International Economics: Homework

Alan G. Isaac

Uncollected Assignments

I strongly recommend that you do the end-of-chapter problems in my posted notes. However, these are not collected.

Preliminaries

The data-based assignments require the use of econometric software or a spreadsheet.

Important Preliminary for Windows Users Before you begin, make sure you can see your file extensions when you look at your files. The default in Windows is to hide standard extensions in Windows explorer. In this course, the main problem is that sometimes you will accidentally save a file with an extra `.txt` extension, and then you will not understand why your software is not treating it as you expect. (A more general problem is that hidden extensions allow you to be tricked into thinking a dangerous file, such as a Visual Basic script named `filename.jpg.vbs`, is really a safe file, such as `filename.jpg`.) Open Windows Explorer, pick `Tools/Folder Options`, click `View`, and then *uncheck* `Hide file extensions`.

Downloading Files

I may sometimes provide you with data for your assignments. (EViews workfiles have extension `.wf1`, gretl data sets have extension `.gdt`, spreadsheet files have extension `.csv` (althought Excel and Gnumeric data sets may have extension `.xls`). To download: Do not left click on the link; right click on the link! Save it with the name you are provided in the assignment.

> Anytime a command or option is new to you, be sure to read about it in the Documentation.

Typing

I have seen students try to program with "hunt and peck" typing. Do yourself a favor and acquire some basic keyboarding skills. These can be quickly acquired. There are surely many sources of cheap software assistance. AU's Academic Support Center offers free access to the inexpensive commercial MavisBeacon application. There are also free and open source typing tutors.

General Considerations

- Create a course folder `econ672` for this course. (This can be on a *personal* computer drive, or a flash drive, or a network drive. But *not* on a lab computer drive!) I will assume that you keep your data and scripts in subfolders: data in `econ672/data` and scripts in `econ672`.

- It is an important work habit to *never* alter your raw data. Keep the data files I give you inviolate. Instead of altering any file I provide, or any raw data you collect, work with a copy. Give the copy a name that clearly distinguishes it from your raw data.

  One way to protect your data is to set the read-only attribute on the file containing your raw data. If you are working with a spreadsheet (such as Excel, Calc, or Gnumeric), you can alternatively "protect" the sheet containing the data. E.g., pick `Tools/Protection/ProtectSheet` from the menus.

- Another important habit is to document your work by adding informative comments to your programs. (See Program Comments.) Provide lots of helpful comments for the reader of your program. Additionally, be sure to document the author(s) and date of the program: usually this information will come at the very top of your program file.

  You should include a comment for most of the commands in your program. Certainly do this for *every* first use of a command in a program: above the first occurrence of the command add a comment line that explains *exactly* what you are doing, in your own words. An *exact* explanation will of course include a discussion of any of the options you invoke when using the command. Your comments should explain the function the command. (Be sure to first read about the command in the Documentation.) Also explain what you are trying to do with the command (i.e., what you are accomplishing in your program).

  The key issue is completeness: making your comments complete is a very important practice. Make sure your program file has lots of helpful comments in it. (If you should stumble across it years from now, you should be able to rely on your comments to learn what the program does.)

- Unless otherwise instructed, you must turn in a program file for each assignment that includes data work. Your program must be completely debugged, so that I can simply run it to produce all of your results. (Programs that do not run will not be graded.) Test your program one last time to make sure it runs, then email it as an attachment to your TA. (See the syllabus for details.)

- For each assignment you must also create a report.

  1. Your report should include a full discussion of your work, with explicit reference to any graphs your create. (Import your graphs into your document.)
  2. If you run any regressions, your report should include full regression results, along with an extensive discussion. (See Reporting Estimation Results.)
  3. Tables should look professional. They should look acceptable for a job seminar or professional publication. This means that they should be easy for others to read and interpret. Note that the natural place to put any notes or comments that belong with your table is *under* the table (not up by the title).
  4. Graphs should look professional. They should look acceptable job seminar or professional presentation. This means strictly limiting the commentary you include on the figure. (See Graph Annotation.) Each graph should have an informative title and axis labels. Each data-based graph should state a data source.
  5. Create your report as a PDF. You will probably need to save your graphs as PNG files and use file upload features. (See Graph Formats.) Once you are done, enable the TA as an editor of your document. (See the syllabus for details.)

- Be sure to save your work! (I.e., retain a copy of your program file, and save a copy of your report to disk as a PDF!)

Assignments

There are no collected homework assignments.

Vim users: here is a hint for accumulating all your show commands in the right order.

Once you have written your program, including all your show commands each time you produce a new graph or table, issue the editor command

```
:g/^show/t 0
```

Then move the copied show commands (which you will find at the top of your file, in reverse order as desired) to the bottom of the file.

1 First Steps

Objective: The primary purpose of this assignment is to introduce your data analysis software. It is also a first look at the relationship between the spot rate and the relative price level. Finally, you will learn how to retrieve data from the International Financial Statistics.

Estimated Time for Completion: 2.5 hours.

1. Make sure you *carefully* read the Preliminaries above. Make sure you *carefully* read the General Considerations above. Get the following monthly data from 1957 to the present from International Financial Statistics: spot rate (USD-CAD), CA price level (CPI), US price level (CPI). (See IFS Online.) Save the data to your `econ672/data` folder. (Do not include whitespace in the filename.) This is your raw data: you do not want to change this file. Start up your software and load your data set so that you can work with it. (See Data Formats.)

2. Learn how to create a new program file or script. (See Program Files.) Learn how to add comment lines to your program file. (See Program Comments.) At the top of your program file, add two comment lines with the following content.

   ```
   Author(s): Your Name(s)
   Last modified: Today's Date
   ```

3. Now document your data. (See Data Documentation.) Specify the sample (length and frequency), as well as what each series represents. Determine the units for the exchange rate. This information will be important in what follows.

4. For the USD-CAD spot rate series, produce a line graph using data from 1957–2000. (See Creating Graphs.) We will name this graph `g1`. Appropriately annotate `g1`. (See Graph Annotation.) Make sure you load the data using a relative location: it should be in the `data` folder just below your script.

5. Produce a new series: the relative price level (CA/US). (See New Series from Old.) Produce a line graph of the relative price level for the same period. We will name this graph `g2`. Appropriately annotate `g2`.

6. Next, combine the information in your previous two plots by producing line plots of the two series in a single chart, which we will call `g3`. Once again annotate your graph appropriately. (You could produce more readily comparable series by normalizing a common date to unity, but this is not required for this assignment.)

7. Now you are ready to start your report on this exercise. Import the graphs you have created into your document and discuss `g3`. (You should import `g1` and `g2` into your report, but you are not required to discuss them separately.)

   Question: How closely does the spot rate track the relative price level?

   Roughly speaking, the relative price level is proportional to the *purchasing power parity* (PPP) exchange rate. Examine your graph visually to determine how closely the spot exchange rate tracks the PPP exchange rate.

   Question: What conclusion do you draw about the stability of the real exchange rate?

8. Recreate the last graph (`g3`) with your full dataset. Call this new graph `g4`. Import `g4` into your report and discuss it.

Question: Does extending the sample change your understanding of what is happening?

2 German Hyperinflation

Objective: The object of this exercise is to explore Frenkel's work on the post-WWI German hyperinflation by attempting a replication and extension of the results reported in Frenkel (1976).

Estimated Time for Completion: 5 hours.

Background: You will need to read the Frenkel (1976) article *carefully* in order to complete this assignment. The data is not identical to Frenkel's, so you will not match his results exactly.

For this exercise, create a report providing a written answer to each question. Also, create a *commented* program file as you analyze the post-WWI German hyperinflation. (See Program Files.)

1. Make sure you carefully review the General Considerations above. Save the data to your `econ672` folder: (Keep the file name I used.) Create a working copy of the data set. Start up your software and load your data set.

   EViews

   gretl

   CSV and header

2. Start a new script. (See Program Files.) Add comments documenting the author(s) and creation date. (See Program Comments.) Add commands to create a working copy of the data set I provided. (See Data Formats and Working Copy.)

3. Get familiar with the series I provided. (Do not skip this necessary step!) (See Data Documentation.) Generate four new series: s (the log of the spot rate), f (the log of the forward rate), m (the log of the money supply), and fd (the foward discount; calculate this as $f - s$). (See New Series from Old.) For example, in order to produce the log of the forward rate, your program will include a computation along the lines of the following.

   ```
   f = log(endf_s_e + ends_e)
   ```

   Question: Is this the proper calculation? (Explain.)

4. We produce fd differently than Frenkel (1976) did. Explain how they are different.

   Question: Using what you know about logarithms: is our fd is a good approximation to Frenkel's $\log \pi$? (Be sure to distinguish his π and π^*.)

5. Next we want to replicate Frenkel's figure 3. Produce a line graph containing the spot rate and the lagged forward rate, which we will call `fig3`. (See Creating Graphs and Lagged Values.)

   Question: Is your graph *essentially* identical to Frenkel's graph? (Pay attention to the labeling of the axes!)

6. Next we annotate your graph. Start by adding your name and date. Also add the data source. (See Graph Annotation.)

Question: In your report, give a brief description of what you learn from this graph.

7. Now create a graph by following the same steps as the previous graph with one exception: use the contemporaneous forward rate instead of the lagged forward rate. I will refer to this graph `fig3c`.

   Question: How does your `fig3c` compare to your previous graph and to your expectations?

8. Next, produce a scatterplot that combines the information in Frenkel's figures 1 and 3. On the vertical axis, put the log of the spot rate. On the horizontal axis, put the lof of the money supply and the log of the lagged forward rate. (See Scatterplots.) We will call this `fig13`.

9. Now it is time to reproduce Frenkel's key equation, equation 4", as closely as we can. Use "least squares" reqression to determine how well a constant c, the money supply m, and the forward discount fd "explain" the behavior of the spot rate s. (See Least Squares Regression.)

   Call the table representing your regression results `f4c_t`. Add the date and a very brief commentary to your table. Include this table in your report.

   Question: Compare your coefficient estimates to those of Frenkel. Do you reach the same *qualitative* conclusions with your data set?

10. Finally, see how closely you can replicate Frenkel's equations 6' and 6" using linear least squares regression. (See Least Squares Regression.) Produce two more commented tables. Include these results in your report.

    Question: What are these regressions supposed to show us?

3 Introduction to Unit Roots

Objectives

- Visually and conceptually distinguish white-noise and random walk series.
- Illustrate the behavior of such series.
- Test series for the presence of a unit root.
- Generate a white noise series and test for a unit root.
- Generate a random walk series and test for a unit root.

Estimated Time for Completion: 3 hours.

Most Common Errors

- Failure to add program comments *fully* describing each new command or option.
- Failure to fully explain and correctly use an iterative procedure to produce an AR(1) series.
- Seeding the RNG more than once.
- Failure to relate the behavior of the regression residuals to the time series properties of the series.

Note: Be sure the read a good discussion of the Dickey-Fuller test before attempting this assignment. Possible sources include Wikipedia, any modern econometrics text, or the EViews User Manual. Be sure to read Stationary and Nonstationary Series.

Your examination of the behavior of white noise processes should confirm that they are "stationary": innovations are transitory. In contrast, innovations to a random walk are permanent: it is non-stationary.

Assignment

1. Seed your random number generator with the seed 314159. (Setting a seed ensures your results will be the same each time you run your program. Having everyone set the same seed ensures each will see the same outcomes.) For this assignment, you will seed your random number generator only once. (See)

2. Generate a variable `wn1` containing 300 independent draws from a standard-normal distribution. (See Random Number Generators.) Produce a properly annotated line graph of this series named `wn1Line`. (See Creating Graphs and Graph Annotation.) (A line graph is probably easiest to interpret visually.) If you label the horizontal axis, use t (for "time").

   Question: What do you learn from your graph? (I.e., what visual clues do you detect indicating whether this is a stationary process?)

3. Examine the mean, variance, skewness, and kurtosis of `wn1`. (See) Report these values and describe the meaning of each. (If you rely on a textbook or web resource, please cite it.) Do they suggest that `wn1` is drawn from a normal distribution? Now test for normality using the Jarque-Bera statistic. Report this value and describe its meaning. (Not all statistics packages will compute this for you, but you can easily implement the formula at `https://en.wikipedia.org/wiki/Jarque%E2%80%93Bera_test` if needed.)

4. Conduct an ADF test on `wn1`. (See Unit Root Tests.)

   Question: What is the value of the ADF test statistic? Compare the ADF test statistic to the critical values. Can you reject the null hypothesis for this series at the 10% level? Can you reject the null hypothesis for this series at the 1% level? (Be sure to interpret the unit root test *after* carefully reading a good discussion of unit roots, including a discussion of Dickey-Fuller tests.)

   > ADF tests are often sensitive to lag length. There are many approaches to lag-length selection. (See Lag Length.) For this exercise, you can either use automatic lag-length selection (if provided by your software) or arbitrarily set the lag length to 4. Also, this time you need not include a constant or trend in your test.

5. Following the same procedure as for `wn1`, create a second white noise process named `wn2`. Now your have two series, `wn1` and `wn2`, that represent two independent white noise processes.

   Question: What would happen if you were to reset the seed for your random number generator to its original value before creating `wn2`? Would that be desirable for this exercise?

6. Create a scatter plot of `wn2` against `wn1`. (See Scatterplots.) Call this graph `wn12scat`.

Question: Should we expect `wn2` to look unrelated to `wn1`? Does it?

7. Regress `wn2` on `wn1`. Name the residuals from this regression as `wnResids`. (We will need these later.) Produce a table of regression results.

   Question: Discuss your regression results. Pay special attention to the p-value for the coefficient on `wn2`: it must be very small for us to reject the null hypothesis that the series are unrelated. Add a regression line to your scatter plot `wn12Scat`: does it look as expected?

8. Generate a random walk named `rw1` by creating the *cumulative sum* of the series of white noise shocks constituting `wn1`. (See Cumulative Sum.) Produce a properly annotated line graph of this series named `rw1Line`.

   Question: What do you learn from your graph? (I.e., what visual clues do you detect indicating whether this is a stationary process?)

   > Make sure you produce two different objects named `wn1` and `rw1`, rather than two different names for the same object.

9. Look at the same descriptive statistics for `rw1` as you did for `wn1`. Do they suggest that this random walk process is drawn from a normal distribution? Now test for normality using the Jarque-Bera statistic. Report this value and describe its meaning.

10. Conduct an ADF test on `rw1`. Compare with your results for `wn1`. Summarize your observations in your report.

11. If we *independently* generate a second random walk process, we might naturally expect it to be completely unrelated to the first. To see if this expectation is met, construct another random walk series `rw2` from `wn2`, following the same procedure you used to construct `rw1` from `wn1`. Examine the two random-walk series just like you examined the white-noise series. (I.e., look at the scatter plot and the regression results, and include *detailed* discussion in your report.) Name the regression residuals `rwResids`.

12. Granger and Newbold (1976) showed that unrelated random walks appear related about 75% of the time. Phillips (1986) showed that the larger your sample, the more likely you are to reject unrelatedness! However, take a look at the two series of regression residuals you stored, and see if you can discover any clues to spurious regression in these. Produce an annotated line graph of your two series of regression residuals (which you should have named `wnResids` and `rwResids` above). In your report, summarize your observations on the differences between the two series of residuals. (Hint: they should be *very* different!)

13. Finally we look at highly autoregressive series. Using copies of your `wn1` and `wn2` series, produce two new series (`ar1` and `ar2`), each described by the relationship

    ```
    x(t) = 0.98 x(t-1) + u(t)
    ```

    Let $u(t)$ be your white noise series `wn1`.

    (You should construct these iteratively. This is almost like constructing random walk series, but with the weight 0.98 instead of 1.0 on the lag.) Repeat the rest of our exploration once again, with your two new autoregressive series. That is, graph your first autoregressive series and conduct an ADF test on it. Create a scatter plot of ar2 against ar1, and produce the regression results. Include the scatterplot in your report. Comment on the scatterplot and the regression results. Examine the regression residuals visually. Make summary observations in your report.

14. Turn in your fully commented program file by email. (See the syllabus for details.)

As always, you may talk about the assignment with your colleagues, but you must *independently* write your own program and comments.

> For this assignment you may be able to find some online guidance, depending on your chosen language. For Python, see `https://subversion.american.edu/aisaac/hw/macro_hw.htm#py_uroot`. For EViews, see `https://subversion.american.edu/aisaac/hw/macro_hw.htm#ev_uroot`.

4 Long-Run Purchasing Power Parity

Objectives

- Gain experience retrieving data from the web for empirical work.
- Learn whether purchasing power parity appears valid as a very long-run phenomenon.
- Apply unit root tests to real data.

Estimated Time for Completion: 3 hours.

Assignment: Long-Run Purchasing Power Parity

1. From Measuring Worth, download the following data.

   - US CPI (annual data from 1774)
   - UK RPI (annual data from 1264)
   - GBP-USD exchange rate (annual data from 1791)

   Alternative data sources:

   - US price level (annual data from 1800) `http://minneapolisfed.org/research/data/us/calc/hist1800.cfm`
   - UK price level (annual data from 1688) `http://www.bankofengland.co.uk/publications/Documents/quarterlybulletin/threecenturiesofdata.xls`

2. The next step is to create a unified data set named `usuklr.csv`. This data set contains the dates for which all three series have no missing values, and the corresponding values of the three series. Note that your data should be in CSV Format and that you should store it as `usuklr.csv` in the *same folder* as your program for this assignment.

   > One approach is to copy your data into your favorite spreadsheet and arrange the data as 4 columns, making sure the dates match. The first line should be a header (in this order please):
   > `date,spot,uscpi,ukrpi`
   > where `spot` is your Exchange Rate. You can then export your data in CSV Format.
   >
   > Another approach is to copy and paste the data into a text file, using your favorite editor to clean it up appropriately. Add a header row as above and any appropriate comment lines.
   >
   > A third (and best) approach is to write a script that reads in your data, cleans it up, and writes out the needed CSV file.

   Question: What are the base years for the price indexes? (Do not take anyone's word for this; check for yourself.)

3. Next you begin creating your program file. Your program will begin by reading in your unified data set.

   As usual, take appropriate steps to protect the raw data. (Treat your unified data set as your raw data from this point forward; make *no* changes to that file.) As usual, make sure you produce an easy to read, fully commented program file. Do not forget the comment lines with your name(s) and the date, as discussed under General Considerations.

   After you load your data into your data analysis application, make sure to examine the result and make sure it looks as it should.

4. Next you will use this data to compute a real exchange rate series. This real exchange rate should be the cost of UK goods in terms of US goods.

   Question: How did you calculate the real exchange rate? What is the meaning (if any) of the level of this real exchange rate? What is the meaning (if any) of changes in this real exchange rate?

5. Next, produce a line graph of your real exchange series. Make a beautiful, appropriately labeled graph. (See Creating Graphs and Graph Annotation.)

Question: What do you learn from your graph?

6. Conduct an ADF test on your real exchange rate. (See Unit Root Tests.) Before adding the code for your ADF test to your program file, experiment with the lag length. (See Lag Length.) In your report, briefly summarize these experiments. You may pick your lag length based on any criterion you choose, as long as you carefully explain it in your report. Also, explain why we might wish to include a constant (intercept) for this test.

7. Produce a clear, commented table summarizing your unit root test.

   Question: What do you learn from this test? Can you reject the presence of a unit root (i.e., is the test statistic larger in absolute value than the critical values)?

8. Construct the first difference of the real exchange rate. Conduct an ADF test on the first difference of the real exchange rate. Go through the same experimentation process to pick a lag length, and again comment on this process in your program file. Once again, add to your program file the commands to produce a beautiful, commented table of your results.

9. As always, email your program file along with your report. (See the syllabus for details.)

5 Real-Interest-Differential Model

Estimated Time for Completion 3 hours.

Objective

- Explore the real-interest-differential model of Frankel (1979).
- Load an ASCII data set and interpret recursive regressions.
- Replicate some key regressions in Frankel (1979). (This paper is available from JSTOR. It is also roughly reproduced in chapter 3 of Frankel (1993).)

1. Review the General Considerations.
2. Download the data as an ASCII text file. Save the data to your `econ672` folder. This is your raw data: you should *never* change it. (See Working Copy.) If you know how to save it as a read-only file, do so. Use your favorite text editor to open the data file and read the information about the data. You will need this information for your homework.

   Comment on editors: some of you are using the Windows NotePad editor. It is perfectly adequate for this viewing need. But learn a better text editor. (I recommend Vim.)
3. Start a new program file. (See Program Files.) As always, start with program comments identifying the author and date. Add code to load your data. (See Data Formats.) Examine your loaded data to make sure it loaded accurately.
4. Ordinarily you would need to transform your raw data to match the series used in a study. This time you are in luck, however: the series have already been transformed, as documented in the `.dat` file. (The downside is: this is the only form in which the data are available. See Isaac and de Mel (2001) for documentation of this data.)

   However the series have awkward names, so let us rename them to get something more convenient, matching the notation used in class. (See New Names for Old). Use the names s, m, y, i, and ilr for the log of the spot rate, the log of the relative money supply, the log of relative income, the interest differential, and the inflation differential (as captured by the long-run interest differential).

   Note: we didn't rename GRLGR as *pi* because in some languages that is a reserved constant. Instead we chose ilr, because that reflects that actual nature of the data used.
5. Now replicate the OLS regression reported in the first table of the Frankel article. We will call these regression results `frankel01a`.

   Question: Discuss your regression results. (See Reporting Estimation Results.) How successful is your replication? What do you learn from your results?

   > If you are referring to chapter 3 of Frankel (1995), you will notice a difference between your estimated coefficients on i and ilr and those reported by Frankel (1979). To understand why, see the last paragraph p.83 and the last full paragraph p.85 of this book.
6. I also want you to run the constrained coefficient OLS regression. We will refer to these regression results as `frankel03a`. (It replicates part of table 3 of the original article.)

Question: How do you impose this constraint? As you think about how to impose the constraint, be sure to examine Frankel's table of results. (What are the right-hand-side variables; what must be on the left? Hint: you can only have one variable on the left.)

Question: Discuss your regression results. How successful is your replication? What do you learn from your results?

7. Finally, let's look at the recursive least squares parameter estimates for *each* of your equations. (See Recursive Least Squares.) Produce a graph of the recursive coefficient estimates on m, y, i, and ilr. Call your graphs `rls01a` and `rls01b`.

   Question: Discuss what you learn from these graphs. How would you assess the "robustness" of Frankel's results, based on your graphs?

8. As usual, you should attach your program file to an email and submit your report as a PDF. Remember, your program file should read in the data and produce *all* of your results. Make everything (tables and graphs) look beautiful!

6 Spot and Forward Rates

Estimated Time for Completion 3 hours.

Objective

- Explore the empirical relationships between spot and forward exchange rates.
- Explore and real-world example where testing for unit roots is crucial to understanding regression results.
- Replicate some key regressions in McCallum (1994). (This paper is available from JSTOR.)

1. This HW uses the data from McCallum (1994). Save the data to your `econ672` folder. Start a new program file named `hw03mccallum` (with an appropriate extension). Add code to load your data. (See Working Copy.)

   Comment: if you are using EViews, you can convert the EViews data into a spreadsheet format. (You need to be at a computer that has EViews installed.) Just click on the data link to open the file in EViews. Once EViews is open, pick `File/SaveAs` from the menus, and save the series you want in Excel format.

   Question: Examine the data I have given you. (See Data Documentation.) Am I offering you in every case the data as it appears in the McCallum paper? How does the series for the British pound differ from the other series?

2. Add code to your program file to produce the GBP-JPY spot exchange rate. (See New Series from Old.) Produce an annotated line graph of this new series. (See Creating Graphs and Graph Annotation.)

3. Change all of your exchange rate series to dollar terms. That is, use the dollar as the quote currency, so an exchange rate is dollars per FX unit. (See New Series from Old.)

   You may reuse the variable names in order to save on notation, but this is a very dangerous practice and should be carefully documented.

For readability, generate the spot exchange rate logarithms as `DEM_s` and `JPY_s`, and the forward exchange rate logarithms as `DEM_f` and `JPY_f`.

4. Conduct a unit root test on (the log of) each of your spot exchange rate series.

   > To produce your tables, consider using a `for` loop. If you are ambitious, you can try to create a summary table of your results.

   Question: What are your unit root test results. What values did you choose for any options, and why?

5. Produce two line graphs, named `yen1` and `yen2`, which explore the behavior of the JPY-USD exchange rate. Let `yen1` plot the yen spot rate and forward rate series; let `yen2` plot the yen spot rate and lagged forward rate series. Then produce a figure that merges the two graphs. (See Merging Graphs.) Make sure the result looks professional.

   Question: Are the results as you expected? Explain. Is there any sense in which these graphs are compatible with a random walk for the spot rate?

6. Add code to your program file to produce a graph named `yen3` that is a scatter plot of the log of the JPY-USD spot rate against the one-period lag of the log of the yen forward rate. (See Scatterplots.) Then produce a graph named `yen4` that is a scatter plot of the rate of spot depreciation against the lagged forward discount for the yen. Finally, combine `yen3` and `yen4` into a single graph, named `yen34`. Produce a beautiful, commented graph containing your plots.

   EViews users can use the `merge` command to combine `yen3` and `yen4` into a single graph.

   Question: What do we learn by comparing your plots?

7. Now we will produce a new series. Suppose we wish to know the JPY-BPD spot exchange rate. We know that this new series, `jpy_bpd`, can be generated from the JPY-USD and USD-BPD spot rates. (See New Series from Old.) Explain how you do this, and why. You should see your new series listed in your workfile.

8. Now we will take at your new `jpy_bpd` series. Produce an annotated line graph of this synthetic spot rate.

9. Do the same thing for the JPY/BPD forward exchange rate.

10. Generate the series of forward rate logarithms (in dollars per FX unit): `dem_lf`, `jpy_lf`, and `bpd_lf`. Conduct a unit root test on each of these forward rate series. What do you find?

11. Next, plot the spot against the lagged forward rates (both as logarithms) for all three currencies.

    As always, be sure to annotate your final (beautiful) graph *and* to add comments to every line of your program file, so that I can see you understand what you are doing. This means, of course, that you will spend some time both thinking about the economics and reading relevant Documentation.

12. Next, generate the (ex post) excess returns `dem_er`, `bpd_er`, and `jpy_er`. Make a figure with line graphs of your three excess return series.

Question: What do the excess returns represent? What do you learn from your plots?

13. Conduct a unit root test on each excess return series. Use an Augmented Dicky-Fuller test on the level of each excess return series. (See Unit Root Tests.) Use four lags, or justify a lag length based on the criterion of your choice.

    Question: Report your results. Should your ADF tests include a constant or trend? (Hint: *look* at the series.)

14. Now generate the forward discount on the dollar versus the three currencies. Produce graphs and tables as you did for the excess return series.

    Question: Compare these two sets of results carefully. (pay attention to the scaling!)

15. Finally, plot spot-rate depreciation against the lagged forward discount for all three currencies. Add to your report a beautiful, annotated graph containing your three plots. What do we learn by comparing this to your plots of the spot rate against the lagged forward rates?

16. Turn in a *commented* script that generates *all* of your results, graphs, and tables. Also turn in a full report (as a PDF) based on your econometric work.

Extra Practice (Not Required)

1. Create a table summarizing your unit root tests for the last exercise.

   Hint: If you look in the Help contents under Command and Programming, you will find a very useful discussion toward the end showing how to automatic the production of such tables.

2. Try to reproduce table 1 in McCallum (1994). What do you learn from these regressions?

3. Use the Hai, Mark, and Wu data at http://qed.econ.queensu.ca:80/jae/forthcoming/hai-mark-wu/ to produce results comparable to Frankel's table 8.1.

4. Apply the method of undetermined coefficients to solve Flood and Garber's model (p.330–331).

5. Consider the function $f(x) = FNL + LNL * x^2$ where FNL is the rank of your first name letter and LNL is the rank of your last name letter. Use starting values of 26|26 and minimize this function.

6. Produce a scatterplot of the nominal exchange rate against purchasing power parities (i.e., relative price levels). Use your long-run data.

7. ML estimation: estimate the nameproblem again but this time by ML. Generate your errors with rndns(). Use seed=20;

8. Augmented Dickey-Fuller tests for real exchange rate series. (Follow Dolado et al. procedure.)

9. Bivariate VAR for exchange rate and relative price levels. Pick optimum lag length. (AIC) Bivariate causality tests.

Past Assignments

Bibliography

Frankel, Jeffrey A. (1979, September). "On the Mark: A Theory of Floating Exchange Rates Based on Real Interest Rate Differentials." *American Economic Review* 69(4), 610–22.

Frankel, Jeffrey A. (1993). *On Exchange Rates.* Cambridge, MA: The MIT Press.

Frenkel, Jacob A. (1976, May). "A Monetary Approach to the Exchange Rate: Doctrinal Aspects and Empricial Evidence." *Scandinavian Journal of Economics* 78(2), 200–224. Reprinted in **?**.

Granger, Clive W.J. and P. Newbold (1976). "Spurious Regressions in Econometrics." 2, 111–20.

Isaac, Alan G. and Suresh de Mel (2001, August). "The Real-Interest-Differential Model after Twenty Years." *Journal of International Money and Finance* 20(4), 473–495.

McCallum, Bennett T. (1994, February). "A Reconsideration of the Uncovered Interest Parity Relationship." *Journal of Monetary Economics* 33(1), 105–132.

Help Topics

A Documentation

For econometric data analysis, there are many commercial alternatives, and a few good free alternatives. Key to a good experience in data analysis is that your software include good documentation.

A.1 EViews

EViews is a state-of-the-art, user-friendly package for time-series econometric analysis. It is widely used in business and government. (So once you learn EViews, you should add familiarity with this package to your cv.)

EViews comes with extensive documentation in the form of two online books: the *Command Reference* and the *User's Guide*. Before starting any work with EViews, you should read chapter 3 ("EViews Basics") of the *User's Guide*. Aside from that, read these resources selectively.

Comment: the *User's Guide* focuses unfortunately on the "point-and-click" approach to data analysis, which is a very bad approach to serious empirical research. (Specifically, it renders replicability impossible.)

Every EViews command introduced in these homeworks is discussed in the very good EViews Command Reference. To access the Command Reference from the EViews menus, pick `Help/Command Reference`. Be sure to read about *each* command before you use it.

EViews also comes with a very helpful EViews User Manual. Before starting any work with EViews, you may wish to read Part III (*Commands and Progrgamming*) of this manual. From the EViews menus, pick `Help/User Manual`.

B Program Files

A program file is just a text file containing executable commands (statements) and comments. Generally, you can create a program file with the text editor of your choice. However many econometric applications include an integrated development environment (IDE), which includes an editor.

Most essentially, your program file is a sequence of executable commands (statements). If your software includes an interactive interpreter (command line), the commands are usually exactly the commands that you would enter at the command line.

Short, simple programs are often called "scripts", especially if they are written for an interpreted language. So a script is just a simple program.

Generally a script contains commands that you could enter at an interactive interpreter. However if you work directly at an interpreter, then once you discard the interpreter session, your work is lost. For this and many other reasons, it is good practice to store the commands you are using in a program file. This is just an ordinary text file that you can create with any text editor. It contains commands and comments in the command language that you have chosen.

B.1 EViews

The EViews IDE includes an editor. While you are not required to use this editor, most EViews users find it convenient to do so. You create a new program file by picking `File/New/Program` from the EViews menus. After entering some commands, use the buttons on your *Program* window to `Save` and then `Run` your new program.

B.2 gretl

The gretl IDE includes an editor. While you are not required to use this editor, most gretl users find it convenient to do so. You create a new program file by picking `File/Script files/New script` from the gretl menus. After entering some commands, use the buttons on your *Program* window to `Save` and then `Run` your new program. (If you hover your mouse over the buttons you will see helpful text.)

C Working Copy of the Data

It is an extremely important habit to *never* alter your raw data, so keep any datafiles I gave you inviolate. Instead of working with the file provided, work with a copy. You can use the operating system to do this: just copy the data file to a new name, which you will work with. Or you can use your application to make the working copy, as follows.

C.1 gretl

After opening your data, add two additional lines to your script, which saves the data as `temp.gdt` and then opens that was your working data file.

```
open ..\data\rawdata.gdt
store ..\data\temp.gdt
open ..\data\temp.gdt
```

If you save and run this script, you should see this new name in the title bar of your datafile window. Now all the changes we make to the data will show up in `temp.gdt` instead of in the datafile I gave you.

D Program Comments

Programs without comments are generally useless to others. They even become useless to yourself as time passes and you forget what your problem was and how you were approaching it. So all programming languages provide a way for you to add comments to your programs. One standard approach is to designate a special comment character; nothing following this character is treated as executable code. Comments provide assistance to humans who read your programs. They are not commands to be executed.

D.1 EViews

All the text following an apostrophe is a comment: EViews ignores this text when running your program. Example::

```
'This is an EViews comment line. So is the next line.
' <-- Note the apostrophe, which makes this a comment.
```

D.2 gretl

All the text following a pound sign is a comment: gretl ignores this text when running your program. Example::

```
#This is a gretl comment line. So is the next line.
# <-- Note the hash mark, which makes this a comment.
```

D.3 Python

All the text following a pound sign is a comment: Python ignores this comment when running your program. Example::

```
#This is a Python comment line. So is the next line.
# <-- Note the hash mark, which makes this a comment.
```

E Creating Graphs

E.1 gretl

gretl provides powerful 2D and 3D plotting capabilities via gnuplot (which is included in the gretl installation). Read about the `gnuplot` command and about `graph` objects in the gretl Manual (User's Guide and Command Reference).

Suppose we have two variables, 'x' and 'y'. Here we declare a graph named `g1` and then fill it with a line graph of 'y' against 'x'. We then display it with it's `show` method. ::

```
g1 <- gnuplot y x --with-lines
g1.show
```

> Looking in your gretl icon view window, you will see a graph named `g1`. To view it, double click on it. Close it by clicking the usual close button for its window. Note that if you close your session without first saving it, you will lose this graph object. (That is fine however: you have saved your script, which produced it!)

Right click on the graph for a variety of options, including printing and saving in various file formats. (The PNG format is recommended for web documents.) You can also copy it to the clipboard and then paste it into other applications. (Windows users may wish to paste with the `Edit/Paste_Special` menu and paste it as an enhanced metafile.)

You can have more than one y variable, say 'y1' and 'y2'. Also, if your x variable is time (e.g., for time series), you can use the gretl keyword `time`. Finally, you can pass additional commands to gnuplot inside braces. (The gretl Command Reference provides detailed documentation of the `gnuplot` command.) The following creates a graph object and also writes the "same" graph to a PNG file. ::

```
g1 <- gnuplot y2 y1 time {set title 'Your Title';}
gnuplot y2 y1 time --output=temp.png {set title 'Your Title';}
```

For fine control of your graphs, you will need to learn some gnuplot commands. Check out the gnuplot manual, especially the section set-show. For example, we could change our previous graph to use a log scale on the vertical axis as follows::

```
gnuplot y2 y1 time {set title 'Your Title'; set logscale y; }
```

E.2 EViews

EViews has fairly powerful interactive graphics capability. Much but not all of this capability is exposed via attributes of `graph` objects. Read about `graph` objects in the EViews online Command Reference.

In EViews, you can produce a line graph with the `line` procedure. For example, if you have a series named `cad_s`, you can produce and display a line graph with the following command::

```
line cad_s
```

While you can readily manipulate this graph with a mouse, it is usually a better idea to produce your graphs with code. Here is the same graph produced as a named graph object::

```
graph g1.line cad_s
show g1
```

The first command declares a graph named `g1` and then fills it with a line graph of your series. The second command displays this graph. The EViews Command Reference documents many capabilities of this graph object. For example, to switch the axes, you could to the following::

```
g1.setelem(1) axis(b)
```

In EViews, you can use `line` procedure to plot multiple series in a single line graph. ::

```
graph g3.line cad_s relp
```

Comment: Close your figure by clicking usual close button for its window. Looking in your workfile, you will see a graph named `g3`. To view it again, double click on it. However if you close your workfile without first saving it, you will of course lose this graph. (That is fine: you have your program, which produced it!)

Comment: If your program creates several figures and shows them, the last one showed will be the top window. To reverse this, at the end of your program, you can give show commands from last to first. E.g., if you have created `fig1`, `fig2`, and `fig3`, add the following lines at the end of your program. ::

```
    show fig3
    show fig2
    show fig1
```

E.3 Matplotlib

Matplotlib is a very powerful 2D plotting package for Python. There is excellent online Matplotlib documentation. At least initially, you will probably rely on Matplotlib's `pyplot` module, so it is a good idea to start with the pyplot tutorial. Do not forget that to use `pyplot` you need to import it::

```
import matplotlib.pyplot as plt
```

In Matplotlib, you can produce a line graph with the `plot` command. For example, if you have a series named `cad_s`, you can produce and display a line graph with the following commands::

```
plt.plot(cad_s)
plt.show()
```

While you can readily manipulate this graph with a mouse, it is usually a better idea to produce your graphs with code. Here is the same graph produced as a named graph object::

```
g1 = pyplot.fig()
ax1 = f1.gca()
ax1.plot(cad_s)
pyplot.show()
```

F Merging Graphs

F.1 EViews

You can use the `merge` command to merge graphs. For example, to merge two garphs named `g1` and `g2` into a graph named `g12`::
graph g12.merge g1 g2 show g12.align(2,.5,0)

G Graph Annotation

Your annotations can include *very* brief explanatory text, so a viewer can know what you have calculated and graphed. Your graph should be understandable without reference to the text, but it should not be cluttered.

G.1 EViews

In EViews, there are two ways to annotate your graphs. (Try both!) The first is quick and dirty, and you should seldom use it. (It is included here just so that you know about it.) Just click `Add Text` and (ahem...) add text. You can put this information anywhere you wish on the graph, but you might try clicking `Top, Center` to get started. (To change the settings, you can just double click the text in your graph.) Click `OK` when you finish, and you can print your finished product.

The second way is a reliable way to produce graphs that you can fully reproduce: use the `addtext` command. For example, to put your name and in the graph, you might use the command ::
g1.addtext(0,-.5) Your Name Date
Use the `addtext` command to place your name, the date, and a very brief descriptive title on your graph. (You may put longer remarks in your program file as comment lines.)

- By the way, don't overlook the possibility of negative numbers when you choose where to `Position` your added text.
- If you want to change the way your graph prints, click the `Print Setup` button on the graph. For full size graphs, make sure you've checked the option `Scale to Page`. You can make it even bigger by choosing `Landscape` instead of `Portrait` graph orientation.
- Named graphs will be saved as part of the workfile whenever you `Save` your workfile. How ever if you overwrite this file (e.g., with the `Save` command you are using in this program), the graph will be lost. If you are using your program file to annotate the graph, of course, then you can always reconstruct the graph by running the program file.

G.2 gretl

In gretl, there are two ways to annotate your graphs. (Try both!) The first is quick and dirty, and you should seldom use it. (You should use it only when you will not ever need to reproduce the graph. Never use it for professional work that you will present to others.) Just click the graph and choose `Edit` to bring up the gretl plot controls, and fill in whatever text you wish. Click `Apply` when you finish, then `Close` the plot control window.

The second way is a reliable way to produce graphs that you can fully reproduce: add text using a useful option to the `gnuplot` command. For example, to add a title to your graph, change the gnuplot command to::

```
g1 <- gnuplot spot_ca time --with-lines \
      {set title 'Your Title'; }
```

The backslash, followed by no spaces, is a line-continuation symbol: it tells gretl to ignore the subsequent end-of-line. You could put the entire command on a single line instead, but it would be harder to read.

> Named graphs will be saved as part of the workfile whenever you `Save` your workfile. However if you overwrite this file the graph will be lost. If you are using your program file to annotate the graph, of course, then you can always reconstruct the graph by running the program file.

G.3 Matplotlib

The pyplot tutorial has a discussion of pyplot annotation.

H Data Documentation

H.1 EViews workfiles

Suppose you have a documented series `series1` in an open EViews workfile. In the workfile window, double click on the series icon, `series1`. A window opens displaying one of several views of this series: we want to see the `label view`. On the series window, click `View, Label`. You will see the information that the series creator associated with this series.

H.2 gretl XML

Suppose you have a documented series `series1` in an open gretl workfile. Each series may be given a descriptive label in addition to its name, and you will see this label displayed next to the series name. If you double click on the series name, `series1`. A window opens displaying the data values of this series.

H.3 Databank Files

Databank (`.db`) files provide a special comment format for series documentation. See http://www.american.edu/econ/pytr
for details.

H.4 CSV Format

Most programs have the capability of reading data from CSV files. CSV files do not have a specific provision for data documentation. Some CSV files will nevertheless provide some documentation in the first few lines of the file. See for example the CSV files produced by the IFS. Generally you will need to instruct your program to skip comment lines, with the possible exception of a line containing names for the series.

I New Series from Old

There are many ways to assign values to a series. Two of the most common are:

- Assign values resulting from the mathematical manipulation of other series. - Assign values element by element.

A basic question that arises when you want to create new series from existing series is whether you can manipulate the series as objects or must explicitly create each element of your new series. From example, if you have series `x1` and `x2`, can you create a new series as `x1 + x2`. Most econometric applications define element-by-element operators for series that allow you to do this.

When working with similar data for multiple entities, such as time-series for various countries, it can be useful to perform analogous operations for each entity. Generally this will involve some kind of looping construct. (However EViews `pool` objects are one convenient and fairly readable solution to this need; see below.)

I.1 gretl

A gretl series contains an ordered set of observations on a single variable. Associated with each observation in the series is a date or observation label. For series in dated workiles, the observations are presumed to be observed regularly over time. For undated data, the observations are not assumed to follow any particular frequency.

In gretl you can use the `genr` command to create a new series. (Read about the `genr` command in the gretl Command Reference.)

Suppose we have an active workfile in which we wish to create a new series named `x3` as the ratio of `x1` and `x2`. We do this by declaring the series with the `genr` command::

```
genr x3 = x1 / x2
```

To look at what we have created, examine the "spreadsheet view" of this series. This is done by double-clicking the series.

Suppose we have the price levels `cpi95_ca` and `cpi95_us` and an open workfile, and we wish to create the relative price level. We can do so with the following commands. ::

```
gen4 relp = cpi95_ca / cpi95_us
setinfo relp -d "Relative Price Level" -n "P/P*"
```

This code declares a new series named `relp`, calculates the relative price level, and assigns the relative price level to `relp`. We also provide an informative displayname (`-n`) for the new series, which will show up in the legend of graphs that use this series, and a brief description (`-d`) of the series. Once you execute these commands, you will see your new series listed in your workfile. (If you do not see the new series immediately, try picking from the gretl menus: `Data/Refresh_window`.) Note that the division operation will be done element-by-element for every observation where neither `cpi95_ca` and `cpi95_us` have a missing value.

We can also assign values element by element. Suppose we want to set the first element of `x1` to the value 15. We do this as follows:: `x1(1)=15`.

I.2 EViews

An EViews series contains an ordered set of observations on a single variable. Associated with each observation in the series is a date or observation label. For series in dated workiles, the observations are presumed to be observed regularly over time. For undated data, the observations are not assumed to follow any particular frequency.

In EViews you can use the `series` command to create a new series. (Read about the `series` command in the online Command Reference.)

Suppose we have an active workfile in which we wish to create a new series named `x1`. We do this by declaring the series with the `series` command::

series x1

To look at what we have created, examine the "spreadsheet view" of this series. This is done with the `sheet` command. ::

```
x1.sheet
```

The `sheet` command causes EViews to open a spreadsheet view of the new series object. Note that all of the elements of the series have been assigned the missing value code `NA`. This is because we have not yet assigned any values to this series.

There are many ways to assign values to the elements of a series. Two of the most common are:

- Assign values resulting from the mathematical manipulation of other series. - Assign values element by element.

Suppose we have the price levels `cpi95_ca` and `cpi95_us` and an open workfile, and we wish to create the relative price level. We can do so with the following commands. ::

```
    series relp
    relp = cpi95_ca/cpi95_us
    relp.displayname Relative Price Level
```

This code declares a new series named `relp`. It then calculates the relative price level and assigns it to `relp`. Finally it provides an informative displayname for the new series, which will show up in the legend of graphs that use this series. Once you execute these commands, you will see your new series listed in your workfile. Note that the division operation will be done element-by-element for every observation where neither `cpi95_ca` and `cpi95_us` have a missing value.

We can also assign values element by element. Suppose we want to set the first element of `x1` to the value 15. We do this as follows::

x1(1)=15

Pool Objects """"""

A `pool` instance allows us to use single commands to do data transformations base on multiple series. (Read about `pool` objects in the EViews *Command Reference*.)

To declare a pool object, use the pool keyword, followed by a pool name, and a list of pool members. (You can always `add` or `delete` pool members later.) ::

```
pool g6 us uk fr de it jp
copy g6 g7
g7.add ca
```

Note that pool members are short text identifiers for the cross section units. These allow you to operate on multiple workfile series with a single `genr` command. ::

```
g7.genr rgdp_? = gdp_? / p_?
```

The `genr` method simply loops across the cross-section identifiers, in order, using them to substitute for the question marks in the expression. After each substitution, the evaluations are evaluated. So the `genr` statement above first computes `rgdp_us`, then `rgdp_uk`, etc.

J Data Formats

J.1 EViews

EViews works with some specialized data formats. The format most commonly used by new users is the EViews workfile, which has a `.wf1` extension. Since this is a proprietary EViews format, there is a native command that makes it trivial to open. Suppose your data is in the file `G:\gh372.wf1`. You can open it as follows::

```
wfopen G:\gh372.wf1
```

EViews will open a workfile window titled `gh372.wf1`, which displays the series in the data set. (See Working Copy.)

If the data is in a different format, you may have to work a bit harder. Let us consider how to load an ASCII data set. Make sure you read about `read` in the EViews Command Reference. For concreteness, we will work with the data from the Real-Interest-Differential Model exercise.

Recall that EViews stores the data you are directly working with, along with the objects you create with this data, in a *workfile*. You can create a workfile to hold your data with the `wfcreate` command. For example, ::

wfcreate frankel79 m 1974.01 1978.02 wfcreate temp m 1957.01 1978.02 2009.01

This command creates a new workfile named `frankel79`. The sample frequency is monthly (`m`), and the sample is `1974.01--1978.02`. (Note that the sample differs slightly from the sample used by Frankel.) This command could just be typed at the EViews command line. However, instead of typing your EViews commands at the command line, I want you to keep them all together in a program file.

Start EViews and pick from the menu `File`, `New`, `Program`. You will see a new program window open. In that window, type the code creating your new workfile.

To read your data into EViews, use the `read` command. Once again, you should put this command into your program file. ::

```
'sample code for ‘read’ command
read(t=dat,rect,na=.,skiprow=36,label=3,d=s,mult) G:\frankel79aer.dat 8
read(t=dat,rect,na=n.a.,d=c,skiprow=10) c:\temp\temp2.csv date Pus Pca spot
```

This `read` command tells EViews to read the data from the file `G:\frankel79aer.dat` into your current workfile. There is a long list of options included in this `read` statement; make sure you have understood why they describe your data. (I.e., read the Command Reference. The 'read' command is described in full detail in the Eviews online Command Reference.) The option `skiprow=36` is required because of all the information that I included in the data file; we need to skip the 36 rows before the labels. The option `label=3` sets the series header offset to 3: the line of labels and the two lines after the label header but before the data. Add *detailed* comments to your .prg file that fully explain this `read` command, specifying the role of each option. Click the `SaveAs` button, and save your program as `frankel79.prg`. Then click the `Run` button and run your program.

K Working Copy

A ”working copy” of the data is the copy you work with and possibly change. (The raw data, in whatever form you receive it, should never be changed.)

K.1 Read Only Attribute

One way good way to protect your raw data is to set the ”read only” attribute on the data file. For example, if your data is in `myrawdata.dat`, at the Windows command line you can give the following command. ::
attrib +r myrawdata.dat

K.2 EViews

Suppose your raw data is `G:\myrawdata.wf1`. The following line in your program will this data set in EViews as the current work file. ::

```
wfopen G:\myrawdata.wf1
```

Recall that we never want to alter the file containing our raw data. Let us save this data into a new file and change the name of the active workfile with the `save` command, and then for extra safety close the workfile containing our raw data. ::

```
wfsave gh_hw.wf1
close myrawdata
```

L Lagged Values

Working with leads and lags can be a little complicated becase it shorten the effective length of your data set.

L.1 EViews

Eviews makes is very easy to work with lags and leads: it automatically adjusts the sample to work only with the valid periods. Suppose your have a series `f` in your active workfile. Then you form the one-period-lagged series as `f(-1)`. The `(-1)` gives you the one-period-lagged value of `f`. (In the EViews help index you can find `lagged series`: read about these.)

M Scatterplots

A 2D scatterplot plots with symbols the corresponding values of two series. So a scatterplot is equivalent line graph with symbols turned on and lines turned off. Multiple scatterplots are sometimes placed in a single graph.

M.1 EViews

Read about `scat` in the Command Manual. The are many variants and options, so you may need to play with this a bit.

The simplest syntax is::

```
scat(options) s1 s2
```

This can be useful for data exploration at the interpreter prompt, but does not give you access to a graph object. The simplest syntax that does is::

```
graph figure1.scat(options) s1 s2
```

This plots series `s2` against series `s1`. You can have more than two series. By default, the first series or column will be located along the horizontal axis, and the remaining data on the vertical axis. For example, with multiple series, Eviews allows you to optionally produce a scatterplot for every pair of series.

Often it is more useful to form a group (say `group1`) and then use its `scat` method. This gives you access to auxiliary specifications, like `linefit`. Let's illustrate this with the similar `scatpair`, which generates a scatterplot for each successive pair of series. ::

```
group group1 f s m s
freeze(fig1) group1.scatpair linefit
```

Note that to produce a graph object `fig1`, we freeze the `scatpair` "view" of this group `group1`. The optional auxilary specification adds fit lines to the scatterplot. In this case, it is regression lines, but you can optionally use kernel fit, nearest neighbor fit, orthogonal regression, or confidence ellipses.

N Graph Formats

Common graph formats and associated extension include: Portable Document Format (.pdf), Encapsulated PostScript (.eps), Enhanced Windows metafile (.emf), Windows metafile (.wmf) bitmap (.bmp), Graphics Interchange Format (.gif), Joint Photographic Experts Group (.jpg or .jpeg), and Portable Network Graphics (.png). For web viewing, PNG is the current natural choice.

N.1 EViews

See the `save` command for graph objects in the Command Reference.

EViews can automatically determine the filetype from the extension if you use the following extensions: .emf, .wmf, .eps, .bmp, .gif, .jpeg, or .png. (You can also specify file type with the `t=` option.)

For example, to save your graph `g1` as a four-inch wide PNG file named `temp.png`, use the following command::

```
g1.save(w=4,t=png) c:\temp\temp.png
```

Note that you need to specify the filetype *and* the full path where you want your file located.

O New Names for Old

When working with a lot of data, it is crucial to avoid duplicating data each time you wish to refer to it by a new name. With large data sets, needless duplication may waste enough memory to be a real computational problem.

O.1 EViews

EViews provides the `rename` command to rename series without copying them. For example::

```
'for convenience, rename variables
rename GRM1 m
```

P Least Squares Regression

P.1 EViews

In EViews, you can use the "least squares regression" command `ls` to do this. The simplest syntax for a least squares regression is::

```
ls y c x1 x2
```

This command causes EView to create an "unnamed" equation object, which is fine for quick work at the interpreter but not very helpful otherwise. It is more useful to declare an equation obect to hold your estimation results. ::

```
equation eq01
eq01.ls y c x1 x2
```

(Read about `equation` objects in the Command Reference.) Conveniently, you can do this in a single step::

```
equation eq01.ls y c x1 x2
```

For example, your replication of the Real-Interest-Differential Model might include the program line::

```
equation frkl4c.ls s c m fd
```

Q Recursive Least Squares

Q.1 EViews

Recursive least squares is discussed in the *User's Manual*. As usual, you will want to read about the `rls` command in the *Command Reference* before you use this commands.

For example, suppose you have a regression equation `frankel01a`. The you might create a graph of the recursive least squares coefficient estimates as follows::

```
'graph the recursive coefficients for m,y,i,ilr
freeze(rls01a) frenkel01a.rls(c) c(2) c(3) c(4) c(5)
```

R Reporting Estimation Results

Your estimation results should be reported in an informative, professionally formatted table.

R.1 EViews

An equation object has an `results` attribute or "view". (The `results` attribute is identical to the `output` attribute. Read about `equation` objects in the EViews Command Reference.) We can use the freeze command to produce a named table from the estimation results. (Read about the `freeze` command in the EViews Command Reference.) For example, given an equation object named `eq01`, we can produce a table named `eq01out` as follows::

```
freeze(eq01out) eq01.results
```

One we have created this table, we can manipulate it and add information using the `setcell` command. (Read about the `setcell` command in the EViews Command Reference.) E.g., ::

```
setcell(eq01out,1,3,"Your Name","l")
```

Note that if you are viewing a table in EViews, you can select and copy its contents and paste these to another application. You only get plain text this way. You can also `save` the table in a variety of formats. ::

```
eq01out.save(-f,t=csv) c:\temp\temp.csv
eq01out.save(-f,t=html) c:\temp\temp.htm
eq01out.save(-f,t=txt) c:\temp\temp.txt
```

Unfortunately, the EViews table headers use multicolumn entries, which CSV does not support. So the potentially most useful format is in fact not all that useful. However you can open the text format in an editor, or the HTML format in a browser, and then copy and paste the result in your document. (Sadly and inexcusably, EViews 6 has a couple bugs in its HTML export for user modified tables.)

The recommended technique for now is export as HTML, open in a browser, and copy to your Google Doc.

S IFS Online

International Financial Statistics (IFS) is a classic source for international macroeconomic data. The library provides online access to this database. For those unfamiliar with the database, here is an example of use.

1. Go the the IFS online.

2. Click the `Change Button` for the Retrieval Period. (After each of the following steps, click the `Next` button.) Pick Single Frequency radio button. Pick Monthly. For Start, pick 1957 M1. For End, pick the most recent date possible.

3. Click "Country Tables". Pick Canada. For exchange rate, pick end of period market rate. For Price level, pick Consumer Prices. Click `+` to add them.

4. Go to the US Country table and get Consumer Prices. Click `+` to add this series.

5. Click Retrieve. Check the format you want (e.g., Comma Separated Value). Click Retrieve. Download your data file.

S.1 gretl

Be sure to read "Creating a data file from scratch" in the gretl User Guide.

Once you have saved your IFS data to disk, you need to prepare it for import into gretl. If you downloaded it in a spreadsheet format (such as .xls or .ods), gretl may be able to `open` it directly. If you choose CSV format, you will need to clean it a little for gretl. To do so, you may use a spreadsheet or a text editor. The easiest way is to use a spreadsheet.

1. Import `myrawdata.csv` into your spreadsheet (E.g., Excel or OO Calc.)

2. Save your spreadsheet as `temp.csv` (so that you do not change your raw data).

3. Delete the first 10 rows and first column.

4. Insert a new first row, and put in your variable names (no spaces!).

5. Save your spreadsheet. (Keep the CSV format!)

Alternatively, you can prepare your data using any text editor (e.g., Notepad). If you know what you are doing, this is often the fastest and most convenient method.

Once your data are ready to import, import them and set the data range. You may also wish to store the data in gretl format for future ease of use. ::

open temp.csv setobs 12 1957:1 store temp.gdt

Comment: if you did not provide variable names in the first line of your file, variable names `v1`, `v2`, etc will be assigned for you. You can `rename` these.

Comment: if you open a spreadsheet file that is in native format (e.g., an Excel file or a OOCalc file), you can use the `--rowoffset` option to `open`.

S.2 EViews

You will need to read your new data into a new workfile. Read about the `wfcreate` and `read` commands in the Command Reference. Then use the following code::

```
wfcreate temp m 1957.01 2020.01
read(t=dat,rect,na=n.a.,d=c,skiprow=10) c:\temp\temp2.csv date Pus Pca spot
```

Be sure to examine your data to make sure you read them in correctly. Be sure to add extensive program comments to explain how the code works: comment on *each* option used.

T Random Number Generators

A random-number generator (RNG) produces numbers that appear to be draws from a random distribution. This appearance is somewhat misleading, since most RNGs produce a sequence of numbers by a deterministic algorithm. We sometimes call them *pseudo* random-number generators (PRNGs) if we wish to emphasize this.

Before computers were widely available, scientists needing random numbers turned to random-number tables. Published random number tables exist as early as 1927. Using such a table, one would "at random" pick a starting point, and then read through the table in a deterministic many. E.g., one might begin with the top left number on say page 69 and then read sequentially down column and down the subsequent columns. Such a procedure produces a determinant set of "random" numbers, and by providing the starting point and procedure you can ensure that other scientists work with the same sequence of "random" numbers.

Nowadays, the starting point is picked by providing a "seed" value to the random number generator. For the experiments you will do, I will supply the value for the seed: 314159. This is just to make sure that everyone's program output looks exactly the same. Seed your random number generator *once*. Seeding your random number generator is like specifying a page and line number in a book of random numbers.

T.1 EViews

Seed the EViews RNG as follows::

```
rndseed 314159
```

Here is an example of creating a workfile and a series of random numbers in that workfile. ::

```
create u 300
series wn1 = nrnd
```

U Descriptive Statistics

Econometric software provides simple ways to examine any variable in terms of a standard collection of descriptive statistics.

U.1 EViews

You can view some descriptive statistics for a series `x` by giving the command `x.stats`. The command `x.hist` shows the same descriptive statistics but adds a histogram.

V Cumulative Sum

A cumulative sum is just what it sounds like. For example the series [1 1 1 1 1] has a cumulative sum of [1 2 3 4 5]. As another example, the series [1 2 3 4 5] has a cumulative sum of [1 3 6 10 15]. Note that the cumulative sum of a series is also a series, of the same length. The i-th element of the cumulative sum is equal to the sum of the first i elements in the original series. You can form the cumulative sum of any series of numbers.

V.1 EViews

The following creates a 300 element series named `seq1`, where every element is 1, and then form `cumseq1` as the cumulative sum of `seq1`. (Try it, and check your results visually.) ::

wfcreate cumsumdemo u 1 300 series seq1 = 1 series cumseq1 = @cumsum(seq1)

Older versions of EViews did not have a `@cumsum` function, which meant that a cumulative sum had to be generated iteratively. Let us do that for a more general case, where the autoregressive parameter `rho` might other values than unity. ::

'declare new series named ar1 series ar1 'assign to ar1 the values of wn1 ar1 = wn1 'set a value for rho (autoregressive parameter) scalar rho = 0.9 'change sample to miss first observation smpl @first+1 @last 'set ar1 = weighted cumulative sum of white noise ar1 = ar1 + rho * ar1(-1) 'restore full sample smpl @all

W Unit Root Tests

Unit root tests are used to distinguish stationary from nonstationary series. The Augmented Dickey-Fuller (ADF) test remains the one of the most popular unit root tests.

Note that you must make some choices when using the ADF test. You must decide whether to include a constant and/or trend. You must also decide how many lags (of the differenced dependent variable) to use.

W.1 Some Details

W.1.1 Stationary and Nonstationary Series

We are interested in distinguishing two types of time series: those that are only temporarily affected by any random change, and those that are permanently affected. We will call these stationary and nonstationary time series. In order to draw this distinction, we will work with very simple examples of each: white noise and random walks.

We define a random sequence to be a *white-noise process* if it is characterized by:

- zero mean - constant variance - no serial correlation

White noise is stationary: we say innovations are transient, or equivalently that shocks are temporary. A unit root test is a test of whether the process is stationary (shocks are temporary) or non-stationary (shocks are permanent). When shocks are permanent, we say the series has a *unit root*.

First consider how to conduct an augmented Dickey-Fuller (ADF) test given a fixed lag length. (We will discuss lag length selection later.) We use the ADF test to determine whether we can reject the null hypothesis that the series has a unit root. To conduct the test, we compare the ADF test statistic with the critical values. If the ADF test statistic is close to zero, then we cannot reject the null hypothesis. In this case we accept that shocks are permanent and the series has a unit root. If the test statistic is far enough from zero, so that *in absolute value* it is larger than the critical values, then we reject the null hypothesis.

W.1.2 Lag Length

Many AR models are sensitive to the lag length chosen. You will need to pick a lag length based on some criterion. One popular approach is minimization of the Akaike information criterion ("AIC"). Another popular approach is minimization of the Schwartz information criterion ("SIC"). The SIC tends to pick short lag lengths.

W.2 gretl

In gretl, you can perform an ADF test with the `adf` command. (Read about this in the gretl Command Reference.)

Before using the `adf` command in your program, experiment with it using the point-and-click facilty. Here is an example. Highlight the series you wish to test in the main gretl window. Click the `Variable` button in the menu, and then pick `Augmented Dickey-Fuller Test`. Using the check boxes, include an intercept but no trend. Set the `lag order` to 4 lags. Click `OK`.

W.3 EViews

In EViews, you can perform an ADF test with the `uroot` command. (Read about this in the EViews Command Reference.)

Before using the `uroot` command in your program, experiment with it using the point-and-click facilty. Here is an example. Click the `View` button in the Series window, and pick `Unit Root Test`. Do an `Augmented Dicky-Fuller` test on the `Level` of the series, with an `Intercept` and `4 lags`. Click `OK`.

Using the `lag` option, you can set the number of lags in the ADF test. You can also let `lag=a` for automatic lag-length selection. (You can set the criterion for the selection with the `info` option.)

Comment: You can produce an editable table of you ADF output using the point and click method. To do so, click the `Freeze` button and then use the `Edit` button to add your commentary. If you save your workfile, this table will be saved there. Of course when you come back to it, you will not know *how* you produced it. That is why you should use an EViews program to do this, using the `freeze` and `setcell` commands.

X Debugging

Finding a mistake is your scripts can be difficult and frustrating. Resist the impulse to blame your computer or the program: with very rare exceptions, they are doing exactly what you tell them to do. The most important step in debugging is to assume that you are overlooking something that you are doing wrong, perhaps something as simple as forgetting punctuation (colon, semi-colon, parenthesis).

X.1 gretl

gretl has fairly modest facilities for debuging scripts, but the ability of its editor to execute code regions is very helpful.

Open your gretl script file in the gretl editor and start executing the script *one line at a time.*

Here is how to do that: using the mouse, select the line of your script that you want to execute, then *right* click on your selection to get a context menu, and then click `Execute region` to execute your selection. (This process also works for more than a single line, so you can always execute lines that you have already debugged all at once as a single region.) Alternatively, you can just put the cursor in a line of your script and press `crtl+Enter`.

Each region you execute produces output in a Window called 'gretl script output'. When you reach a line in your program with a problem, gretl will print an error message in this window. Read the error message carefully and reexamine the buggy line in your code. Usually, gretl provides enough information for you to figure out the problem.