1; \]

Given a beginning date and an ending date, create a dataset with all the month-ends in between:

```plaintext
data beg_end;
  beginning_date='31Jan1996'd;
  ending_date='31Dec2007'd;
run;
data all_dates;
  set beg_end;
  date= beginning_date;
  date=intnx('month', date, 1)-1;
do while(date<=ending_date);
  output;
  date=intnx('month', date+1, 1)-1;
end;
run;
```
The first

\[
\text{date}=\text{intnx('month', date, 1)}-1;
\]

statement is to ensure that the first date is an end-of-month.

- Produce a list of all Wednesdays between two dates:

```sas
data beg_end;
    beginning_date='31Jan1996'd;
    ending_date='31Dec2007'd;
run;

data all_dates;
    format date date9.;
    set beg_end;
    date=beginning_date;
    if weekday(date)>4 then date=intnx('week', date, 1)+3;
    else date=intnx('week', date, 0)+3;
    do while(date<=ending_date);
        output;
        date=intnx('week', date, 1)+3;
    end;
run;
```

The weekday() function returns the day of the week, 1 being Sunday. I make the first date a Wednesday (weekday=4) with

\[
\text{if weekday(date)>4 then date}=\text{intnx('week', date, 1)}+3;
\]

The logic is simple. If the current date is “more” than a Wednesday (ie, weekday(date)>4), then the first admissible date is Wednesday of next week. Sunday of next week is

\[
\text{date}=\text{intnx('week', date, 1)};
\]

Add 3 to that to get to Wednesday. If the current date is “less than or equal to” Wednesday, then go to the beginning of this week

\[
\text{date}=\text{intnx('week', date, 0)};
\]

and add 3 to that.

The rest of the scheme is obvious. This is useful in constructing weekly returns (Thursday-to-Wednesday to avoid any weekend effect).
7.2 Avoiding the lag() function

A common task in time-series data is to find the lag of a variable: that is, the value the variable took in the previous period. Suppose I wanted to find the value of return in the previous month for every permno at every month-end. I could do this by saying:

```sas
data msf;
  set crsp.msf(keep=permno date ret);
  date=intnx('month', date, 1)-1;
  lagret=lag(ret);
run;
```

If you're puzzled by the

```sas
date=intnx('month', date, 1)-1;
```

statement, please see section 6.1 for an explanation.

One problem with this is that the first value of lagret for each permno is set to the last value of ret for the previous permno. To avoid this, I write

```sas
data msf;
  set crsp.msf(keep=permno date ret);
by permno;
  date=intnx('month', date, 1)-1;
  lagret=lag(ret);
if first.permno then lagret=.;
run;
```

Observe that I did not write:

```sas
data msf;
  set crsp.msf(keep=permno date ret);
by permno;
  date=intnx('month', date, 1)-1;
  if not(first.permno) then lagret=lag(ret);
run;
```

which, at first glance, seems equivalent. The problem is that the lag() function pushes the current variable value onto a stack every time it is called, and returns that value the next time it is called. This means that if the lag() function is not called on every observation, it will give counterintuitive results. In the code just shown, lag() is not being called for the first observation of each permno. Using the lag() function poses another problem. Suppose the value of return is missing in a particular month, as for July 1986 for permno 10000 below:
<table>
<thead>
<tr>
<th>PERMNO</th>
<th>DATE</th>
<th>RET</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>19860228</td>
<td>-0.257143</td>
</tr>
<tr>
<td>10000</td>
<td>19860331</td>
<td>0.365385</td>
</tr>
<tr>
<td>10000</td>
<td>19860430</td>
<td>-0.098592</td>
</tr>
<tr>
<td>10000</td>
<td>19860531</td>
<td>-0.222656</td>
</tr>
<tr>
<td>10000</td>
<td>19860630</td>
<td>-0.005025</td>
</tr>
<tr>
<td>10000</td>
<td>19860831</td>
<td>-0.615385</td>
</tr>
<tr>
<td>10000</td>
<td>19860930</td>
<td>-0.057143</td>
</tr>
<tr>
<td>10000</td>
<td>19861031</td>
<td>-0.242424</td>
</tr>
<tr>
<td>10000</td>
<td>19861130</td>
<td>0.060000</td>
</tr>
<tr>
<td>10000</td>
<td>19861231</td>
<td>-0.377358</td>
</tr>
<tr>
<td>10000</td>
<td>19870130</td>
<td>-0.212121</td>
</tr>
<tr>
<td>10000</td>
<td>19870228</td>
<td>0.000000</td>
</tr>
<tr>
<td>10000</td>
<td>19870331</td>
<td>-0.384615</td>
</tr>
<tr>
<td>10000</td>
<td>19870430</td>
<td>-0.062500</td>
</tr>
<tr>
<td>10000</td>
<td>19870531</td>
<td>-0.066667</td>
</tr>
<tr>
<td>10001</td>
<td>19860228</td>
<td>0.020408</td>
</tr>
<tr>
<td>10001</td>
<td>19860331</td>
<td>0.025200</td>
</tr>
<tr>
<td>10001</td>
<td>19860430</td>
<td>0.009901</td>
</tr>
<tr>
<td>10001</td>
<td>19860531</td>
<td>-0.009804</td>
</tr>
</tbody>
</table>

This would obviously create trouble for the approach above. One way to handle this would be to say:

```r
data msf;
  set crsp.msf(keep=permno date ret);
  by permno;
    date=intnx('month', date, 1)-1;
    lagret=lag(ret);
    lagdate=lag(date);
    if first.permno then lagret=.;
    if not(lagdate=intnx('month', date, 0)-1) then lagret=.;
run;
```

which checks to see if the lagged date is indeed a month previous to the current date. This is inelegant, and difficult to extend, if, for instance, you have several variables to lag. Instead, do the following:

```r
data msf;
  set crsp.msf(keep=permno date ret);
  date=intnx('month', date, 1)-1;
run;

data msflag(rename=(ret=lagret));
```

18
set msf;
date=intnx('month', date+1, 1)-1;
run;

data msf;
merge msf msflag;
by permno date;
run;

This creates a new dataset msflag with all variables identical, except that the date is moved forward one month. This means that, in msflag, the return for January 1986 for each permno is going to have the date set to February of 1986. When you merge msflag with msf (by permno and date), the returns line up exactly as you want them to. Explicitly, msflag looks like this:

<table>
<thead>
<tr>
<th>PERMNO</th>
<th>DATE</th>
<th>LAGRET</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>19860331</td>
<td>-0.257143</td>
</tr>
<tr>
<td>10000</td>
<td>19860430</td>
<td>0.365385</td>
</tr>
<tr>
<td>10000</td>
<td>19860531</td>
<td>-0.098592</td>
</tr>
<tr>
<td>10000</td>
<td>19860630</td>
<td>-0.222656</td>
</tr>
<tr>
<td>10000</td>
<td>19860731</td>
<td>-0.005025</td>
</tr>
<tr>
<td>10000</td>
<td>19860930</td>
<td>-0.615385</td>
</tr>
<tr>
<td>10000</td>
<td>19861031</td>
<td>-0.057143</td>
</tr>
<tr>
<td>10000</td>
<td>19861130</td>
<td>-0.242424</td>
</tr>
<tr>
<td>10000</td>
<td>19861231</td>
<td>0.060000</td>
</tr>
<tr>
<td>10000</td>
<td>19870131</td>
<td>-0.377358</td>
</tr>
<tr>
<td>10000</td>
<td>19870228</td>
<td>-0.212121</td>
</tr>
<tr>
<td>10000</td>
<td>19870331</td>
<td>0.000000</td>
</tr>
<tr>
<td>10000</td>
<td>19870430</td>
<td>-0.384615</td>
</tr>
<tr>
<td>10000</td>
<td>19870531</td>
<td>-0.062500</td>
</tr>
<tr>
<td>10000</td>
<td>19860228</td>
<td>-0.066667</td>
</tr>
<tr>
<td>10001</td>
<td>19860331</td>
<td>0.020408</td>
</tr>
<tr>
<td>10001</td>
<td>19860430</td>
<td>0.025200</td>
</tr>
<tr>
<td>10001</td>
<td>19860531</td>
<td>0.009901</td>
</tr>
<tr>
<td>10001</td>
<td>19860630</td>
<td>-0.009804</td>
</tr>
</tbody>
</table>

This handles all the exceptions you can think of. The merged dataset is below:

<table>
<thead>
<tr>
<th>Obs</th>
<th>PERMNO</th>
<th>DATE</th>
<th>RET</th>
<th>lagret</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
<td>19860228</td>
<td>-0.257143</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>19860331</td>
<td>0.365385</td>
<td>-0.257143</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>19860430</td>
<td>-0.098592</td>
<td>0.365385</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>19860531</td>
<td>-0.222656</td>
<td>-0.098592</td>
</tr>
</tbody>
</table>
This will not work with daily data. First, Friday is not one day before Monday, but that can be handled by using `weekday` as the incrementing unit in `intnx`. Second, holidays make any systematization impossible.

For daily data, you need another method. Suppose I want to find lag returns in the DSF file, that is, the return on the previous trading day.

The way I do it is to first create a dataset that contains, for every date in DSF, the date of the previous trading day. Then I merge this dataset into DSF by DATE, creating a dataset MAIN, which therefore has a LAGDATE associated with each DATE. Then I re-merge DSF with MAIN, but by PERMNO LAGDATE, so that the values of RET being merged into MAIN are the lagged values.

First create a file with all the dates in the dataset, and create a lagdate:

```sql
proc sort data=crsp.dsf(keep=date) out=dsfdates nodupkey;
  by date;
run;

data dsfdates;
  set dsfdates;
  lagdate=lag(date);
run;
```

Then merge that into DSF, by date:

```sql
proc sort data=crsp.dsf(keep=permno date ret) out=dsf;
  by date;
run;
```
data main;
    merge dsf dsfdates;
    by date;
run;

MAIN now has PERMNO DATE RET and LAGDATE. I need to get LAGRET. To do this, I merge DSF into MAIN, but by PERMNO LAGDATE, not PERMNO DATE:

data lagdsf(rename=(date=lagdate ret=lagret));
    set crsp.dsf(keep=permno date ret);
run;

/*NOTE: DSF is already sorted by permno date*/

data main;
    merge main(in=a) lagdsf;
    by permno lagdate;
run;

This is horribly inefficient, because you have to sort DSF by date, which takes 3 minutes at the best of times, but it serves to illustrate the point.

7.3 Rolling regressions without macros

Boehmer, Broussard, and Kallunki (2002) recommend using macros to run rolling regressions. While macros make impossible tasks possible, they aren’t particularly efficient. I describe here a macro-independent way of running rolling regressions, and doing similar tasks.

The task is to run rolling 24-month regressions of monthly stock excess returns on the Fama-French-Carhart factors. That is, at the end of each month, I need to:

- Get the last 24 months of excess returns for each stock, checking that there are at least, say, 12 months of data available.
- Get the Fama-French-Carhart factors.
- Run rolling regressions

I first create a dataset containing “rankdates”, which are the date identifiers for the rolling regression. A rankdate of 31Dec2001, for instance, uses data from 31Jan2000 to 31Dec2001, inclusive.

To do this, I first get the first and last date for each permno, and then create a complete list by filling in the in-between dates. I do this because if I were to just use the permno-date pairs available in MSF, if a return is missing for a particular month, then that month will not be a rankdate, even though it might be perfectly valid to make that date a rankdate.
data firstandlastdates;
  set crsp.msf(keep=permno date);
  by permno; /*MSF is always sorted by permno date*/
  retain firstdate;
  date=intnx('month', date, 1)-1;
  if first.permno then firstdate=date;
  if last.permno then do;
    lastdate=date;
  output;
  end;
run;

data permnosrankdates(rename=(date=rankdate));
  set firstandlastdates;
  date=firstdate;
  do while(date<=lastdate);
    output;
    date=intnx('month', date+1, 1)-1;
  end;
run;

For each rankdate, I then get the list of the 24 dates from which that rankdate will use data.

data permnosrankdates;
  set permnosrankdates;
  date=rankdate;
  i=1;
  do while(i<=24);
    output;
    date=intnx('month', date, 0)-1;
  i=i+1;
  end;
run;

*permnosrankdates* is a dataset that looks, in part, like this:

<table>
<thead>
<tr>
<th>Obs</th>
<th>PERMNO</th>
<th>rankdate</th>
<th>date</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
<td>19851231</td>
<td>31DEC1985</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>19851231</td>
<td>30NOV1985</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>19851231</td>
<td>31OCT1985</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>19851231</td>
<td>30SEP1985</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>19851231</td>
<td>31AUG1985</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>10000</td>
<td>19851231</td>
<td>31JUL1985</td>
<td>6</td>
</tr>
</tbody>
</table>
We don’t need to keep i, but I kept it for clarity. Once we have this, all we need to do is merge it with the factors and the returns:

data ff;
  set ff.factors_monthly(keep=date rf smb hml umd mktrf);
  date=intnx('month', date, 1)-1;
run;

proc sort data=permnosrankdates;
  by date permno;

data permnosrankdates;
  merge permnosrankdates(in=a) ff(in=b);
  by date;
  if a and b;
run;

data msf;
  set crsp.msf(keep=permno date ret);
  where ret is not missing;
  date=intnx('month', date, 1)-1;
run;

proc sort data=msf;
   by date permno;
run;

/*/permnosrankdates is already sorted*/

data permnosrankdates;
   merge permnosrankdates(in=a) msf(in=b);
   by date permno;
   if a and b;
run;

Notice that I merged by date, not rankdate.
And now all that remains is to calculate excess returns and run the regressions:

data permnosrankdates;
   set permnosrankdates;
   exret=ret-rf;
run;

proc sort data=permnosrankdates;
   by permno rankdate;
run;

proc reg data=permnosrankdates outest=est edf;
   by permno rankdate;
   model exret=mktrf smb hml umd;
run;

Notice I run proc reg by rankdate, not date.
Running proc print on EST yields the following output:

<table>
<thead>
<tr>
<th>Obs</th>
<th>PERMNO</th>
<th>rankdate</th>
<th><em>MODEL</em></th>
<th><em>TYPE</em></th>
<th><em>DEPVAR</em></th>
<th><em>RMSE</em></th>
<th>Intercept</th>
<th>mktrf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
<td>19860228</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>.</td>
<td>-0.2624</td>
<td>0.000</td>
</tr>
<tr>
<td>2</td>
<td>10000</td>
<td>19860331</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>.</td>
<td>1.9027</td>
<td>-32.219</td>
</tr>
<tr>
<td>3</td>
<td>10000</td>
<td>19860430</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>.</td>
<td>1.4644</td>
<td>-31.110</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
<td>19860531</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>.</td>
<td>4.3923</td>
<td>2.102</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>19860630</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>.</td>
<td>-14.0666</td>
<td>-191.074</td>
</tr>
<tr>
<td>6</td>
<td>10000</td>
<td>19860731</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>0.48716</td>
<td>-0.1872</td>
<td>-4.129</td>
</tr>
<tr>
<td>7</td>
<td>10000</td>
<td>19860831</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>0.34479</td>
<td>-0.1559</td>
<td>-2.784</td>
</tr>
<tr>
<td>8</td>
<td>10000</td>
<td>19860930</td>
<td>MODEL1</td>
<td>PARMS</td>
<td>exret</td>
<td>0.29117</td>
<td>-0.1216</td>
<td>-0.492</td>
</tr>
</tbody>
</table>
### Obs smb hml umd exret _IN_ _P_ _EDF_ _RSQ_

<table>
<thead>
<tr>
<th>Obs</th>
<th>smb</th>
<th>hml</th>
<th>umd</th>
<th>exret</th>
<th><em>IN</em></th>
<th><em>P</em></th>
<th><em>EDF</em></th>
<th><em>RSQ</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.00</td>
<td>0.00</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>0.000</td>
<td>0.00</td>
<td>0.00</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>3</td>
<td>-71.326</td>
<td>0.00</td>
<td>0.00</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>4</td>
<td>340.707</td>
<td>477.88</td>
<td>0.00</td>
<td>-1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>5</td>
<td>-884.257</td>
<td>-1322.51</td>
<td>506.315</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>6</td>
<td>-13.672</td>
<td>-16.38</td>
<td>7.106</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>0.05020</td>
</tr>
<tr>
<td>7</td>
<td>-8.529</td>
<td>-10.71</td>
<td>5.470</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>0.54571</td>
</tr>
<tr>
<td>8</td>
<td>7.327</td>
<td>3.24</td>
<td>4.754</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>0.51848</td>
</tr>
<tr>
<td>9</td>
<td>7.303</td>
<td>3.22</td>
<td>4.751</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>0.52996</td>
</tr>
<tr>
<td>10</td>
<td>3.312</td>
<td>-0.84</td>
<td>3.657</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>0.42145</td>
</tr>
<tr>
<td>11</td>
<td>4.391</td>
<td>0.97</td>
<td>3.748</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0.30088</td>
</tr>
<tr>
<td>12</td>
<td>4.378</td>
<td>0.91</td>
<td>3.725</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>0.30569</td>
</tr>
<tr>
<td>13</td>
<td>4.342</td>
<td>-0.04</td>
<td>3.353</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0.31433</td>
</tr>
<tr>
<td>14</td>
<td>0.837</td>
<td>-3.20</td>
<td>2.548</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>0.26778</td>
</tr>
<tr>
<td>15</td>
<td>0.669</td>
<td>-3.35</td>
<td>2.531</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>0.27495</td>
</tr>
<tr>
<td>16</td>
<td>0.604</td>
<td>-3.41</td>
<td>2.454</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>0.27148</td>
</tr>
<tr>
<td>17</td>
<td>0.604</td>
<td>-3.41</td>
<td>2.454</td>
<td>-1</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>0.27148</td>
</tr>
</tbody>
</table>

We can tell how many observations we have for each date by looking at the _EDF_: the error degrees of freedom. We asked for this by including edf in the proc model statement. If we want to keep only regressions for which there were 12 observations or more, we keep those where _EDF_>7.

EST is the rolling regression output dataset.

This technique has wide application: rolling regressions, obviously, but also rolling standard deviations, moving averages and whatnot.

### 8 Missing values

In procedures: means ( FREQ, N), freq In additions: the sum() function
9 Code debugging

9.1 Run on a small subset of observations, making sure your code doesn’t overwrite anything

There are two ways to run your code on a small subset of observations this. One is to identify the largest dataset in your program and to use the “obs=” dataset option.

```sas
data msf;
   set crsp.msf(keep=permno date ret obs=2000);
   date=intnx('month', date, 1)-1;
run;
```

The other is to set the global “obs=” option. This is done by putting the following statement in open code: i.e., not within a data or proc step.

```sas
options obs=2000;
```

What this does is to read a maximum of 2000 observations from any dataset the program encounters. It’s quick to put this at the head of your program and run it to check for coding errors. This will find, for instance, variables you have inadvertently left out and referred to later, but other errors may not show up. For example, if observations in a data or proc step that uses by-variables were not properly sorted, you might not know because the particular subset that you run the code with happens to be properly sorted.

To prevent your code from overwriting datasets while you’re debugging it, use the “nreplace” option, which prevents overwriting of existing datasets:

```sas
options obs=2000 noreplace;
```

9.2 See what the macro sees

Useful options (placed in open code) to run macro code with are

```sas
options symbolgen mprint mlogic;
```

which cause the program to print the SAS code that the macro generates to the log. You are apparently not supposed to use these in actual runs: they slow the program down. To turn these off say

```sas
options nosymbolgen nomprint nomlogic;
```
9.3 Set msglevel=i

9.4 Print variables to the log within the data step

At any point in the data step you can say

    put <variable-list>

and the observations of the variables in variable-list will be written to the log. You can also use this in a conditional:

    data msf;
    set crsp.msf(obs=200);
    if permno>10006 then put permno date ret;
    run;

The “obs=200” is so that my log file does not get huge.

9.5 Using the DEBUG option

10 Macros

10.1 Macro variables

10.1.1 Subsetting observations from large datasets efficiently

In the section on efficiency (Section 14), I advocate using where statements to subset datasets before reading them in. For instance, say

    data dsf;
    set crsp.dsf(keep=permno date ret);
    where permno in(10001, 10002, 10003);
    run;

instead of

    data dsf;
    set crsp.dsf(keep=permno date ret);
    if permno=10001 or permno=10002 or permno= 10003;
    run;

What happens when you do not know in advance what permnos you want to keep? Or if you do know, but the list is a headache to type out?

In that case, you can replace the contents of the where with a macro variable. Suppose some previous steps produce a dataset, named KEEPTHESE that contains the permnos you want to keep:

27
Given this dataset, I want to subset CRSP.DSF, keeping only the permnos listed in KEEPTHESE. To do this, I first run the following data step, which creates several macro variables.

```
data _null_;  
  set keepthese nobs=nobs;  
  if _n_ =1 then call symput("nobs", nobs);  
    call symput("permname"||trim(left(put(_N_,4.))), permno);  
  run;
```

The first line after the SET statement creates a macro variable named NOBS, which contains the number of observations in KEEPTHESE. The second line creates a series of macro variables named `permname1`, `permname2`, `permname3`, ... containing the permnos in the order in which they appeared in KEEPTHESE. Now inside a macro, I concatenate all these macro variables.

```
%macro justconcatenate;  
  %let listofpermnos=&permname1;  
  %do i =2 %to &nobs;  
    %let listofpermnos =%sysfunc(catx(%str(,),&listofpermnos, "%trim(&&permname&i)"));  
  %end;  
  %let listofpermnos=(&listofpermnos);  
%end;
%justconcatenate;
```

This macro just creates a macro variable named LISTOFPERMNOS whose value is set to the contents of the macro variable PERMNAME1. I then join all the remaining PERMNAMEs to this LISTOFPERMNOS with commas as delimiters.

Finally, I can use the macro variable so created:

```
data dsf;  
  set crsp.dsf(keep=permno date ret);  
  where permno in &listofpermnos;  
  run;
```

This might seem like a lot of bother for nothing, but supposing you are work with a large dataset, like TAQ, and you need to keep observations with symbols in a dataset like KEEPTHESE. Sorting and merging is not an option. This method works well, as would hashes (see section [12]). I have not compared this method against a hash.
10.2 Loops

Macro loops allow you to %DO something for an index in some range of integers. For instance, you can write

```sas
%macro justaloop;
   %do i = 1 %to 200;
      <some code>
   %end;
%justaloop;
```

and some code will be run for all values of &j.

10.2.1 Counters are dates

A useful thing would be to run a macro loop with a changing date. It would be great if I could say

```sas
%macro justaloop;
   %do date = '31Jan1980'd %to '31Jan2000'd;
      <some code>
   %end;
%mend;
%justaloop;
```

This is invalid because only integer values can be supplied as index values. SAS thinks that you’re trying to loop starting with &date set to be the string ‘31Jan1980’d.

One option is to get the numeric version of the starting and ending dates and then use those in the loop:

```sas
%let startdate= '31Jan1980'd;
%let enddate= '31Jan2000'd;

%macro justaloop;
   %let startdate=%sysfunc(putn(&startdate, 8.));
   %let enddate=%sysfunc(putn(&enddate, 8.));
   %do date = &startdate %to &enddate;
      <some code>
   %end;
%mend;
%justaloop;
```

The statement

```sas
%let startdate=%sysfunc(putn(&startdate, 8.));
```
converts the contents of &STARTDATE from '31Jan1980'd to 7335. The %sysfunc macro function allows you to use DATA step functions in macro code. In this case, it allows me to use the PUTN function.

Using a %put statement to tell me how &STARTDATE changes (with options symbolgen on: see Section 9.2):

%let startdate= '31Jan1980'd;
%let startdate=%sysfunc(putn(&startdate, 8.));
%put startdate is &startdate;

the following is written to the log:

%let startdate= '31Jan1980'd;
%let startdate=%sysfunc(putn(&startdate, 8.));
SYMBOLGEN: Macro variable STARTDATE resolves to '31Jan1980'd
%put startdate is &startdate;
SYMBOLGEN: Macro variable STARTDATE resolves to 7335
startdate is 7335

The loop above does whatever some code does for each day between the two dates. However, in this loop I cannot adjust the interval by which the date counter is incremented. For instance, I might want to do something for each end-of-month between startdate and enddate.

To do that, I don’t increment the date by 1, instead I increment it manually using a %SYSFUNC call to INTNX:

%let startdate= '31Jan1980'd;
%let enddate= '31Jan2000'd;
%macro justaloop;
%let startdate=%sysfunc(putn(&startdate, 8.));
%let enddate=%sysfunc(putn(&enddate, 8.));
%let date=&startdate;
%do %while(&date <= &enddate);
   <some code>
   %let date=%eval(%sysfunc(intnx(month, &date+1, 1))-1);
%end;
%mend;
%justaloop;

Another option is to simply do it by year and month and use the mdy() function to convert it into a date as needed:

%macro justaloop;
This is not very flexible.

10.2.2 %FOR variable in &LIST

I often want to provide a list of variables, say, to a macro and have it run for each of them, instead of cluttering up my code with several invocations of the same macro for each of the variables.

Suppose I want to run the macro %calculate_means for three variables, ret, retx and shrout. I could say:

%calculate_means(ret);
%calculate_means(retx);
%calculate_means(shrout);

Or I could write a foreach macro:

%macro foreach(listofvariables);

%let i=1;
%do %while(%scan(&listofvariables, &i, ', '>0));
  %let thevar=%scan(&listofvariables, &i, ', ');
  %put We have the variable &thevar;
  %calculate_means(&thevar);
  %let i=%eval(&i+1);
%end;
%mend;

And call it with

%foreach(ret retx shrout);
Suppose you want to calculate the average correlation between stocks in each month in your sample. Calculating correlations means you will have a dataset with N-squared elements, where N is the number of stocks in that month. To do this, you write a macro loop (see section 10.2.1):

```
%let startdate= '31Jan1980'd;
%let enddate= '31Jan2000'd;

%macro justaloop;
%let startdate=%sysfunc(putn(&startdate, 8.));
%let enddate=%sysfunc(putn(&enddate, 8.));

%let date=&startdate;
%do %while(&date <= &enddate);
   data tempdsf;
   set crsp.dsf;
   where intnx('month', date, 1)-1 = &date;
   run;
<code to calculate correlations>
   %let date=%eval(%sysfunc(intnx(month, &date+1, 1))-1);
%end;
%mend;
%justaloop;
```

This is inefficient because it takes long to subset DSF each time. Instead, do the following:

1. Outside the loop, sort DSF by date
2. Outside the loop, create an “index” dataset that tells you for each month where the data for that month begins and ends in the sorted DSF
3. Within the loop, use FIRSTOBS= and OBS= to read only the required observations

Sorting DSF and creating an index are simple:

```
proc sort data=crsp.dsf out=dsf;
   where date>=&startdate and date<=&enddate;
   by date;
run;

data dsfindex(keep=date firstobs lastobs);
   set dsf;
```

32
DSFINDEX now contains, for each date, the observation numbers at which the observations for that date begin and end. Within the loop, you then say:

```plaintext
data _null_;  
set dsfindex;  
where date=&date;  
call symput('firstobs', firstobs);  
call symput('lastobs', lastobs);  
run;
```

```plaintext
data tempdsf;  
set dsf(firstobs=&firstobs obs=&lastobs);  
run;
```

The DATA _NULL_ statement just means that I am not creating a new dataset when I subset DSFINDEX. The only reason I am opening DSFINDEX is to get the values of FIRSTOBS and LASTOBS into the corresponding macro variables. I then subset DSF to get TEMPDSF, keeping only the observations numbered between &FIRSTOBS and &LASTOBS. Note that the obs= statement tells you the last observation that will be read, not the number of observations to read.

TEMPDSF now contains all the observations for which DATE equals &DATE. The savings here are enormous: this takes virtually no time.

The complete code thus reads:

```plaintext  
%let startdate= '31Jan1980'd;  
%let enddate= '31Jan2000'd;
```

```plaintext  
proc sort data=crsp.dsf out=dsf;  
   where date>=&startdate and date<=&enddate;  
   by date;
```
data dsfindex(keep=date firstobs lastobs);
  set dsf;
  by date;
  retain firstobs;
  if first.date then do;
    firstobs=_n_; 
  end;
  if last.date then do;
    lastobs=_n_; 
    output;
  end;
run;
%macro justaloop;
  %let startdate=%sysfunc(putn(&startdate, 8.));
  %let enddate=%sysfunc(putn(&enddate, 8.));

  %let date=&startdate;
  %do %while(&date <= &enddate);
    data _null_; 
      set dsfindex;
      where date=&date;
      call symput('firstobs', firstobs);
      call symput('lastobs', lastobs);
    run;
    data tempdsf;
      set dsf(firstobs=&firstobs obs=&lastobs);
    run;
    <code to calculate correlations>
      %let date=%eval(%sysfunc(intnx(month, &date+1, 1))-1);
    %end;
  %mend; 
%justaloop;

10.3 Writing frequently-used code as macros

Several operations are so standard that if you write them up as macros and save them in a macro file, you will be spared a lot of typing. You can then automatically include this macro file in all your SAS programs - see the section on autoexec.sas, Section [15.9]

An example is the merge algorithm. Merging always proceeds in the same way. You are given two datasets and the by-variables, you sort them by the by-variables and then merge them. I wrote
up a “standard merge” macro named %stdmerge:

```sas
%macro stdmerge(inds1, inds2, byvar, outds);
   proc sort data=&inds1 nothreads;
      by &byvar;
   run;

   proc sort data=&inds2 nothreads;
      by &byvar;
   run;

   data &outds;
      merge &inds1(in=a) &inds2(in=b);
      by &byvar;
      if a and b;
   run;
%mend;
```

In the SAS programs I write, when I want to merge two datasets, all I need to do is say

```sas
%stdmerge(firstdataset, seconddataset, by-variable-list, outputdataset)
```

instead of laboriously typing out the whole thing every time. This clears up an enormous amount of clutter from my SAS programs, making them much more readable.

### 10.3.1 Merging macros

%stdmerge you have already met. The others just have varying “if” conditions.

- %kbmerge: “Keep both” merge keeps an observation if it is in either of the input datasets. In addition, it creates variables that tell you which dataset an observation came from:

```sas
%macro kbmerge(inds1, inds2, byvar, outds);
   proc sort data=&inds1 nothreads;
      by &byvar;
   run;

   proc sort data=&inds2 nothreads;
      by &byvar;
   run;

   data &outds;
      merge &inds1(in=a) &inds2(in=b);
      by &byvar;
      if a and b;
   run;
%mend;
```
• %kfmerge: “keep first” merge keeps only observations from the first dataset. This is just
%stdmerge with the “if” line replaced.

```sas
%macro kfmerge(inds1, inds2, byvar, outds);
    proc sort data=&inds1 nothreads;
        by &byvar;
    proc sort data=&inds2 nothreads;
        by &byvar;
    data &outds;
        merge &inds1(in=a) &inds2(in=b);
        by &byvar;
        if a;
    run;
%mend;
```

• %rmsecmerge: “remove second” merge removes the observations from the first dataset that
occur in the second. The only difference again is the if line:

```sas
%macro rmsecmerge(inds1, inds2, byvar, outds);
    proc sort data=&inds1 nothreads;
        by &byvar;
    proc sort data=&inds2 nothreads;
        by &byvar;
    data &outds;
        merge &inds1(in=a) &inds2(in=b);
        by &byvar;
        if not b;
    run;
%mend;
```

10.3.2 Hash-merging without having to type

See section 12.2 for how to use hashes to efficiently merge small datasets into large ones. Here I
describe a macro that does it without typing.

This macro consists of three separate macros. The two small ones are called %enquote and
%sepbycommas. They accept space-separated lists of variable names and print out, in the first
case, that list with double-quotes around each name, separated by commas, and, in the second
case, just the variable names separated by commas. If you provide
to the first macro, it will return

```
'permno',''date',''ret'
```

And the second will return

```
permno,date,ret
```

The macro that does the actual work, `%hashmerge`, just prints out the data step that does the hash merge.

```sas
%macro hashmerge(largeds, smallds, byvars, extravars, outds);
  data &outds;
    call missing(%sepbycomma(&extravars));
    if _n_=1 then do;
      declare hash h(dataset:''&smallds'');
```

```sas
%macro sepbycomma(varlist);
  %let delim=%str( );

  %let outputvar=%qscan(&varlist,1,&delim);
  %let i=2;
  %do %while(%length(%qscan(&varlist,&i,&delim)) GT 0);
    %let varhere=%qscan(&varlist,&i,&delim);
    %let outputvar=&outputvar,&varhere;
    %let i=%eval(&i + 1);
  %end;
  &outputvar
%mend;

%macro enquote(varlist);
  %let delim=%str( );

  %let outputvar=''%qscan(&varlist,1,&delim)'';
  %let i=2;
  %do %while(%length(%qscan(&varlist,&i,&delim)) GT 0);
    %let varhere=%qscan(&varlist,&i,&delim);
    %let outputvar=&outputvar,''&varhere'';
    %let i=%eval(&i + 1);
  %end;
  &outputvar
%mend;
```
10.3.3 Checking for duplicate by-variable combinations

Before a merge, I frequently want to check that there is only one instance of every by-variable combination in one or both of my datasets. If I am doing what I think is a one-to-one merge, then obviously each dataset should contain only one instance of every by-variable combination. If I am doing a many-to-one merge, then one dataset should have only one instance of every by-variable combination.

(Alternatively, suppose I am merging two datasets using a fuzzy merge variable. After the merge is done, I want to see which observations in one dataset were matched to a single observation in the other, and which were matched to two or more.)

Suppose I have a mutual fund holdings dataset that only contains, as stock information, the ticker of the stock. That is, it contains

\begin{verbatim}
CRSP_FUNDNO DATE STOCKTICKER SHARES_HELD
\end{verbatim}

I need to get the permno for these stocks. To do this, I first create a permno-date-ticker file as described in Section 6.2.

\begin{verbatim}
data stocknames(keep=permno date ticker);
  set crsp.stocknames(keep=namedt nameenddt permno ticker);
  format date date9.;
  where namedt is not missing;
  date=intnx('month', namedt, 1)-1;
  do while(date<=nameenddt);
    output;
    date=intnx('month', date+1, 1)-1;
  end;
run;
\end{verbatim}

The first thing to do is to check that this dataset has, at each date, only

1. One ticker for each permno and
2. One permno for each ticker.
To check that each ticker has only one permno, I do the following:

```sas
proc sort data=stocknames;
  by date ticker permno;
run;

data doubles singles;
  set stocknames;
  by date ticker permno;
  if not (first.ticker and last.ticker) then output doubles;
  else output singles;
run;

proc print data=doubles(obs=20);
run;
```

The data step could do with some explaining. Since stocknames is sorted by DATE TICKER PERMNO, if each ticker linked to only one permno on each date, it should be the case that each date-ticker combination has only one entry in the dataset. If it does not, then that date-ticker combination links to more than one permno.

This produces output like this:

<table>
<thead>
<tr>
<th>Obs</th>
<th>PERMNO</th>
<th>ticker</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29938</td>
<td>BFD</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>2</td>
<td>29946</td>
<td>BFD</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>3</td>
<td>27414</td>
<td>B0</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>4</td>
<td>27561</td>
<td>B0</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>5</td>
<td>29866</td>
<td>BQ</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>6</td>
<td>29874</td>
<td>BQ</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>7</td>
<td>30891</td>
<td>CDL</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>8</td>
<td>30904</td>
<td>CDL</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>9</td>
<td>31093</td>
<td>CLI</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>10</td>
<td>31106</td>
<td>CLI</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>11</td>
<td>32854</td>
<td>HG</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>12</td>
<td>32862</td>
<td>HG</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>13</td>
<td>32934</td>
<td>HUB</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>14</td>
<td>32942</td>
<td>HUB</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>15</td>
<td>35553</td>
<td>PDL</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>16</td>
<td>35561</td>
<td>PDL</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>17</td>
<td>35115</td>
<td>PKR</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>18</td>
<td>35123</td>
<td>PKR</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>19</td>
<td>22517</td>
<td>PPL</td>
<td>31JUL1962</td>
</tr>
<tr>
<td>20</td>
<td>35051</td>
<td>PPL</td>
<td>31JUL1962</td>
</tr>
</tbody>
</table>
where, as you can see, the doubles dataset contains the desired elements. I then use the singles dataset, which has only one permno for each ticker, in further tests, notably, seeing whether each permno matches to only one ticker.

This process is easily automated in a macro:

```sas
%macro separatedups(inds, byvar, dupbyvar);
    proc sort data=&inds;
        by &byvar;
    run;

    data zzsingles zzdoubles;
        set &inds;
        by &byvar;
        if not(first.&dupbyvar and last.&dupbyvar) then output zzdoubles;
        else output zzsingles;
    run;

    %let doubsnobs=0;

    data _null_;
        set zzdoubles nobs=nobs;
        if _n_=1 then call symput("doubsnobs", trim(left(nobs)));
    run;

    %if "&doubsnobs"="0" %then %do;
    %end;
    %else %do;
        %put WARNING: &doubsnobs observations in doubles;
    %end;
    %mend separatedups;
```

And called with:

```sas
%separatedups(stocknames, date ticker permno, ticker);
```

I can then check both ways by saying:

```sas
%separatedups(stocknames, date ticker permno, ticker);

data stocknames;
    set zzsingles;
run;

%separatedups(stocknames, date permno ticker, permno);
```
data stocknames;
    set zzsingles;
run;

In the first call to %separatedups, there are multiple permnos for each ticker, so zzdoubles is not empty and the following message is written to the log:

WARNING: 35222 observations in doubles

In the second call, zzdoubles is empty: each permno matches to only one ticker.

10.3.4 Sundry macros

• Printing:
  When I debug, I like to print out small subsets of my datasets at each point. Typing

    proc print data=datasetname(obs=20);
      title "datasetname";
    run;
    title;

  just doesn’t cut it. The %stdprint macro allows me to avoid this

    %macro stdprint(inds, sopt);
      proc print data=&inds(obs=20);
        title "&inds : &sopt";
      run;
      title;
    %mend;

  I call this by saying

    %stdprint(datasetname);

  This prints the first 20 observations, together with the dataset name as title. It also allows me to have an optional extra subtitle (&sopt in the macro definition), so I can distinguish different printings of the same dataset.

  %comprint does the exact same thing, except that it prints the whole dataset. It is %stdprint without the “obs=20”.

• Cleaning up the WORK directory:
  If you’re dealing with large datasets and you have a space constraint, you might want to delete each dataset after you’re done with it, so that space in your WORK directory is freed.
  One way to do this is to say

41
proc datasets lib=work nolist;
   delete datasetname;
quit;

The nolist option stops proc datasets from printing a list of members of the work directory to the log file.
This can be replaced by:

%macro dsdel(inds);
   proc datasets lib=work nolist;
      delete &inds;
   quit;
%mend;

And the one-liner in open code

%dsdel(useless_dataset);

10.3.5 Regressions

Often, in preliminary analysis, I want to run simple regressions and have printed output that isn’t
the reams and reams of inessential statistics that SAS produces when I run proc reg. To do this, I
use the following macro:

%macro regme(inds, depvar, indvars, subt);

    proc reg data=&inds;
       model &depvar = &indvars;
       ods select ParameterEstimates;
       title ‘‘&subt’’;
       run;

%mend;

And now I can write, for instance,

%regme(crsp.msf, ret, retx, Regression of return on return without dividends);

which produces the beautifully simple output

    Regression of return on return without dividends

    Obs  Variable  Estimate  tValue  Probt
    1    Intercept  0.00192  467.15  <.0001
    2    RETX      0.99877  41951.9 <.0001

which is all I am really interested in anyway.
%regmewhite
11 The DOW loop

The DOW loop is a construct in which observations are read from an input dataset via a set statement, and the set statement is inside a do-loop.

For instance, suppose I wanted to find the mean return for each permno in the MSF file. I could do this in two ways.

The obvious way is:

```plaintext
proc means data=crsp.msf noprint;  
   by permno;  
   var ret;  
   output out=means mean=meanret;  
run;
```

The DOW-loop way is:

```plaintext
data means;  
   count=0;  
   sum=0;  
   do until (last.permno);  
      set crsp.msf;  
      where ret is not missing;  
      by permno;  
      sum=sum+ret;  
      count=count+1;  
   end;  
   meanret=sum/count;  
   output means;  
run;
```

What’s going on here? Briefly, the do-loop iterates over each permno’s returns, exiting when the last observation for that permno has been reached. When this happens, meanret is calculated and output.

This might seem like a fancy construct that has no obvious practical use, but its utility is enormous, particularly when you want to merge two datasets and then calculate summary statistics on the merged dataset, as will be explained in section 12.5.

For further reading, see [Dorfman and Shajenko (2007)]

12 Merges

12.1 Merging a dataset with a single observation into another dataset

A common task is merging a dataset which has summary statistics into the dataset that contained the original data from which those summary statistics were calculated.
For instance, I may want to calculate the demeaned value of a variable. To do this, I first use proc means to calculate the mean of that variable and then merge proc means’ output dataset into the parent dataset.

I describe a construct that does this efficiently.

Suppose the parent dataset is crsp.msf, and I wish to calculate the demeaned value of returns.

I first find the mean value using proc means:

```sas
proc means data=crsp.msf noprint;
var ret;
output out=summarystats(drop=_type_ _freq_) mean=meanret;
run;
```

SUMMARYSTATS has one variable named meanret, and one observation. To merge this back into msf, I say:

```sas
data msf;
  if _n_ =1 then set summarystats;
  set msf;
run;
```

This creates the additional variable meanret in msf and copies its one value to all observations in msf.

Why does this work? It’s a complicated answer. This relies on two things, the implicit RETAIN in every SET statement and the behaviour of SAS with respect to the endings of input datasets. These are fairly complicated and not essential, so I shall not go into them here. Interested readers may look at Sections 16.1 and 16.2.

12.2 Merging without sorting: the DATA step HASH object

I often have to merge a small dataset into a much larger dataset. If the larger dataset is not appropriately sorted, this is a process that consumes time and resources. An option is to use the DATA step HASH object.

The HASH object allows you to load a “small” dataset into RAM as a hash. A hash, or associative array, is a list indexed by a key. For instance, for the MSF dataset, the key variables are permno and date – which means that for a particular permno-date combination, the MSF dataset contains one observation, with the data being return, price, shares outstanding and so on. Given a permno-date combination, I could find the data for that combination by opening the MSF dataset and going through it until I hit the appropriate value. What a hash does is to allow you to instantly access the data for any specified permno-date combination without going through the entire dataset.

How do you create and use the hash? An example is probably the best way to explain.

In this example, I want to merge the monthly-frequency Fama-French factors with the MSF dataset, so I can run regressions to calculate betas.
```sas
data msf;
   set crsp.msf(keep=permno date ret);
   where ret is not missing;
   date=intnx('month', date, 1)-1;
run;

data ff;
   set ff.factors_monthly(keep=date mktrf smb hml umd rf);
   date=intnx('month', date, 1)-1;
run;

All standard so far. Observe that I make the dates in both datasets end-of-month.
Now I define and use the hash.

data msf;
   if _n_=1 then do;
      declare hash h(dataset:'ff');
      h.defineKey('date');
      h.defineData('mktrf', 'smb', 'hml', 'umd', 'rf');
      call missing(mktrf, smb, hml, umd, rf);
      h.defineDone();
   end;
   set msf;
   if h.find() = 0 then output;
run;

What this does is:

1. On the first observation (_N_=1), before any data is read from the SET MSF statement, a
   hash named H is defined. This hash consists of data from the dataset FF. The hash key,
   which will be used to perform lookups, is defined to be DATE. The hash data is defined to
   be MKTRF, SMB, HML, UMD, RF. The CALL MISSING statement sets the value of these
   variables in the output dataset to missing initially. If you omit this, SAS will complain that
   the variable are not defined.

2. Then the first observation is read from the SET MSF statement, loading values of DATE,
   PERMNO, and RET. Suppose these values are 10000, 19860228, and -0.257143. The H.FIND() statement immediately below the SET statement looks at the values of the KEY variables currently being processed. In this case, the KEY variable is DATE, which currently has the value 19860228. The H.FIND() statement searches the hash for a key value of 19860228. If it finds it, it populates the variables MKTRF, SMB, HML, UMD, RF with the data corresponding to that KEY in the hash, and returns a value of zero. If it does not find it, the variables MKTRF, SMB, HML, UMD, RF continue to have the missing value, and the H.FIND() returns a nonzero value.
```
3. If H.FIND() returns a zero value, indicating that it found that date in the dataset FF, the observation is output to the output dataset. Otherwise the observation is not output. If you wanted to keep both, observations for which a corresponding observation in FF exists, and those for which a corresponding observation does not exist, you could say:

```sas
data msf;
  if _n_=1 then do;
    declare hash h(dataset:'ff');
    h.defineKey('date');
    h.defineData('mktrf', 'smb', 'hml', 'umd', 'rf');
    call missing(mktrf, smb, hml, umd, rf);
    h.defineDone();
  end;
  set msf;
  if h.find() = 0 then foundinFF=1;
  else foundinFF=0;
run;
```

The foundinFF variable will tell you whether the observation existed in the FF dataset or not.

Some caveats:

- The hash is only useful when you are subsetting a larger dataset. If you wanted to keep observations in FF that did not exist in MSF, this form of the hash would not help.

- The dataset that is read into the hash has to be below a certain size, otherwise it will not fit into your memory. In my situation, MSF just about fits into memory. SAS will die with an out-of-memory error if you try to load a dataset that is larger than available memory.

The hash is of enormous utility in performing sortless merges, but it is also very useful in other situations. An example of this is many-to-many merges, explored in Section 12.4.2.

Because it is a headache to type out all this when you want to do merges with hashes, I wrote a macro that does it for me. See section 10.3.2 for the macro.

A nice introduction to the hash object can be found in Secosky and Bloom (2006).

### 12.3 One-to-many with the DOW loop

If you have to do a one-to-many merge on sorted datasets, you can do it with the DOW loop, which I will define as a DO-loop which contains a SET statement (see section 11). The utility of this is doubtful: since your datasets are already sorted, you might as well use the data step merge, but it's a nice illustration of what the DOW loop can do, and serves to sharpen your understanding of the DATA step.

Suppose I want to merge crsp.msf[keeping price and shares outstanding], with the CRSP mutual fund holdings dataset, which has crsp_portno, permno, nbr_shares, and eff_dlt. eff_dlt is the effective date of the holding.
data msf(rename=date=eff_dt);
    set crsp.msf(keep=permno date ret prc shrout);
    date=intnx('month', date, 1)-1;
run;

data holdings;
    set crsp.holdings(keep=crsp_portno eff_dt nbr_shares permno);
    where permno is not missing;
run;

proc sort data=msf;
    by permno eff_dt;
proc sort data=holdings;
    by permno eff_dt;

data complete;
    set msf;
    do until(last.eff_dt);
        set holdings;
        by permno eff_dt;
        output complete;
    end;
run;

This reads an observation from MSF. It then enters the DO-loop, and reads all observations from HOLDINGS until the LAST.EFF_DT=1, after which it goes to to the top of the DATA step and reads the next observation from MSF.

The obvious question is: how do you know that the permno-eff_dt combination read from HOLDINGS is the same as that read from MSF? In this setup, you don’t. To make it work, before the DATA step with the DOW loop, you need to do the following:

    proc sort data=holdings(keep=permno eff_dt) out=permnoeffdts_holdings nodupkey;
    by permno eff_dt;

    proc sort data=msf(keep=permno eff_dt) out=permnoeffdts_msf nodupkey;
    by permno eff_dt;

    data msf;
        merge msf(in=a) permnoeffdts_holdings(in=b);
        by permno eff_dt;
        if a and b;
        run;

47
data holdings;
  merge holdings(in=a) permnoeffdts_msf(in=b);
by permno eff_dt;
if a and b;
  run;

This restricts the two datasets to contain only those permno-eff_dt combinations which exist in both.

When this is done, you run the DOW-loop as above:

   proc sort data=msf;
      by permno eff_dt;
   
   proc sort data=holdings;
      by permno eff_dt;
   
   data complete;
      set msf;
   do until(last.eff_dt);
      set holdings;
         by permno eff_dt;
      output complete;
   end;
   run;

What happens now? Now you know that for every observation read from MSF, there is at least one observation in HOLDINGS with the same permno-eff_dt combination. Let me now go through the logic again.

The first observation is read from MSF. The permno-eff_dt combination of that observation exists in HOLDINGS, and because HOLDINGS contains only those combinations that are also in MSF, that combination must be the first combination encountered in HOLDINGS. We keep reading in observations from holdings (and outputting to COMPLETE) until LAST.EFF_DT=1, at which point there are no more observations in HOLDINGS with that particular combination of PERMNO and EFF_DT. Then we exit the DO-loop and read the next observation from MSF, and repeat the process.

12.4 Many to many

12.4.1 Using proc sql

Not recommended.
12.4.2 Using hashes with multidata

12.5 Post-merge processing: merge and then calculate summary statistics

eg: merge with proc sql and then sum The DOW loop

13 Statistics

13.1 Weighted means with negative weights

I often want to calculate the return to portfolios of stocks where the weights of some stocks are negative. For instance, suppose we have a dataset PORTWEIGHTS that contains

<table>
<thead>
<tr>
<th>permno</th>
<th>portweight</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>0.2</td>
</tr>
<tr>
<td>10006</td>
<td>0.3</td>
</tr>
<tr>
<td>10010</td>
<td>-0.4</td>
</tr>
</tbody>
</table>

We want to calculate the returns to this portfolio over January 2001.

So we say:

```plaintext
data msf;
  set crsp.msf(keep=permno date ret);
  where intnx('month', date, 1)-1 = '31Jan2001'd); *Moves date to end-of-month;
run;
```

and then merge this with PORTWEIGHTS by PERMNO to get a dataset PORTWEIGHTSRETS, and then just say

```plaintext
proc means data=portweightsrets;
var ret/weight=portweight;
run;
```

The problem is that SAS sets negative weights to zero when it calculates means. So permno 10010, for instance, will not be included in the portfolio.

The straightforward way to handle this is to say:

```plaintext
data portweights;
  set portweights;
  modifiedret=ret;
  if portweight<0 then modifiedret=-ret;
  absportweight=abs(portweight);
run;
```

```plaintext
proc means data=portweightsrets;
  var modifiedret/weight=absportweight;
run;
```
13.2 Robust standard errors
Clustered standard errors White standard errors Newey-West standard errors

13.3 Fixed effects

13.3.1 Brute-force
The brute-force way -which I do not recommend - to run fixed-effects models is to actually create
the dummies and run proc reg including the dummies as independent variables.

Suppose I have monthly data on stock returns and I want to naively test the hypothesis that
smaller stocks have higher returns. From crsp.msf I can construct a dataset with return and lagged
size, following the prescription in Section 7.2 to construct lagged sizes:

```
data msf(keep=permno date ret);
msflag(keep=permno date size rename=(size=lagsize));
set crsp.msf(keep=permno date ret prc shrout);
date=intnx('month', date, 1)-1;
output msf;
size=abs(prc)*shrout;
date=intnx('month', date+1, 1)-1;
output msflag;
run;
```

```
data msf;
merge msf msflag;
by permno date;
run;
```

MSF now has

PERMNO DATE RET LAGSIZE

Without dummies, proc reg is:

```
proc reg data=msf;
   model ret=lagsize;
run;
```

Suppose I want permno fixed effects. Let me describe the steps I need to do this brute-force.
First, I find out how many permnos there are. Then I create that many dummies, which are
variables such that, for each observation, only one can be 1, the others are all zero. Then I run
proc reg including these variables on the right-hand side.

So first, I must first find out how many permnos there are. To find this, I say:

```
proc sort data=msf out=count_of_permnos(keep=permno) nodupkey;
   where ret is not missing and lagsize is not missing;
```

COUNT_OF_PERMNOS contains a list of permnos which have data for the regression. To find out how many there are, we can do:

```sas
data count_of_permnos;
  if _n_=1 then put nobs;
  set count_of_permnos nobs=nobs;
run;
```

This might strike you as peculiar: why is the IF statement before the SET statement? Will the NOBS value have been populated when the IF statement is read? The answer is that NOBS is populated as soon as the data step code has been compiled, before any observations have been read. At the same time, the datasets named in SET, MERGE, UPDATE, and DATA statements are created with the variables they are expected to contain.

Now I open the log and see what value nobs took. The log contains

```sas
27865
NOTE: There were 27865 observations read from the data set WORK.COUNT_OF_PERMNOS. 
NOTE: The data set WORK.COUNT_OF_PERMNOS has 27865 observations and 1 variables. 
```

which tells me what I want to know.

I then proceed to create my dummies. I will do this using arrays:

```sas
data count_of_permnos;
  set count_of_permnos;
  array permnonames (27865) permnonames1-permnonames27865;
  do i = 1 to dim(permnonames);
    permnonames(i)=0;
  end;
  permnonames(_n_)=1;
run;
```

For each observation, this creates 27865 variables named permnonames1 through permnonames27865. It puts them into an array named permnonames, so that we can do something to each of them programmatically. The do-loop sets all members of the array to zero, and then the permnonames dummy corresponding to the current permno (i.e., the _N_-th permno: see Subsection 3.2.2 for a description of _N_). I do not recommend printing this dataset out.

I then merge these dummies back into the original dataset and run my regression:
data msf;  
   merge count_of_permnos msf;  
   by permno;  
run;  
proc reg data=msf;  
   model ret=lagsize permnonames1-permnonames27865/noint;  
run;  

The noint excludes the intercept from the model. Alternatively, you could say:

proc reg data=msf;  
   model ret=lagsize permnonames2-permnonames27865;  
run;  

If you'd rather not manually look at your log file to see how many dummies there need to be, you can use CALL SYMPUT in the following way:

data count_of_permnos;  
   if _n_=1 then call symput('nobs', nobs);  
   set count_of_permnos nobs=nobs;  
run;  
data count_of_permnos;  
   set count_of_permnos;  
   array permnonames (&nobs) permnonames1-permnonames&nobs;  
   do i = 1 to dim(permnonames);  
      permnonames(i)=0;  
   end;  
   permnonames(_n_)=1;  
run;  

All I have done is to replace the number 27865 with the macro variable &NOBS, which I created in a previous data step. You cannot use the non-macro variable nobs in defining the array because array definition requires an integer constant as dimension, not a variable.

Running this poses several problems. First: you have 27866 independent variables! The matrix to invert is enormous. This is clearly massively inefficient. Second: when you run proc reg like this, it’s going to print out the values of those 27865 dummy coefficients, which you don’t care about at all. See Section [10.3.5](#) for an example of how to avoid this.

### 13.3.2 An efficient way with one-way fixed effects

The efficient way is to use the Frisch-Waugh theorem which says that when you run a regression of Y on two sets of independent variables X_1 and X_2, the coefficients on X_2 are the same as if you did the following:

1. Run each variable in X_2 on X_1, get the residuals
2. Run Y on the residuals.
In our application, $X_1$ are the dummies, $X_2$ is lagsize.

The residual from regressing a variable on dummies is the variable demeaned by group. So regressing $X_2$, i.e. lagsize, on dummies is the same as demeaning lagsize by group - in this case, by permno.

To do this you can actually calculate the means using proc means by permno, merge the means in by permno, and then subtract the means, or you can do it in one step by using proc standard:

```
proc standard data= msf mean=0;
   by permno;
   var lagsize;
run;
proc reg data=msf;
   model ret=lagsize;
run;
```

You can check that the coefficient is the same as with the dummies.

An alternative is to use proc glm with the absorb statement, but I do not completely understand proc glm, and I am reluctant to use it. This is simple, transparent, and as easy to code. Should you have multiple independent variables (other than the dummies), this extends easily: just put the additional variables on the VAR statement in the PROC STANDARD and, as usual, on the MODEL statement in PROC REG.

13.3.3 Two-way fixed effects

Two-way fixed effects are easily coded up using the inefficient dummy method. Since we’re trying for efficiency, a better solution is to use a combination of the Frisch-Waugh Theorem and the inefficient method.

Explicitly, first determine which one of your dummy sets has fewer elements.

Suppose we want to run the regression in the previous subsection — returns on lagged size — with dummies for date and permno. Since data is monthly and the msf file has data from 1927-2007, there will be $12 \times 81 = 810 + 162 = 972$ date dummies. That’s much smaller than the 27865 permno dummies.

Then we create the date dummies as shown in Section 13.3.1 Explicitly:

```
data dates;
   date='31Jan1927'd;
   do while(date<='31Dec2007'd);
      output;
      date=intnx('month', date+1, 1)-1;
   end;
run;
data dates;
   set dates;
   array dateids (972);
   do i = 1 to dim(dateids);
```
and merge these in with the original dataset by date. There are 972 date dummies named dateid1-dateid972.

Once I have the date dummies, I treat the date dummies as additional independent variables:

```plaintext
proc standard data= msf mean=0;
    by permno;
    var lagsize dateid1-dateid972;
run;

proc reg data=msf;
    model ret=lagsize dateid1-dateid972/noint;
run;
```

which produces the two-way fixed effects coefficient on lagsize.

To suppress the printing of the dummy coefficients, I can say:

```plaintext
proc reg data=msf outest=est(keep=lagsize _type_) noprint tableout;
    model ret=lagsize dateid1-dateid972/noint;
run;

proc print data=est;
run;
```

13.4 Pairwise statistics with large datasets
Correlations

14 Efficiency

14.1 Reading into the PDV: where, keep, and in

When using a WHERE statement, don’t use formulae as far as possible. That is, say

```plaintext
where date>='01Jan1996'd;
```

rather than

```plaintext
where year(date)>=1996;
```

WHERE statements can be used with OUTPUT statements. (WHERE=)
14.2 Never use proc sql

14.3 Don’t sort more than strictly necessary

Sorting takes by far the most time of most programs.

Use the fact that the datasets you use are already sorted. DSF and MSF are sorted by permno date. A splendid example of this is the code on the WRDS website which merges TAQ.CT with TAQ.CQ.

Instead of sorts to merge, use hashes - Section [12], which do not require sorting and are faster anyway.

Instead of using proc means with a by statement, that is, instead of saying

```sas
proc sort data=crsp.msf out=msf;
   by date;
run;

proc means data=msf noprint;
   by date;
   var ret;
   output out=summarystats(drop=_type_ _freq_) mean=meanret;
run;

say
proc means data=crsp.msf noprint nway;
   class date;
   var ret;
   output out=summarystats(drop=_type_ _freq_) mean=meanret;
run;
```

I replaced the “by” with a “class”. I also used the NWAY option, which restricts the output to the “outremost level”. Without it, SAS produces output for all combinations of the CLASS variables. MEANING WHAT?

14.4 Macro loops make things possible, they don’t make them fast

Instead, use data steps. One thing that macros seem good for is running rolling regressions, or computing rolling means or standard deviations. A modification of the method used in section [7.3] should be used instead.

15 SAS on Unix

15.1 Transferring files from DOS: line endings

Files created on Windows have different line-endings from the ones used on Unix. Transferring files between systems (even program files) leads to strange results. After you transfer a file from DOS to Unix, be sure to say
dos2unix dosfilename unixfilename

The DOS line ending will often appear as \M in files opened in Emacs. If you see that you will know what to do.

15.2 Locating your work directory

To tell where your work directory is, run the following code in any SAS file:

```sas
proc options option=work;
run;
```

This prints your work directory to the log file. On Unix on WRDS, the work directory will have a name that looks like

```
/sastemp7/SAS_workCB0F00002C49_wrds.wharton.upenn.edu.
```

15.3 Relocating your work directory

On WRDS, your work folder is created in one of the sastemp directories. To choose which sastemp directory it is assigned to, at the prompt, you can say:

```bash
sas -work /sastemp11 myprogram.sas&
```

instead of

```bash
sas myprogram.sas&
```

This will place your work folder in /sastemp11.

You can check which sastemp directory is most full at the moment by saying

```bash
du -sh /sastemp*
```

at the prompt. This is good to know because when you are about to run a program which you know will create an enormous file which might put you over the 200G sastemp folder limit, it is best to start your program in the most lightly-loaded sastemp folder.

There is no reason that changing the location of your work directory should affect how fast your programs run on WRDS, but I’ve found that if my programs seem to get “stuck”, that is, if

```bash
ps -fu 'whoami'
```

constantly shows that my programs are not using up any CPU time, I find that killing them and moving the work directory to a different, less lightly-loaded sastemp folder helps.
15.4 Relocating your log and lst files

Log and lst files are created, by default, in the folder in which you are when you type the sas command. This may be different from the folder in which your SAS program is.

If your lst or log files are likely to be large and you are at or close to your space quota, running SAS will put you over the limit, causing SAS to die with a can’t-write-to-file error. To get around this you can use proc printto, which places the log and lst files where you say:

```
proc printto new log='/sastemp7/test.log' print='/sastemp7/test.lst';
run;
```

<further commands>

The log and the lst output from 'further commands' will be placed in the files named.
You could do the same with command line options:

```
sas -log /sastemp7/logfile -print /sastemp7/lstfile program.sas
```

The difference is that with command-line options, all output is directed as you stated; with proc printto, only commands after the proc printto statement direct output as stated.

15.5 Memory issues

15.6 Why does proc import hang on Unix?

See subsection 5.3

15.7 Searching for files on Unix

To search for all files in a directory that match a particular expression, use Unix’s `find` command. For instance

```
find . -name '*sas'
```

finds all files in the current directory (.) whose names end in `sas`.
This is even more useful than it seems at first glance. You can search for text within the files:

```
find . -name '*sas' | xargs grep 'sastemp'
```

searches all the SAS files found for the string 'sastemp'. The `xargs` command takes the output of `find` and gives it to `grep` one line at a time.
15.8 Utilities

Working with SAS on Unix is much easier if you have several small scripts to help you out. I find that the idea for these utilities is what’s important: the actual coding is trivial. You can use your own favourite language or shell to code these. I used perl.

I describe three such utilities.

15.8.1 Parsing the SAS log: logsasparse

After a SAS program finishes running, you should read the SAS log to see that there were no errors or unanticipated events. The log may be hundreds or thousands of lines. An easier way is to write a perl script that searches the log for keywords. My perl script is called logsasparse and prints out suspicious lines in the log.

I save this perl program as a file named logsasparse in a directory in my path, and I call it by saying

```
logsasparse mysasfile.log
```

The code itself is below:

```
#!/usr/bin/perl
foreach my $ARG( @ARGV)
{
    open(FH, "<$ARG") || die "$ARG inaccessible: $!";
    while(<FH>)
    {
        print $ARG," : ", $., " ", $_ if /(not found|multiple lengths|has more than one data set with|overwrit|uninit|warning|no observations|\[^\_\]\(?!set \)\)error(?! detection))/i;
    }
    close FH;
}

Particularly insidious errors this catches are when a variable is uninitialized, when a variable is overwritten in a merge, and when a data step merge is many-to-many without your realizing it. These errors are insidious because they are not reported as errors in the log, merely notes or warnings.

As a matter of course, you should have the option MSGLEVEL set to I. See Section 9.3 for details.

One nonobvious thing about this way of doing things is that if you create datasets named ERROR, which, if your program is working correctly, will have zero observations, logsasparse will print out how many observations they contain to the log file.

15.8.2 Looking at SAS datasets: sasprint

An apparent advantage of Windows for SAS is that SAS datasets (*.sas7bdat files) cannot be opened and read on Unix. This bash utility I call sasprint allows you to print the first N observations of a SAS dataset.

I save this bash script as a file named sasprint in a directory in my path, and I call it by saying
The code follows:

```bash
#!/bin/bash

if [ ! -f $1 ]; then
    echo "File $1 does not exist"
    exit
fi

fullfilename="$1"

filepath="${fullfilename%/*}"
filename=$(basename $fullfilename) #only the filename
filename=${filename%.*} #without the extension

numberoflines=$2
if [ -z $2 ]; then
    numberoflines=20
fi

wherestatement=""
if [ ! -z "$3" ]; then
    wherestatement="where $3;"
fi

varstatement=""
if [ ! -z "$4" ]; then
    varstatement="var $4;"
fi

command="libname xtemp '$filepath/'; proc print data=xtemp.$filename (obs=$numberoflines); $wherestatement $varstatement"

blank=" " #Hack: sas seems to chop off the first few characters

echo "$blank $command" | sas -sysin /dev/stdin -print /dev/stdout -log /dev/null

#Check if SAS returned an error
```

59
if [ $? -ne 0 ]; then
  echo "Error in program: rerunning for log"
  echo "$blank $command" | sas -sysin /dev/stdin -log /dev/stdout -print /dev/null
fi

Notice the use of the sysin option, which tells SAS where to find code (in this case, in standard input: STDIN). Notice also the redirection of the log and lst files (see section 15.4).

15.8.3 Killing all sas processes: saskill
I often want to stop a SAS process midway. The proper way to do this is to list my processes:

    ps -fu 'whoami'

to find the process ID (PID) of the SAS process and kill it by typing

    kill -9 <PID>

On some Unix systems this can be done in one step my using the killall command, which takes as its argument the commandname which you want to kill all instances of:

    killall sas

This command is not available on all systems. However, there’s nothing easier than automating the two-step process above:

```
#!/usr/bin/perl -w

my @userpids = 'ps -u $ENV{'USER'} -o pid='; #Get a list of PIDs I (that is, $ENV{'USER'}) am running
my @saspids='ps -C sas -o pid='; #Get a list of PIDs corresponding to SAS processes

chomp(@userpids); #Remove trailing newlines
chomp(@saspids); #Remove trailing newlines

my %original = ();
my @isect = ();
map { $original{$_} = 1 } @userpids;
@isect = grep { $original{$_} } @saspids; #Find the intersection of the two lists

foreach my $pid (@isect) #Kill each process whose PID is in the intersection
{
  print "Killing pid: $pid\n";
  my $a='kill -9 $pid';
```

60
I save this as a file named saskill in a folder in my path, make it executable, and call it by saying

saskill

at the prompt.

15.9 The autoexec.sas file

The autoexec.sas file is a file in your home directory (Unix) or in C: (Windows). It contains SAS statements that are executed each time a SAS session is started. It is a useful place to keep library definitions, options statements, and %include statements for macro files that you use constantly.

A section of my autoexec file follows:

```
%sysexec(perl ~/utils/cleanworksas);
%include '~/utils/sasmacros/macrolist.sas';
options linesize=130;
options dtreset;
options mprint symbolgen msglevel=i;
proc options option=work;
run;
%put The PID of this SAS process is: &sysjobid;
```

The %sysexec statement launches a shell to execute the command given. Here it runs perl on my cleanworksas program, which cleans out old work directories from previous SAS sessions. You’re unlikely to need this on WRDS, because this is done automatically there.

macrolist.sas contains all the macros I referred to in Section 10.3.

linesize sets the width of the page (80 is good for printing on A4 paper). options dtreset prints the current date and time anytime the date or time has to be written to the log or lst files, instead of the date and time the program started, which is the default.

The proc options statement prints the name of the work folder to the log. This is useful in case I want to look at the datasets in WORK while the program is running, or delete them manually if the program exits abnormally.

&sysjobid is a macro variable, automatically created by SAS, containing the Unix process ID of the current SAS job. This is a useful thing to have when you have three jobs with the same name running and you want to kill one, but aren’t sure which is which.
15.10 The .sasv9.cfg file

The .sasv9.cfg file is a place to put options that would otherwise go on the command line when you invoke SAS. For instance, if I wanted to change my work directory to /sastemp7, I would ordinarily say

```
sas -work /sastemp7 mysasfile.sas &
```

when I run mysasfile.sas. If I want my work directory to always be in /sastemp7, I would open the file .sasv9.cfg and put this single line in it:

```
-work /sastemp7
```

Now every time SAS is invoked, it behaves as if it were invoked with

```
sas -work /sastemp7
```

15.11 SAS with X-Windows

Horrendously slow, even with a wired connection. Windows pop up all over the place. Not recommended.

16 Further interesting topics

16.1 What really happens in the merge statement

16.2 The fascinating Do-loop

References


