THE INTERDYME COOKBOOK

International Model Building Cuisine

Explore the world of Inforum model building using powerful free software!

Dr. Douglas S. Meade

IWC 26 – August 2018

THE INTERDYME COOKBOOK

International Model Building Cuisine

Operational proof … it's all theory until you see for yourself whether or not something works. *Julia Child*[1]

This guide to *Interdyme* model building accompanies Clopper Almon's text *The Craft of Economic Modeling*, which is a three part textbook. This 'Cookbook' is designed for self-teaching, although the help of a trained chef/teacher is always a plus!

Interdyme was designed to help build models that use vectors, matrices and scalar variables (macrovariables) in a dynamic setting, solving annually for a time frame as short as 5 years to as long as 100 years. Some call this type of model an *interindustry macroeconomic* (IM) model. In a typical model of this kind, the input-output (IO) quantity and price calculations form the core of the model, with projections for final demands, employment and value added usually being calculated using regression equations. Macrovariables can be handled flexibly, in a way that easily leads to the combination of regression equations and identities.

The materials needed to work through the Cookbook are not as big or as expensive as those required for real-life cooking. All of the software and data used here are available in the form of a datastick or thumb drive, either provided directly or downloadable. All you need in addition is access to a computer. At present (2018) the software still only works under Windows, although work is underway to port to Linux and Mac environments.

The material is presented in a set of roughly 90 minute sessions, and is meant to be read while doing.

SESSION 1. KITCHEN TOOLS

Always start out with a larger pot than what you think you need.

The main tool in our well-stocked kitchen is *G7*, which in the words of Julia Child above, is a "large pot". *G7* is a program designed for building models; estimating regressions; developing databanks of scalar, vector and matrix variables; and comparing scenarios and/or counterfactual historical simulations using plots and tables. *G7* is available for free from the Inforum website, and is under continuous development.[2] Its current features owe their presence to the many ideas, wishes and work of current and past Inforum staff, international partners, students and other users.

The use of *G7* for developing macro models is described in *The Craft of Economic Modeling, Part I*, which is used in a course for teaching macro modeling at the University of Maryland. *Part II* describes a fully operational model known as *QUEST*. *Part III* describes the use of *G7* and *Interdyme* to develop Interindustry

---

[1] All quotes below chapter or section headings are taken from Julia Child, who had many nuggets of wisdom for both the cook and the model builder. Google "Julia Child quotes".

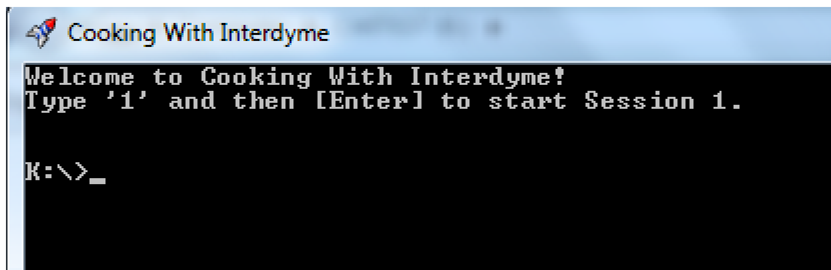
[2] http://www.inforum.umd.edu/software/g7.html . Extensive documentation is also available there, which is viewable as either html, pdf or windows help files.

macro and other multisectoral models.[3] However, you have what you need right in front of you. Let's get started!

1.1 Getting Started and Viewing Data

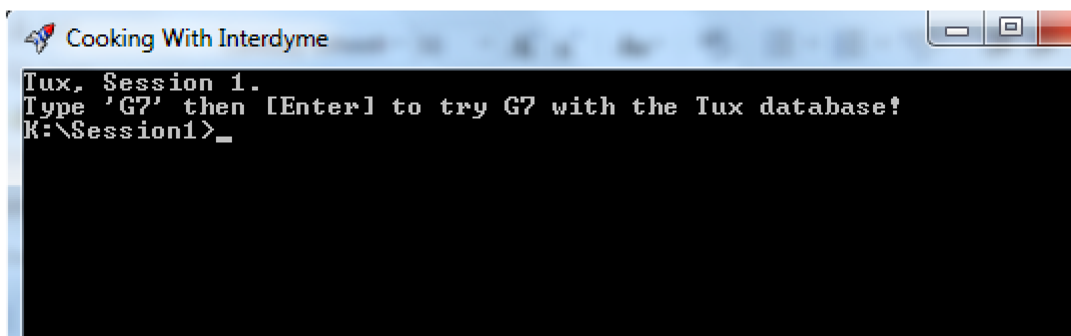
Insert the datastick into your computer. We will assume that it shows up on your computer as K:. Find the datastick in Windows Explorer and double-click on the Launch program (Rocket icon) at the top level. You are now in the 'Command Prompt' application of Windows. The Launch program has now set up a convenient environment for these sessions.

Figure 1.1. Initial Launch Window



Now, type '1', and then the [Enter] key on your computer, and you'll be taken to the \Session1 folder. This folder has an IO databank and working forecasting program for the *Tux* (**T**iny **U**nderstandable e**X**tensible) model of the U.S. This version has 9 sectors. Another version on the datastick, which you may explore later, has 17 sectors[4]. These data were aggregated from the Inforum U.S. IO model named *Lift*.

Figure 1.2. Session 1 Folder



As instructed, type 'g7' and then hit the [Enter] key. You have just started *G7* for the first time!

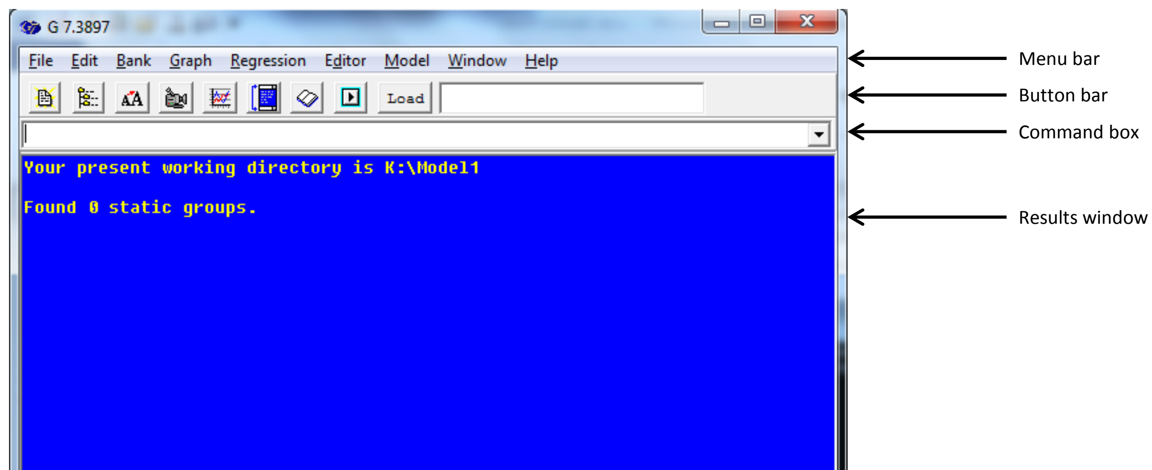
Figure 1.3 shows the main components of the *G7* application when it first starts. Many of the features of the program are controlled by menu choices or buttons. The full set of *G7* commands can be given in the Command box, or input through a command file, as we will show below. The results window shows the results of running commands. Previous commands in the Command box can be retrieved and

[3] All three parts are available at http://www.inforum.umd.edu/papers/TheCraft.html or on the datastick as \Docs\CraftAll2.pdf

[4] The 17-sector versions are in folders with an 'A' suffix. For example, Session1A is essentially the same as Session1, but with 17 sectors instead of 9. The datastick also includes 47 sector versions, with have a 'B' suffix.

reused by clicking the down arrow to the right of the command line, or simply hitting the up or down arrows on the keyboard.

Figure 1.3. G7 Main Window



G7 commands are typed into the command box. When you have finished typing, hit the [Enter] key. Let's try a useful command, which allows us to look at a dictionary of the database. In the command box, type

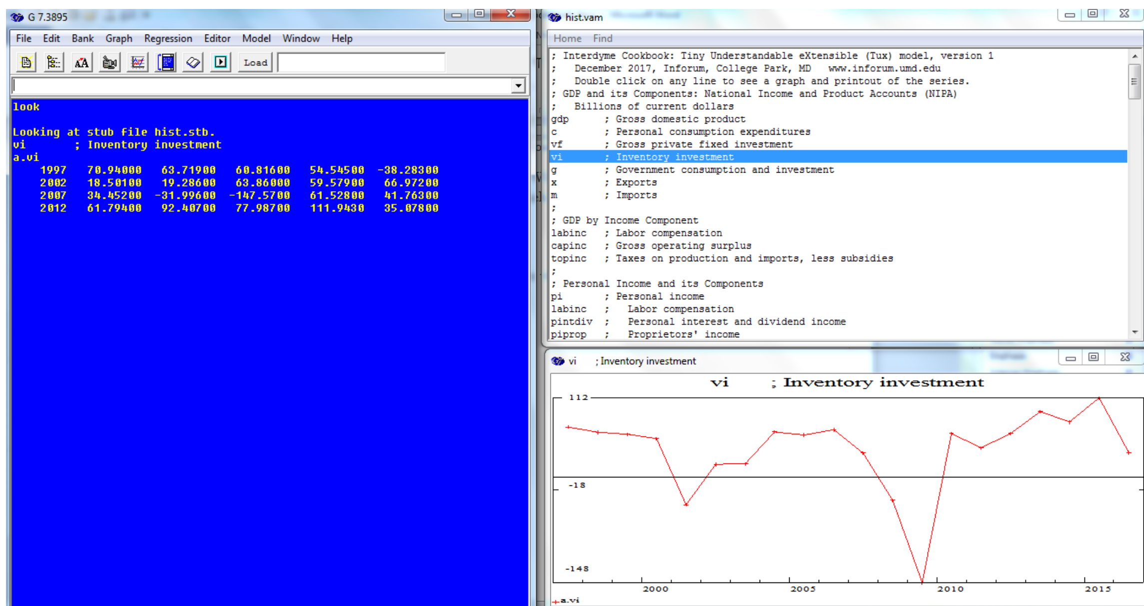
```
look a
```

and hit [Enter]. A window with a white background will appear to the right of the main window, as shown in Figure 1.4. This is called the "look window". If you double-click on any line in that window, a graph of that variable will appear. If you don't have the layout shown in Figure 1.4, it is best to move and resize your look and graph windows so that you do. These positions will be remembered, and help prevent windows from becoming hidden.

Try clicking on *gdp*, *vf*, *x*, *m*, and whatever other variables you would like to view. The first section of the look window shows macroeconomic variables such as these. These are scalar variables (not vectors or matrices). The *Tux* model databank also has several vectors and matrices. Some of these are shown in the second part of the look window. For example, if you double-click on the line for *outz1*, you will see nominal commodity output for Agriculture, forestry & fisheries, which is sector 1 in the *Tux* model 9-sector aggregation[5].

[5] Note that in this database, macrovariables from the national accounts are in billions of dollars. Data from the input-output accounts are in millions of dollars.

Figure 1.4. G7 Look Window with Graph



Close the look window and graph window now by clicking the 'X' in the top right hand corner of each.

Another way to view matrix and vector data is with the "(s)how" command[6]. To show the vector of exports over time, type:

```
show exz
```

Figure 1.5. Viewing a Vector Using the "show" Command

File Edit Options

	exz	1997	1998	1999	2000	2001	2002
1	AgForestFish	25311.00	21045.00	19248.00	20840.00	20693.00	21186.00
2	MiningExtrac	6814.00	6654.00	5377.00	5379.00	4926.00	5045.00
3	ElectGasWatr	1884.00	1517.00	1387.00	1482.00	2268.00	1406.00
4	Construction	154.00	171.00	81.00	82.00	85.00	86.00
5	Mfg	517391.00	512668.00	521522.00	576318.00	529975.00	499882.00
6	Trade	62545.00	62321.00	65280.00	73236.00	69635.00	70041.00
7	Transport	58560.00	55264.00	56667.00	60482.00	55985.00	55251.00
8	Services	218085.00	225247.00	244255.00	260841.00	247343.00	254447.00
9	Government	296.00	302.00	250.00	265.00	279.00	261.00

There are three alternative ways to show a matrix. To show the entire matrix for one year, use the 'y' option:

```
s amfz y 2010   # Show the nominal interindustry flows matrix
```

[6] The letter(s) inside the parentheses is the abbreviated version of the command.

Figure 1.6. Viewing a Matrix Using the "show" Command

	amfz	1	2	3	4	5	6
	2010	AgForestFish	MiningExtrac	ElectGasWatr	Construction	Mfg	Trade
1	AgForestFish	80258.41	119.39	7.42	1787.54	207357.36	5129.82
2	MiningExtrac	2254.63	40900.88	64601.75	11005.60	444426.31	1623.89
3	ElectGasWatr	4656.72	4011.86	3678.72	3880.37	54575.30	17988.36
4	Construction	2134.92	4810.67	18514.89	1060.22	11812.05	3351.07
5	Mfg	68357.11	30244.55	31231.45	239190.95	1542583.50	106520.45
6	Trade	20254.33	5174.66	6385.39	104751.99	234752.67	59450.48
7	Transport	10245.71	7208.26	18534.73	16407.97	116058.88	96818.66
8	Services	23652.94	37037.92	70607.78	87338.45	492831.41	550444.56
9	Government	65.11	8.73	1118.98	201.63	5365.89	14622.94

To show a column of the matrix over time, use the 'c' option.

```
s amfz c 3   # Show column 3
```

To show a row of the matrix over time, use the 'r' option.

```
s amfz r 5   # Show row 5
```

There are two other commands that are commonly used to view data, the "(ty)pe" command and the "(gr)aph" command. To type the values of gdp, use:

```
ty gdp
```

To graph the values of imports of sector 5, use:

```
gr imz5
```

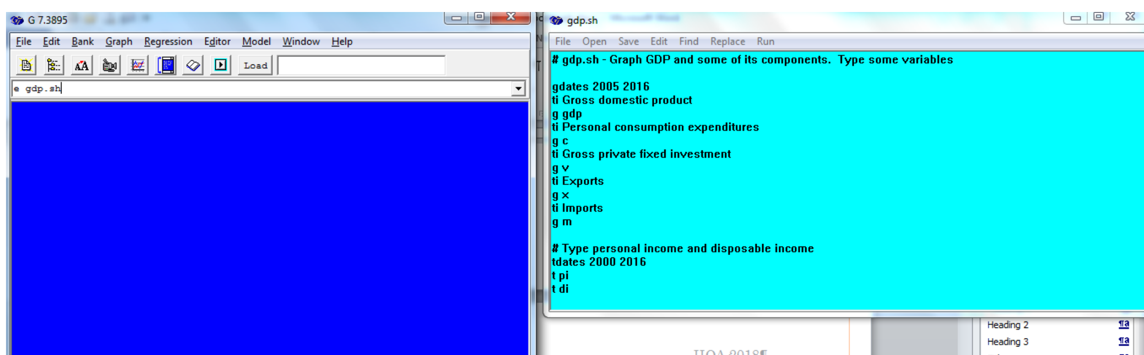
Although most commands can be given by using the command box, it is more common to type a sequence of commands into a file, called an *add file*. *G7* has its own dedicated editor that can be used to execute an entire file, or portions of it.

The command to edit a file is simply "(e)dit". Type the following in the command box:

```
e gdp.sh
```

and a blank editor window will appear to the right of the main window.

Figure 1.7. Using the *G7* Editor



Type the following text into the editor window:

gdp.sh – A "show" file

```
# gdp.sh - Graph GDP and some of its components.  Type some
#  variables

gdates 2005 2016
ti Gross domestic product
gr gdp
ti Personal consumption expenditures
gr c
ti Gross private fixed investment
gr vfix
ti Exports
gr x
ti Imports
gr m

# Type personal income and disposable income
tdates 2000 2016
ty pi
ty pidis
```

Click the "Save" command in the editor menu to save the file. To execute the file, click the "Run" command in the editor menu. As each graph displays, the add file will pause, and show the following window:

Figure 1.8. Continue Add File Window

You can always click the "Stop" button, or type the [Esc] key to stop the add file. Click "Cont." to continue. The "Max" button will make the graph full screen. "Normal" brings it back to the size it was. "Save" allow you to save it to a picture file which can later be imported into other software, such as Word or Powerpoint. If a printer is available, "Print" will make a printout of the graph. "Shift1" and "Shift2" leave the current graph on the screen, but in a smaller format, so that it can be compared with the next graph that appears. The "Zip" command instructs *G7* not to print any graphs. If you click this by mistake, you can bring graphing back by giving the command:

```
zip off
```

Keep on clicking "Cont." to view the graphs you have set up in the add file. Hitting the space bar or the [Enter] key will repeat the last choice you made.

In this gdp.sh file, there are some new commands: Anything following a '#' on a line is treated as a comment, and not processed by *G7*. It is always a good idea to start an add file with a comment that includes the file name and an explanation of what the file does. The "(ti)tle" command provides a title for the graph. There is also a "(subti)tle" command for giving subtitles, and the "(vaxti)tle" command for a vertical axis title. The "gdates" command specifies the dates for graphing. The "tdates" command gives the dates for the "(t)ype"command. These dates can also be specified

along with a "graph" or "type" command, in which case they remain in effect for subsequent commands.

Another way to run the *gdp.sh* file is to use the "add" command. The format is shown below:

```
add gdp.sh
```

The name for a file of commands for *G7* is an *add file*. This is similar to the idea of a *do file* in *Stata*. Note that add files can use the "add" command to call other add files. The add command can also take arguments, or be run in a loop.

1.2 Calculations in *G7*

The most commonly used command for calculating variables in *G7* is the "f" command. The following statements calculate unemployment (*unemp*) and the unemployment rate (*un*), given total household employment (*emp*) and the labor force (*lfc*).

```
f unemp = lfc - emp
f un = unemp/lfc * 100
```

The following command calculates the personal savings rate:

```
f savrat = pisav/pidis * 100
```

Elements of a vector can be summed using the @csum() function. All functions in *G7* start with the '@' symbol. The following command sums all elements of nominal final demand in millions and then converts to billions. We then check this against the national accounts GDP:

```
f gdpchk = @csum(fdz,1-9)/1000   # millions to billions
ti Compare Total Final Demand and GDP
gr gdpchk gdp 1997 2016
```

The expression "1-9" is an example of a *group*, a concept that comes up repeatedly in *G7* and *Interdyme*. The above group indicates all 9 sectors from 1 to 9 inclusive. Another example is "1,3,5,7,9", which would be all odd sectors, or "1-4,6-9", meaning all sectors except for 5. Another way to write that group is "1-9(5)", as element in parenthesis are excluded or removed from the group. In the U.S. data, the following identities hold:

```
gdp = @csum(fdz,1-9)/1000
c = @csum(pceioz,1-9)/1000
vfix = @csum(gpfiz,1-9)/1000
vi = @csum(venz,1-9)/1000
g = @csum(govz,1-9)/1000
```

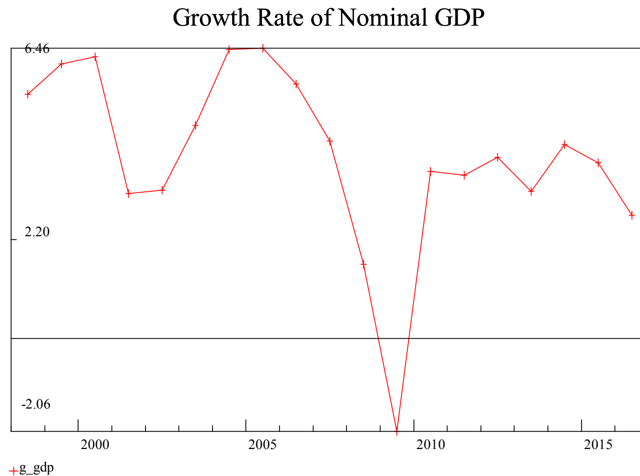
The NIPA measures of exports and imports are not the sum of the corresponding IO vectors. However, the two measures of *net exports* are equal:

```
netex = x - m
netex = @csum(exz,1-9)/1000 - @csum(imz,1-9)/1000
```

The logarithmic function is @log(), and the exponential function is @exp(). The following commands calculate the growth rate of nominal *gdp* and graph it.

```
f lgdp = @log(gdp)
f g_gdp = (lgdp - lgdp[1])*100.
ti Growth Rate of Nominal GDP
gr g_gdp 1998 2016
```

Figure 1.9. Growth Rate of Nominal GDP



1.3 Databanks in *G7*

One of the powerful features of *G7* is its ability to access several databanks in a flexible and convenient way. There are two main types of databanks. The first is called simply a "G bank". It consists of two files, one with the extension ".bnk", which holds the data, and the other with the extension ".ind", which is a list of series names and their locations in the bank. The historical bank for the *Tux* model contains data for the macrovariables in the model. This bank is named *hist.bnk*. The command for "assigning" a bank, which makes the data available to *G7* is "(b)ank". An example of this command is:

```
bank hist
```

(Notice that the .bnk extension should not be given.) Each bank can also be assigned a letter, which can be used to access series in that bank. For example, there is another bank in the directory called *macro.bnk*. Let's assign that bank to position 'b', using the "(ba)nk" command:

```
bank macro b
```

The command "listbanks" or "lb" is used to get a list of the currently assigned banks, and to show their positions, or bank letters. If you type

```
lb
```

You'll now see the following output:

```
There currently are 2 assigned banks.
Bank a: hist.bnk                   Type: bnk
Bank b: macro.bnk                  Type: bnk
```

Another bank is always present, called the "workspace bank". By default, its name is *ws.bnk*. It is always assigned at position 'w'. Therefore there are 25 possible positions for an assigned bank: the letters 'a' through 'z', excluding 'w'. The

workspace bank is the destination of any new series that are created, for example using the "f" command.

To see the list of the series in a databank, use the "(lis)tnames" command. To see the list of series currently in the workspace bank, use:

```
lis w
```

The other type of bank is called a "Vam file" and is used to store vectors and matrices. The structure of a Vam file is fixed, and determined by a configuration file named *vam.cfg*. Here is an extract from the *vam.cfg* file for the first version of the *Tux* model:

vam.cfg

```
# vam.cfg for the Tux9 model, a very simple model based on an aggregated
#   9-sector IO framework.  This is for model 1, with only nominal variables.
# Years of the vam file.
1997 2030
# Name  |Number of  |Files of titles of|   Description
#       |row col lag|  rows   cols     |
amfz       9   9  0  sec9.ttl sec9.ttl # Input-output flow matrix
amz        9   9  0  sec9.ttl sec9.ttl # Input-output coefficient matrix
linv       9   9  0  sec9.ttl sec9.ttl # Leontief inverse
# Nominal vectors
outz       9   1  3  sec9.ttl # Output
pceioz     9   1  0  sec9.ttl # Personal consumption expenditure
gpfiz      9   1  0  sec9.ttl # Gross Private Fixed Investment
venz       9   1  0  sec9.ttl # Inventory change
govz       9   1  0  sec9.ttl # Government spending
exz        9   1  0  sec9.ttl # Exports
imz        9   1  0  sec9.ttl # Imports
fdz        9   1  0  sec9.ttl # Total final demand
ddz        9   1  0  sec9.ttl # Domestic demand = outz+imz-exz
impshrz    9   1  0  sec9.ttl # Imports share = imz/ddz
interz     9   1  0  sec9.ttl # Total row intermediate, cur$
intcolz    9   1  0  sec9.ttl # Total column intermediate, cur$
```

All text after a '#' on a line are comments. The first line which is not a comment indicates that this Vam file will contain data from 1997 to 2030. The rest of the file (that is not comments) has one line for each matrix or vector. The format of these lines is described briefly in the comment lines 5-6. For example, the intermediate flow matrix is called *amfz*. The line describing its structure is:

```
amfz       9   9  0  sec9.ttl sec9.ttl # Input-output flow matrix
```

The fact that it is a matrix is indicated by both row and column dimensions being greater than 1. It has 9 rows and 9 columns. The fourth entry, which is zero, we'll skip for now. The next two items are filenames, indicating the "titles files" for the rows and columns. The rest of the line is a comment. For the vector *outz* the line is as follows:

```
outz       9   1  3  sec9.ttl # Output
```

To assign a Vam file, use the “(v)am” command. The historical data for vectors and matrices in the *Tux* model is named “hist.vam”. To assign this bank to position ‘d’, use the command:

```
vam hist d
```

You could then type the values of exports for sector 1 using the command:

```
ty d.exz1
```

You could use the “show” command to show values of labor compensation with:

```
s d.lab
```

1.4 The *Tux* Model Database

The *Tux* model of the U.S. is a small 9-sector model, but based on a much larger database of a U.S. model called *Lift*, which has 121 commodities and 71 industries. We have used *G7*’s aggregation tools to create this teaching databank, but the skills you learn in this course can be applied to much larger models.

Table 1.1 shows a list of the sectors in *Tux,* and their correspondence with the North American Industry Classification System (NAICS).

Table 1.1 – *Tux* Sectoring Classification

#	NAICS	Description
1	11	Agriculture, forestry, fishing and hunting
2	21	Mining and extraction
3	22	Utilities
4	23	Construction
5	31-33	Manufacturing
6	42,44-45	Trade
7	48-49	Transportation
8	51,52-56,61-62,71-72,81	Services
9	92	Government

The first version of *Tux* we present in this session has data in current prices only. Our naming convention is for matrix and vector variables that may be present either in current or constant prices, for the current price versions to end in a ‘z’. The database has the IO table, final demands, output, value added, and employment variables. The list of matrix and vector variables is shown in Table 1.2. Total final demand, the vector *fdz* is defined by:

fdz = pceioz + gpfiz + venz + govz + exz – imz

The row sum of the intermediate matrix *amfz* is *interz*. Nominal output satisfies the identity:

outz = interz + fdz

Total value added *va* is defined by:

va = lab + gos + topi

The column sum of the intermediate matrix is *intcolz*. Nominal output also satisfies the identity[7]:

outz = intcolz + va

Table 1.2 – Matrices and Vectors in *Tux*, Version 1

Name	Description
A-Matrix	
amfz	A-matrix in flows, Current dollars
amz	A-matrix coefficients
linv	Leontief inverse
Output, Final Demands and Related Vectors	
outz	Output
pceioz	Personal consumption expenditure
govz	Government spending
gpfiz	Gross Private Fixed Investment
venz	Inventory change
exz	Exports
imz	Imports
fdz	Total final demand
ddz	Domestic demand = outz+imz-exz
impshrz	Imports share = imz/ddz
interz	Total row intermediate, cur$
intcolz	Total column intermediate, cur$
Value Added	
lab	Labor compensation
gos	Gross operating surplus
topi	Taxes on production and imports
va	Value added
Employment	
empv	Employment
hrsv	Hours worked
prdv	Productivity

Domestic demand *ddz*, which is used to calculate imports in the model, is defined as:

ddz = outz + imz – exz

The import share, or share of imports of domestic demand, is defined simply as:

impshrz = imz/ddz

The direct coefficients matrix is *amz*. The *Leontief inverse*[8], defined as $(I - amz)^{-1}$ is in the matrix named *linv*. Employment by sector, in thousands of persons, is in the vector *empv*. Hours worked, in millions, is in the vector *hrsv*. Productivity, defined as *outz/hrsv* is in the vector *prdv*. (Technically, productivity should be calculated using real output, but we will overlook that for now.)

Using the *G7* "(s)how", "(ty)pe" or "(gr)aph"commands you learned in the previous section, feel free to examine any of the vectors or matrices in *Tux*.

In addition to matrices and vectors, a typical *Interdyme* model also has *macrovariables*. These are scalar variables, such as an interest rate, aggregate GDP,

[7] There is no distinction between industry and commodity in this Tux model. All IO data are commodity based.
[8] The Leontief Inverse will be tasted more fully in Session 3.

or the government deficit. Table 1.3 shows a list of the variables available in the *Tux* historical databank. Not all of them are currently forecast in the model. However, any of them can be displayed for the historical period, using the "look" command, or the "(t)ype" or "(g)raph" commands. Note that the variables *gdp* to *topinc* are sums of the corresponding vectors, although they are in billions of dollars, whereas the vectors are in millions.

Table 1.3 – Macrovariables in *Tux*, Version 1

Name	Description
GDP and its Components	
gdp	Gross domestic product
c	Personal consumption expenditures
vfix	Gross private fixed investment
vi	Inventory investment
g	Government consumption and investment
x	Exports
m	Imports
GDP by Income Components	
labinc	Labor compensation
capinc	Gross operating surplus
topinc	Taxes on production and imports, less subsidies
Personal Income and its Components	
pi	Personal income
labinc	Labor compensation
pintdiv	Personal interest and dividend income
piprop	Proprietors' income
piren	Rental income
pigsb	Government social benefits
pibtp	Business transfer payments to persons
picsi	Less: Social insurance contributions
piptax	Less: Personal taxes
pidis	= Disposable income
	Less:
pisav	Personal savings
piipcb	Personal interest payments
piptt	Personal current transfer payments
c	= Personal consumption expenditures
Population, Labor Force and Employment	
pop	Population
lfc	Labor force
emp	Household employment
empind	Total industry employment
unemp	Unemployment
un	Unemployment rate
Financial	
m2	M2 - Money supply
rtb	3-month Treasury bill rate
rtb10y	10-year Treasury bill rate
raaa	AAA bond rate
rcmor	Mortgage interest rate

1.5 What Is The *Tux* Model?

Tux is an *Interdyme* model. This means that it is written in C++, and the code is compiled to make an executable program, named dyme.exe. The program works with two simulation files as it runs. One is a *Vam File* (with extension .vam), which holds the vectors and matrices. The other is called a *G Bank* (with extension .bnk) and holds the macrovariables. We will be running a base case in the next section. When it is done, we will have two files BASE.VAM and BASE.BNK that hold the simulation results.

Table 1.4 – General Sequence of the Model

Declare and initialize vectors, matrices and macrovariables
Loop through the years of the simulation
Load vectors and matrices for the current year
Initialize convergence variables: consumption and investment
Forecast final demand vectors
Calculation of the IO Solution
Forecast value added vectors
Calculate macro identities
Check for convergence
Store vectors and matrices for the current year
Finish annual loop
Store macrovariables

Table 1.4 shows the general sequence of the model solution. First there is an initialization phase, where vectors, matrices and macrovariables are declared and initialized. The main part of the model is a loop through the years of the simulation. In the model code, the current year is always represented by the variable *t*. Within each year, vectors and matrices which have been declared are loaded from the Vam file. Next a loop is set up for convergence. In *Tux* we check convergence on the values of consumption and investment. Within this convergence loop, the first section of the model calculates values for the final demand vectors. Next, total final demand is given to the IO solution to calculate output. After this, the value added vectors are calculated. Next comes a section that calculates macroeconomic identities. The model checks for convergence. If it hasn't converged, it goes back to the top of the loop to calculate final demands again. If it has converged, it stores the matrices and vectors for the current year, and then moves on to solving the next year. When the annual loop has finished, the model writes out the macrovariables to the *G Bank* and finishes.

In this first simple version of *Tux*, there are no individual equations for the final demand and value added vectors. Rather, we specify the totals exogenously, and share to the industries using share vectors. This type of model is called a *top down* model.

Personal income (*pi*) is calculated within the macroeconomic identities. Disposable income (*pidis*) is then formed by removing personal taxes (*piptax*). Finally, savings (*pisav*) and a few other items are removed from disposable income to obtain total consumption (*c*).

We will look at the computer code that implements these steps in Session 3.

1.6 Running the *Tux* Model

You don't spring into good cooking naked. You have to have some training. You have to learn how to eat.

Good model building presupposes understanding what you want from a model. What problems should it solve? What questions should it answer? What good advice can be learned from it? Although our current version of *Tux* is extremely simple, it helps to run it and look at the forecasts to understand something of the operation of the model. In this section we'll make a base run of the model, and in the next section run an alternative scenario. Both scenarios can be loaded at the same time in *G7* and variables from each scenario can be compared graphically.

The *Tux* model can be run from within *G7*. Choose the menu item Model | Run Dyme Model, or simply hit the F8 function key. The dialog box, or "Interdyme Run Form" shown in Figure 1.11 appears.

Figure 1.10

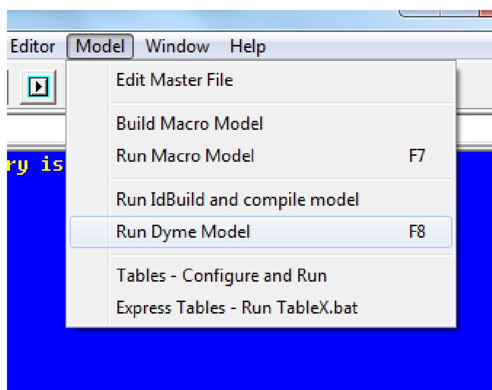


Figure 1.11

IdRun

Interdyme Run Form

Title of Run: Tux Version 1 - Base Case

Start Date: 2016 End date: 2025

Macro equation start date: 2016 Discrepancy year: 2016

base — RESULT -- root name of the banks (G and Vam) which are the result of this run.

hist — START -- root name of banks (G and Vam) which are to be copied as the starting values for this run.

base — EXOG -- root name of the .xog file to change the exogenous data in the copied bank and vam files.

base — MACFIXES -- root name of the .mfx file of input to MacFixer or "none" if none.

base — VECFIXES -- root name of the .vfx file of input to VecFixer or "none" if none.

Use all data

100 Max Loop Iterations

2100 Debug Start Year

Type of Run: Deterministic / Optimizing / Stochastic

50 Iterations / Additive errors / Coefficient errors

none — Optimization specification file

Cancel OK

Fill it out as shown in the figure. Here are the fields, and their meanings:

Field Name	Meaning	File Name
Title of run	Descriptive title, which will appear in the databank and in tables.	
Start date	Start of the simulation, usually the last year of data	
End date	The ending year of the simulation	
Macro equation start date	When the macro equations start working, usually the last year of data.	
Discrepancy year	Year for calculating IO discrepancy, also usually the last year of IO data	
Result	Name of forecast bank (.VAM, .BNK)	BASE.VAM, BASE.BNK
Start	Starting bank, usually HIST.VAM, HIST.BNK	HIST.VAM, HIST.BNK
Exog	Name for exogenous variables file	BASE.XG
Macfixes	Name for macro fixes file	BASE.MFX
Vecfixes	Name for vector fixes file	BASE.VFX

The last two items refer to "Fixes". A fix is a way to specify values for exogenous variables, or to modify calculated values of endogenous variables in the model. *Endogenous variables* are variables "born within" the model. In other words, there are regression equations or identities which calculate these variables. *Exogenous variables*, on the other hand, are "born outside" the model, and must be made up or projected by the person running the model.

For this first run, please fill out the fields as shown, and then click the OK button. When the first message box appears, click OK. Then several operations will run in the background. If you get to a blank screen, just hit the [Enter] key. You should then see the model run in the *G7* main window.

Figure 1.12 – First Model Run in *G7*

```
C:\windows\system32\cmd.exe
Seidel iterations: 5 Iter  5 pce = 14334.8 pcedif =  13.32 invdif =   0.00
Seidel iterations: 5 Iter  6 pce = 14343.1 pcedif =   8.33 invdif =   0.00
Seidel iterations: 4 Iter  7 pce = 14348.3 pcedif =   5.21 invdif =   0.00
Seidel iterations: 4 Iter  8 pce = 14351.6 pcedif =   3.26 invdif =   0.00
Seidel iterations: 4 Iter  9 pce = 14353.6 pcedif =   2.04 invdif =   0.00
Seidel iterations: 4 Iter 10 pce = 14354.9 pcedif =   1.28 invdif =   0.00
Seidel iterations: 4 Iter 11 pce = 14355.7 pcedif =   0.80 invdif =   0.00
Seidel iterations: 3 Iter 12 pce = 14356.2 pcedif =   0.50 invdif =   0.00
2024 Seidel iterations: 8 Iter  1 pce = 14444.5 pcedif =  88.33 invdif =  74.29
Seidel iterations: 5 Iter  2 pce = 14499.8 pcedif =  55.26 invdif =   0.00
Seidel iterations: 5 Iter  3 pce = 14534.4 pcedif =  34.58 invdif =   0.00
Seidel iterations: 5 Iter  4 pce = 14556.0 pcedif =  21.63 invdif =   0.00
Seidel iterations: 5 Iter  5 pce = 14569.5 pcedif =  13.53 invdif =   0.00
Seidel iterations: 5 Iter  6 pce = 14578.0 pcedif =   8.47 invdif =   0.00
Seidel iterations: 4 Iter  7 pce = 14583.3 pcedif =   5.30 invdif =   0.00
Seidel iterations: 4 Iter  8 pce = 14586.6 pcedif =   3.31 invdif =   0.00
Seidel iterations: 4 Iter  9 pce = 14588.7 pcedif =   2.07 invdif =   0.00
Seidel iterations: 4 Iter 10 pce = 14590.0 pcedif =   1.29 invdif =   0.00
Seidel iterations: 4 Iter 11 pce = 14590.8 pcedif =   0.81 invdif =   0.00
Seidel iterations: 3 Iter 12 pce = 14591.3 pcedif =   0.51 invdif =   0.00
Seidel iterations: 3 Iter 13 pce = 14591.6 pcedif =   0.32 invdif =   0.00
2025 Seidel iterations: 8 Iter  1 pce = 14681.2 pcedif =  89.61 invdif =  75.87
Seidel iterations: 5 Iter  2 pce = 14737.3 pcedif =  56.06 invdif =   0.00
Seidel iterations: 5 Iter  3 pce = 14772.4 pcedif =  35.08 invdif =   0.00
Seidel iterations: 5 Iter  4 pce = 14794.3 pcedif =  21.95 invdif =   0.00
Seidel iterations: 5 Iter  5 pce = 14808.1 pcedif =  13.73 invdif =   0.00
Seidel iterations: 5 Iter  6 pce = 14816.6 pcedif =   8.59 invdif =   0.00
Seidel iterations: 4 Iter  7 pce = 14822.0 pcedif =   5.37 invdif =   0.00
Seidel iterations: 4 Iter  8 pce = 14825.4 pcedif =   3.36 invdif =   0.00
Seidel iterations: 4 Iter  9 pce = 14827.5 pcedif =   2.11 invdif =   0.00
Seidel iterations: 4 Iter 10 pce = 14828.8 pcedif =   1.32 invdif =   0.00
Seidel iterations: 4 Iter 11 pce = 14829.6 pcedif =   0.83 invdif =   0.00
Seidel iterations: 3 Iter 12 pce = 14830.1 pcedif =   0.51 invdif =   0.00
Seidel iterations: 3 Iter 13 pce = 14830.5 pcedif =   0.32 invdif =   0.00
```

Hit [Enter] one more time to return control to *G7*. (Please don't forget!)

Now you have run the base scenario. A new vam file BASE.VAM and a new G bank, BASE.BNK have been created, that include the same historical data as in HIST.VAM and HIST.BNK, but also contain the forecasted values of the vectors, matrices and macrovariables calculated by the model.

From *G7*, give the command

```
vam base
```

to load the just completed base scenario. (The "vam" command treats the BASE.VAM and BASE.BNK as a single unit, so both are loaded.)

Now, we'll create an "add file" in the *G7* editor. From the *G7* Command Box, enter the following command:

```
ed base.sh
```

and you should see an empty editor window. Now, type the commands into the base.sh file as shown in Figure 1.13.

Figure 1.13 Editing the base.sh File

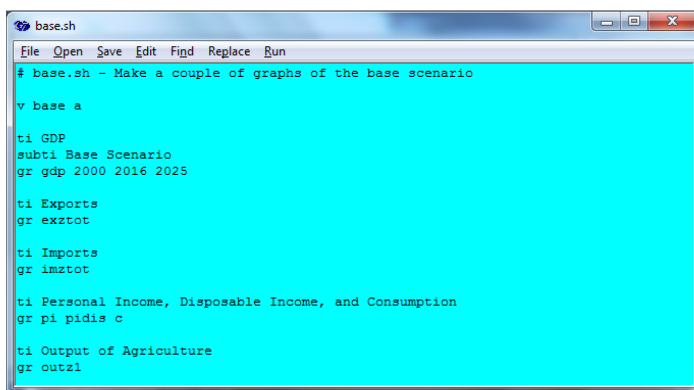
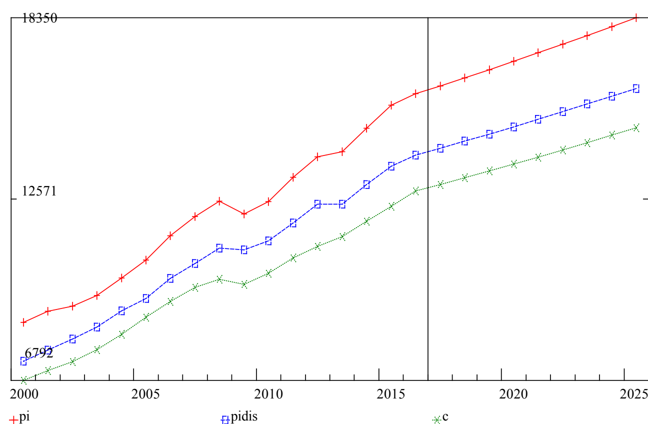


Figure 1.14 Graph of Personal Income, Etc.



When you are finished, click the Save menu item to save the file. To run all the commands in the file, you can click the Run menu, or hit the F9 function key. Figure 1.14 shows one of the graphs from this base.sh file. The vertical line in the graph shows the end of the historical data and the beginning of the forecast.

1.7 Making an Alternate Scenario

The base scenario we have set up is only one of many possible scenarios. Each scenario can be thought of as a combination of:

1. The current version of the model, identities and equations.
2. The starting year, and the values of the data in that year.
3. Assumptions on endogenous or exogenous variables.

We'll be making changes to the model in the next session. For these sessions, we will always use 2016 as the last year of historical data, and as the starting year for running scenarios. However, we will change an assumption, and then see how this affects the model results.

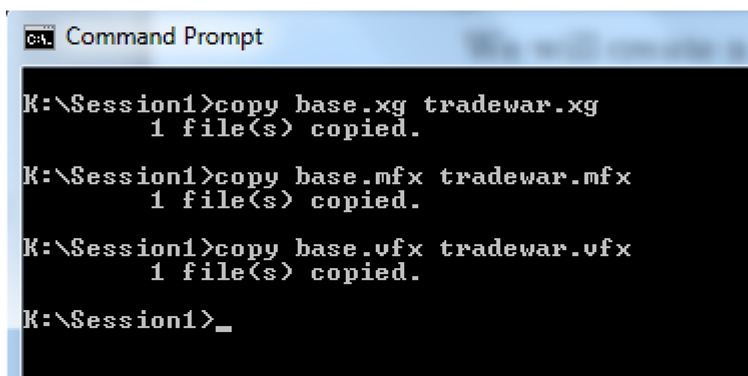
There are three files, briefly mentioned above, that are used to specify assumptions to the model:

1. BASE.XG – A file of *G7* commands, that can be used to specify projected values for exogenous variables
2. BASE.MFX – The file for typing in *macro fixes*.
3. BASE.VFX – The file for typing in *vector and matrix fixes*.

We will create a scenario that assumes that the U.S. has lower imports and lower exports in the forecast, due to withdrawal from trade agreements and the imposition of punitive tariffs. We will call the resulting scenario TRADEWAR.

First, we will copy the above three files to files with the root name TRADEWAR. If you are more comfortable with the command prompt, here are the commands:

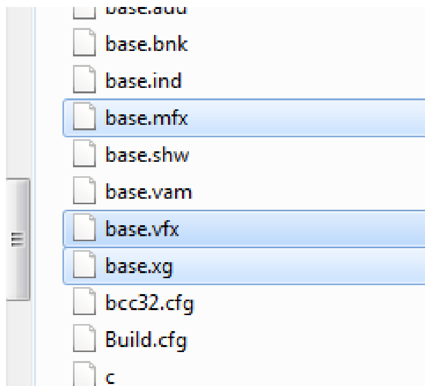
Figure 1.15 Copying in Command Prompt



If you are more comfortable with Windows Explorer, you can copy them there. Simply hold down the 'Ctrl' key and click on base.mfx, base.vfx and base.xg. Then right-click, and pick 'Copy'. Right-click in any white area of *Explorer*, and pick 'Paste'. To rename the copies, click twice on each name very slowly. This puts a box

around the filename, where you can type in a new name. Name the copied files tradewar.mfx, tradewar.vfx and tradewar.xg.

Figure 1.16 Copying in Explorer



Several text editors have been provided on the data stick, and placed in the \Util folder, which is on the path. These are Notepad++ (np), WordPad (wp) and the FTE editor (ed). They can each be accessed from the Command Prompt, with the short name in parenthesis. Let's start Notepad++ to open the TRADEWAR.MFX file that we have just copied from BASE.MFX. Type the command:

```
np tradewar.mfx
```

You will see the following contents:

TRADEWAR.MFX

```
# Interdyme Cookbook, 2018
# Macfixes.mfx - Macrovariable fixes for the Tux model, version 1

rho vfix 0.6 2016
rho c     1   2016
rho imztot 1 2016
rho exztot 1 2016
rho g      1 2016
rho vi     1 2016
# Fixed investment
gro vfix
    2017 2.1
    2030 2.1
# Government
gro g
    2017  1.3
    2030  1.3
# Imports
gro imztot
    2017   2.1
    2030   2.1
# Exports
gro exztot
    2017   2.1
    2030   2.1
```

Lines in the macro fixes file that begin with a '#' are comments, and are ignored by the program. Fixes in this file are indicated by the following commands[9]:

- *rho* – Specify a "rho" fix, which sets the last year of data, and a value for the *rho* parameter, which may have come from a regression equation.
- *gro* – Specify a growth rate fix
- *ovr* – Specify actual values of a variable
- *ind* – Move a variable by an index
- *mul* – Multiply a variable by specified values
- *cta* – Add specified values to a variable

The format of a macrofix is:

<fixtype> <variablename>
<year> <value> [<value> <value> <value> …]
<year> <value> [<value> <value> <value> …]

For example, there is already a growth rate fix on total exports in the base case:

```
gro exztot
    2017   1.5
    2030   1.5
```

This specifies growth rates for two years only, and in this case, the growth rates are the same. The *Macfixer* program will interpolate the growth rates for the missing years, which means in this case that growth will be a constant 1.5 percent. The first growth rate is specified for 2017, since 2016 is the last year of data. Let's reduce the growth rate of exports from 2.1 to 0.5, and reduce the growth of imports from 2.1 to 1.8, resulting in the following:

```
# Imports
gro imztot
    2017   1.8
    2030   1.8
# Exports
gro exztot
    2017   0.5
    2030   0.5
```

Now, save the file using File | Save, or Ctrl+S.

Start up *G7* and we'll run the TRADEWAR scenario. Again, choose the menu item Model | Run Dyme Model, or press the F8 function key. Fill in the form as shown in Figure 1.17 below.

[9] There are several other types of fixes, but we'll cover them later.

Figure 1.17 Run Form for "Trade War"

IdRun

Interdyme Run Form

Title of Run: Trade War for the U.S.

Start Date: 2016 End date: 2025

Macro equation start date: 2016 Discrepancy year: 2016

tradewar — RESULT -- root name of the banks (G and Vam) which are the result of this run.

hist — START -- root name of banks (G and Vam) which are to be copied as the starting values for this run.

tradewar — EXOG -- root name of the .xog file to change the exogenous data in the copied bank and vam files

tradewar — MACFIXES -- root name of the .mfx file of input to MacFixer or "none" if none.

tradewar — VECFIXES -- root name of the .vfx file of input to VecFixer or "none" if none.

☑ Use all data

Type of Run: ◉ Deterministic ○ Optimizing ○ Stochastic

100 Max Loop Iterations

2100 Debug Start Year

50 Iterations ☑ Additive errors ☑ Coefficient errors

none Optimization specification file

Cancel OK

Run the model as before. Don't close down G7 quite yet. Let's load the file base.sh into the *G7* editor again ("ed base.sh"):

Figure 1.18 Comp.sh, to Compare the Base and Tradewar Scenarios

J:\Session1\comp.sh

File Open Save Edit Find Replace Run

```
# comp.sh - Make a couple of graphs to compare the base scenario with the "Trade
#   War" scenario.

vam base a
vam tradewar b

ti GDP
subtitle Comparison of Base and Trade War
gr a.gdp b.gdp 2000 2016 2025

ti Exports
gr a.exztot b.exztot

ti Imports
gr a.imztot b.imztot

ti Personal Income
gr a.pi b.pi

ti Consumption
gr a.c b.c

ti Output of Agriculture
gr a.outz1 b.outz1
```

The choose the editor menu File | Save as … and pick comp.sh as the new file name. Then edit the file as shown in Figure 1.18 above.

When you are finished, you can run this file in *G7* by either clicking on the Run menu, or hitting the F9 function key. If you would like to add other variable comparisons to comp.sh, you can refer to tables 1.2 or 1.3 for vector, matrix or macrovariable names.

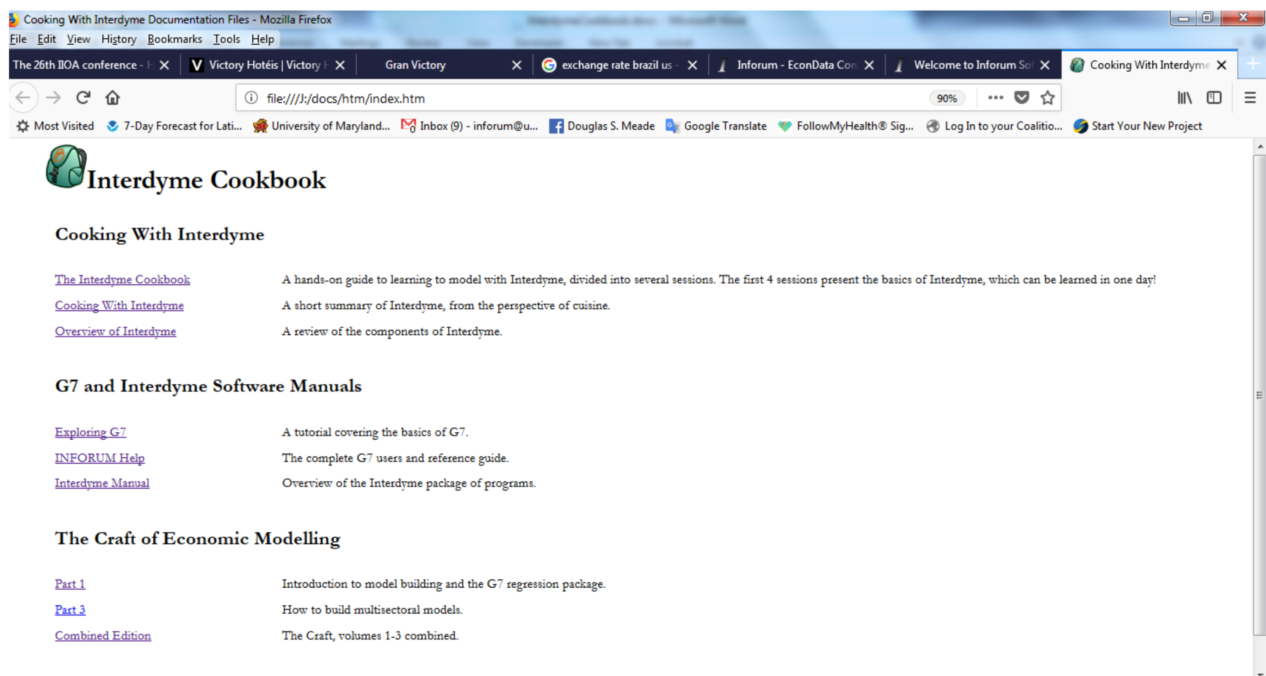
In the next session, we'll explore the macroeconomic identities, and make a few changes to the macroeconomic part of the model.

1.8 Where to Find More Information

The "*G7* Cheat Sheet" at the back of this session summarizes some of the most commonly-used *G7* commands.

From the Command Prompt on the datastick (the "launch" window), type "help" to bring up page shown in Figure 1.19. Click on "The Interdyme Cookbook" to read this document in PDF. "Cooking With Interdyme" is a more general description of what Interdyme modeling is about. "Overview of Interdyme" is slightly more technical. "Exploring G7" provides a quick tutorial of using G7 with one of the U.S. databases that can be downloaded from the Inforum web page. "INFORUM Help" is a large document that contains practically complete documentation on *G7, Compare* and related tools. The "Interdyme Manual" is somewhat outdated, but still contains a good description of building a model with *Interdyme*. Finally, there are various volumes of *The Craft of Economic Modeling*, by Clopper Almon. Volume 1 is an introduction to modeling, and building macroeconomic models using *G7* and *Build*. Volume 2 contains more advanced material on macroeconomic modeling. Finally, Volume 3 is about interindustry macroeconomic modeling, which is also the subject of these sessions.

Figure 1.19 Help Page for the Interdyme Cookbook



All of these materials can be found under the \Docs folder of the datastick. Inforum Help Documentation is in Help.pdf, and is the same as the online help in *G7*. The full 3 volumes of the *Craft of Economic Modeling* is in CraftAll2.pdf.

Many of the most important files that need to be edited in these sessions can be accessed through the *PSPAD* editor which is supplied on the data stick. If you move to the root directory of the data stick by typing:

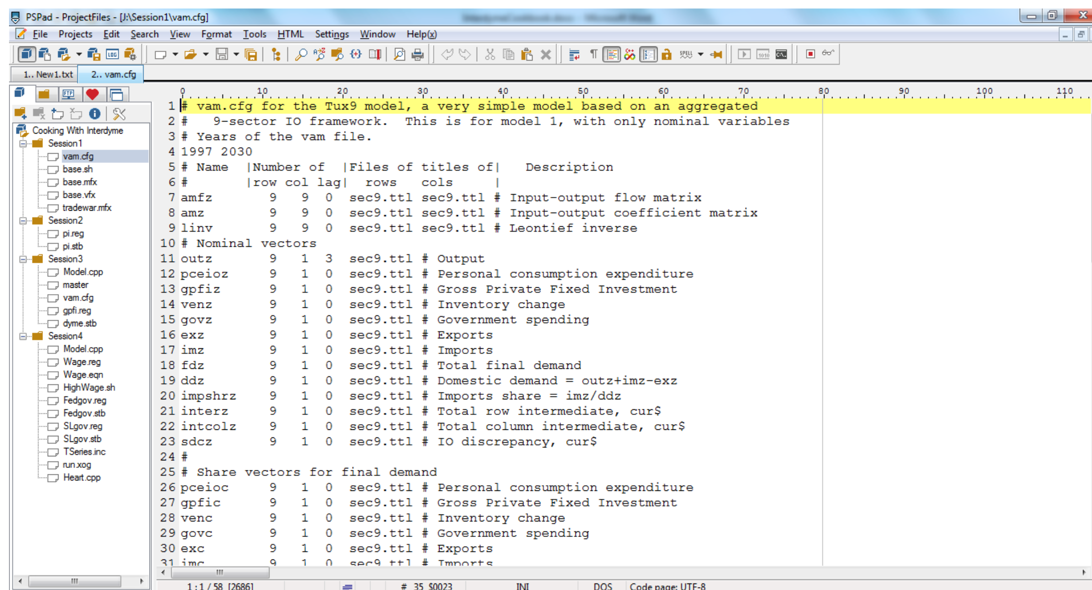
```
cd \
```

and then type

```
edit
```

a PSPAD window will open. It has a tree on the left that contains the most important files in the various sessions. In this way, these files can be accessed immediately. The organization also provides an alternative guide to the sequence of the sessions. A screen shot is shown in figure 1.20.

Figure 1.20 PSPAD Editor Window



G7 Cheat Sheet

Command	Description	Example
(ba)nk	The command will make the standard workspace data bank the assigned bank. If specified with a letter location after, then the bank will be placed in that position, otherwise it is automatically placed in position "a". The syntax is as follows ba <bank_name>[bank_location]	Without location ba gdp With Location ba gdp c To print or type with location ty c.price
(v)am	Makes the specified vam file the assigned bank. The syntax is as follows vam<bank_name>[bank_ location]	Without location vam gdp With Location vam gdp c
fdates	Sets or resets the dates used by subsequent *f* commands. When an *fdate* command is used, it defines the time period in which the subsequent *f* commands act	fdates 1990 2016
gdates	Sets the dates used by the graph or plot command. With 2 dates provided, the series will be graphed from the first date to the second date. With 3 dates, a vertical line is drawn at the 2nd date.	gdates 2000 2016 2025
tdates	Selects "automatic dates" for graph and type commands. The automatic dates are the first and last date of the series actually provided	tdates 2000 2016
(t)ype	The command to display values for any series for the specified dates. The syntax for the command is: ty <series name>[start date][end date] If no dates are assigned, the *tdate* will automatically be used	ty gdp 2005 2016
(g)raph	The command to graph any series. The graph command can graph 1 to 6 series from one date to another. The syntax for the command is: gr <series name>[start date][end date] If no dates are assigned, the *gdates* will automatically be used	For one variable gr gdp 2005 2016 For multiple variables gr gdp gnp 2005 2016
(ti)tle	Provides a title for regressions and graphs	ti Consumption vs GDP
(subti)tle	Provides a subtitle for graphs	subti Consumption vs GDP
(vaxti)tle	Provides the vertical axis title	vaxti money
(e)dit	The command to edit a new or existing file in the editor window	ed makevam.add
look	Brings up the stub file in a scrolling list box, from which you can select various series in the bank that you want to print out and graph.	look a

(s)how	The show command shows vectors and matrices in a grid similar to a spreadsheet. For Vectors: sh <bank letter>.<vector_name><first row><first period> For matrices: sh <bank letter>.<matrix_name><view><first row><first period>	For vectors sh a.vector For Matrix Row View show b.am r 5 For Matrix Column View show b.am c 7 For Matrix Year View show b.matrix y 1997
(a)dd	Execute commands from the named file. The syntax is : add <filename>	add makevam.add
data	Introduces data into the work space. The first number on each line is the date of the first observation on the line. End data with a ; The Syntax is : data<name> <date> <observation1><observation N>	data sales 2010 117 123 134 142 2014 137 143 145;
f	Defines the variable on the left in terms of the variable on the right. It is typically used in order to do calculations. The syntax for the command is: f<variable> = <expression>	f gdpP = gdp/gdpR
(lim)its	Sets limit dates for regressions. lim<start_date><end_date>[forecast_date}	lim 2000 2016
r	Runs regression r<y> = <x>,<x2>,<x2>	r investment = cap,energy,services
gr*	Graphs the predicted values of the regression	gr *
(lis)tnames	Lists all the series in a given bank	lis a
listbanks (lb)	Lists the currently assigned databanks, their type, position letter, and title (if any).	
#	Comment character	
zip <on|off>	The “zip” command prevents printing of graphs and allows an add file to run with no pauses. Use “zip off” to turn off.	
(q)uit	Terminates *G7*. You can also use File | Exit; Alt+F4, or click the red X box in the upper right-hand corner.	

C++ Interlude 1: Basics

A C++ program is built from functions and objects. There are 4 basic data types in C++: int (integer), float (real number), char (single character) and bool (true or false). For the most part, C++ is free-format; anywhere that a blank may appear, any number of blanks or a new line may appear. A statement is a single or combined operation terminated with a semicolon (';'). Every variable used in a C++ function must have been previously declared in that function or declared globally. Every C++ program is a combination of functions, one of which is named main(). With the exception of main(), functions must be declared before they are used. For library or system functions, this will usually be done by an #include statement. Our first example below will illustrate some of these ideas.

apple.cpp

```
#include <stdio.h>
int main(void) {
    int apples;
    apples=2;
    float pie=3.1415926, product;
    product = apples*pie;
    printf(" 2 pi is about %8.5f.\n", product);
}
```

The main() function returns an int, and this version takes no arguments, which is indicated by the word void inside the parentheses. The first variable declared is an integer named 'apples', and it is set to 2 in the following statement. The second is a real number (float) named 'pie', and it is initialized to a value in the same statement where it is declared. The third variable is a float named 'product'. In the next statement, product is set to the product of apples and pie.

The most difficult statement in the program is the call to the printf() library function. The program knows about printf() because it is declared in the system include file stdio.h. The printf() function prints one or more lines of output to the screen. The function takes a formatting string, and zero or more arguments. The formatting string may be just text, or it may include format specifiers which tell it that the value of a variable is to be printed. The format specifiers all start with the '%' character. The '%8.5f' specifier is for a float ('f'), which will occupy 8 columns, with 5 digits after the decimal point. The variable that will be printed is product, which is the other argument to printf().

A program starts with one or more .cpp (text) files. These are *compiled* to *object* (binary) files, with the extension .obj. One or more object files may be *linked* into an *executable* file, with extension .exe.

There is a folder named \CPP on your datastick. You can get there by typing 'c' at the command prompt. You can edit the apple.cpp program in Notepad++ by typing 'np apple.cpp'.

You can compile and link the apple program with 'bc apple'.

Run the program by simply typing 'apple'. Here is the output of the program:

```
J:\CPP>apple
 2 pi is about  6.28319.
```

SESSION 2. MACRO EQUATIONS, MAKING TABLES

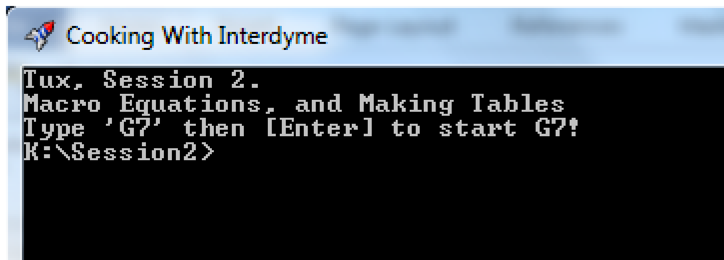
Nothing is too much trouble if it turns out the way it should.

The first session introduced you to *G7* and some of its capabilities, including using the "look" command to navigate a databank, typing and graphing data, viewing matrix and vector data with the "show" command, creating new series with the "f" command and how to handle multiple databanks. The *Tux* model also made its appearance. *Tux* is a basic IO model with 9 sectors, and simple macro accounts. We covered the structure of the model, how to run it, and then how to make alternative assumptions to create a scenario.

In this session, we'll get out our knives and measuring spoons, and learn some of the basics of preparing an interindustry macroeconomic (IM) model, of which *Tux* is an example.

If you have *G7* running, please close it (click on the red 'X' in the top right corner, choose the menu item File | Exit, press the key combination Alt+F4, or type the command "q" in the command box). Go back to the 'Cooking With Interdyme' window you had opened earlier, using the *Launch* program. If you have closed that, you need only navigate back to K: in *Explorer*, and then double-click on Launch (the rocket icon). Type '2' to move to Session 2.

Figure 2.1 Session 2 Folder



The session 2 folder is still working with *Tux*, version 1. The model databanks and base scenario have been copied to \Session2. Looking ahead, session 3 will include an expanded model (*Tux 2*) that will include constant price variables and prices. Session 4 will include *Tux 3* which includes equations for productivity, hours, employment and wages, as well as detail on government receipts and expenditures.

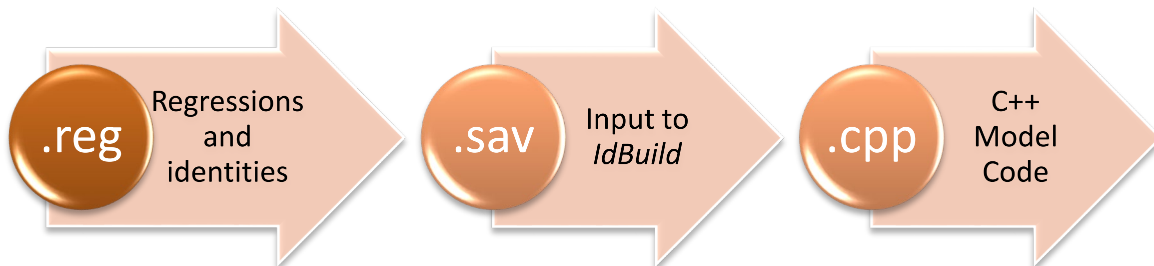
2.1 Macroeconomic Equations and Identities

A macrovariable is a single variable that is neither a matrix or a vector. Macrovariables may be formed as aggregates of parts or entire vectors or matrices, as functions of other macrovariables (identities), or as the left hand side of a regression equation. Figure 2.2 summarizes the model building process with macrovariables.

Macrovariables are kept in a *G Bank*, with file extension .bnk. Vectors and matrices are kept in a *vam file* with extension .vam. In *Interdyme* we have followed the practice of naming the historical files hist.bnk and hist.vam. In these sessions,

hist.vam has already been created for you. However, you'll have the opportunity to build and change hist.bnk using a program called *IdBuild*.

Figure 2.2 The Sequence for Modeling with Macrovariables



Although *IdBuild* is the powerful food processor which spins out C++ code and builds the hist.bnk file, most of the initial ingredients are prepared using *G7*.

The process starts with a ".reg" file, or regression file, which contains regressions, identities and other statements. These files are conventionally named with the extension ".reg", to indicate their purpose. They are processed in *G7* using the "add" command, as we'll see shortly. Part of the .reg file is sandwiched between "save" statements, and from this part a ".sav" file is created. One or more .sav files are then processed by a program called *IdBuild* to create function in C++ code, which can be compiled and linked into a working model. We will look at the identities for calculating personal income in the *Tux* model, to see how this all works. But first, we need a little background on the U.S. national accounting system.

2.2 Background on Personal Income

Unlike most countries in the world, which follow the *System of National Accounts* (SNA), the U.S. has the National Income and Product Accounts (NIPA). In the U.S., the measure of income that provides an estimate of income from all sources, less social insurance contributions, is called Personal income.[10] In the *Tux* model, that variable's name is *pi*. Table 2.1 shows the components of Personal income, and their values for recent years.

The largest component of personal income is Compensation of employees, which consists of wages plus benefits of workers. The next largest is Government social benefits, which consists of Social Security, Medicare, Medicaid and other government transfer payments. Dividend and interest income is also large, and represents income from personal investments. Proprietors' income is income from businesses which are not incorporated. Other components are Rental income and Business transfer payments to persons. After adding up these components, Contributions for social insurance are removed to arrive at Personal income.

[10] Note that titles of NIPA variables usually have the first word capitalized.

Table 2.1 Personal Income and its Components

Title	Variable	2010	2012	2013	2014	2015	2016
Personal income	***pi***	**12,477**	**13,915**	**14,074**	**14,818**	**15,553**	**15,929**
Compensation of employees	*labinc*	7,961	8,610	8,842	9,256	9,708	9,979
Proprietors' income	*piprop*	1,033	1,241	1,285	1,316	1,319	1,342
Rental income	*piren*	403	525	567	612	662	707
Dividend and interest income	*pintdiv*	1,740	2,124	2,056	2,245	2,387	2,378
Government social benefits	*pigsb*	2,282	2,324	2,387	2,499	2,631	2,711
Business transfer payments to persons	*pibtp*	43	43	41	46	53	57
Less: Contributions for social insurance	*picsi*	984	952	1,105	1,155	1,208	1,245
Personal income check		**12,477**	**13,915**	**14,074**	**14,818**	**15,553**	**15,929**

Personal taxes are removed from Personal income to obtain Personal disposable income (*pidis*). Savings (*pisav*) and two other variables are then removed from *pidis* to obtain Personal consumption (*c*).

2.3 Regression Files in *G7*

Regression (.reg) files are designed to be run in *G7* using the "add" command. We'll be working with pi.reg, which is shown below:

pi.reg

```
# pi.reg - Personal income calculations
ba macro    # Source data bank
# -----------------------------------------------------------------
# Define or copy some variables before turning on the save file
# Labor income
f labinc = nice
# Taxes on production and imports
f topinc = nitpils
# Capital income
f capinc = gdp - labinc - topinc

# national income variables
f pi = pi
# Interest and dividend income
f piint = piint
f pidiv = pidiv
f pintdiv = piint+pidiv
# Proprietors' income
f piprop = piprop
# Rental income
f piren = piren
# Government social benefits
f pigsb = pigsb
# Business transfer payments to persons
f pibtp = pibtp
# Contributions for social insurance
f picsi = picsi

f pichk = labinc + pintdiv + piprop + piren + pigsb + pibtp - picsi
ti Personal income: Check identity
subti Billions of $
gr pi pichk 2000 2016
# -----------------------------------------------------------------

save pi.sav
```

```
# Personal income calculation in Tux
# Personal interest and dividends
fex pintdivrat = pintdiv/capinc
f pintdiv = pintdivrat*capinc
# Proprietors' income
fex piproprat = piprop/capinc
f piprop = piproprat* capinc
# Rental income
fex pirenrat = piren/capinc
f piren = pirenrat* capinc
# Business transfer payments
fex pibtprat = pibtp/capinc
f pibtp = pibtprat*capinc
# Government social benefits
fex pigsbrat = pigsb/g
f pigsb = pigsbrat*g
# Contributions to social insurance
fex picsirat = picsi/pi[1]
f picsi = picsirat*pi[1]

# Personal income
id pi = labinc + pintdiv + piprop + piren + pigsb + pibtp - picsi
save off
```

From the command prompt in k:\session2, start up *G7* again by typing 'G7' and then hitting the [Enter] key.

Open up pi.reg in the *G7* editor, by typing the command

```
ed pi.reg
```

in the white command box. Once you have it loaded, go ahead and run it by either clicking the 'Run' menu, or pressing the F9 function key. In the middle of the process, it will show a graph of *pi* and *pichk*, just to make sure that this identity is correct (it is). The file executes quickly, and the file pi.sav has been created, which is the main reason for running pi.reg.

The pi.reg file has two parts. The first part, up to the line "save pi.sav" brings variables into the workspace bank from the source bank, which is macro.bnk. (This macro.bnk was prepared also using *G7*, from source data.) Any line beginning with the '#' character is a comment. Portions of lines preceded by '#' are also comments. The first command

```
ba macro
```

assigns the macro bank as the default source for data.

The statement

```
f labinc = nice
```

forms the new variable *labinc* (labor income, or Compensation of employees) in the workspace by copying the variable *nice* from the source bank. After creating

> If you're curious about the contents of macro.bnk, simply type
>
> ```
> look a
> ```
>
> in the *G7* command box. Then you can scroll through the variable list, double-clicking on any variables you'd like to examine. The macro.bnk has nearly 800 variables, from which the macrovariables in *Tux* are drawn.

several variables in the workspace bank, a check value for personal income is calculated:

```
f pichk = labinc + pintdiv + piprop + piren + pigsb + pibtp - picsi
```

(These correspond to the variables shown in table 2.1.) A graph is then created to compare this check value with the actual value of *pi*.

The second part of the file starts with “save pi.sav”. This command opens up a new file pi.sav for writing, and writes several types of statements into it. These are usually either “f”, “fex”, “id” or “r” statements. Table 2.2 summarizes the function of each of these commands.

Table 2.2 Save File Commands for *IdBuild*

Command	Function	In the model code?	In the hist.bnk?
f	Forms a variable as a function of other macrovariables and vector variables	Yes	Yes
fex	Same	No	Yes
id	Same	Yes	No
r	Performs a regression estimation, and writes out code to go into the model	Yes	Yes

The pi.reg file has examples of “f”, “fex” and “id”. The command “save off” closes and completes the writing of the file pi.sav. Before discussing the differences between these commands, let’s continue to trace through the macrovariable modeling procedure.

2.4 The Save File

The save file (conventionally ending with “.sav”) contains commands which will be passed to the *IdBuild* program, which will then write out a C++ program file named heart.cpp. While you have *G7* open, look at the pi.sav file you just generated with

```
ed pi.sav
```

The pi.sav file is shown below:

pi.sav

```
fex pintdivrat = pintdiv/capinc
f pintdiv = pintdivrat*capinc
fex piproprat = piprop/capinc
f piprop = piproprat* capinc
fex pirenrat = piren/capinc
f piren = pirenrat* capinc
fex pibtprat = pibtp/capinc
f pibtp = pibtprat*capinc
fex pigsbrat = pigsb/g
f pigsb = pigsbrat*g
fex picsirat = picsi/pi[1]
f picsi = picsirat*pi[1]
id pi = labinc + pintdiv + piprop + piren + pigsb + pibtp - picsi
```

Comments in the .reg file do not get passed through to the pi.sav file. The “f”, “fex” and “id” lines are passed through verbatim.

2.5 The *IdBuild* Program

The *IdBuild* program processes commands, usually in an input file, and writes out C++ code, into a file named heart.cpp by default. We conventionally name the input file master, but it can have any name. The master file for the *Tux* model is shown below:

master

```
#  Master File for Tux9: Model 1
iadd pseudo.sav
iadd pi.sav
iadd account.sav
iadd vfix.sav
ba exim
iadd exim.sav
iadd Fixes.sav
end
```

This version of the master file uses 6 save files.

As in many files, lines beginning with '#' are a comment. Except for with the pseudo.sav file, the "iadd" (Interdyme add) command tells *IdBuild* to process that .sav file and write a function into heart.cpp that has the same name as the first part of the filename, but with an "f" on the end. For example, the function for pi.sav will be named pif().

Here is the code for the pif() function, as written to heart.cpp by *Idbuild*:

pif() Function

```
void pif()
{
 pintdiv[t]= pintdivrat[t]* capinc[t];
 piprop[t]= piproprat[t]* capinc[t];
 piren[t]= pirenrat[t]* capinc[t];
 pibtp[t]= pibtprat[t]* capinc[t];
 pigsb[t]= pigsbrat[t]* g[t];
 picsi[t]= picsirat[t]* pi[t-1];
 pi[t]= labinc[t]+ pintdiv[t]+ piprop[t]+
 piren[t]+ pigsb[t]+ pibtp[t]- picsi[t];
}
```

This is C++ code. Even if you have experience with programming C++, it won't hurt to review, as well as to describe a few special features of *Interdyme* C++ code. The first line "void pif()" indicates that this is a C++ **function**, which can be **called** by the main program. The function **body** is between the "{" and "}" characters. Every C++ **statement** is terminated by a ";' (semicolon) character. Note that the names of macrovariables in the .reg and .sav files have been rewritten with a "[t]" after them. In the C++ program, each macrovariable is actually stored as a **vector** of data, indexed over time. The variable *t* indicates the number of the year in which the model is solving, for example 2017. Typical operators are '+' for addition, '-' for subtraction, '*' for multiplication and '/' for division. The '=' operator sets the left hand side variable equal to the right hand expression. In other words, it copies the value of the right hand side expression into the left hand side variable. Notice that

the last statement, beginning with “pi[t] = …” extends over two lines. The line is not finished until the terminating “;”.

All “f” and “id” statements were passed through as statements in the model code, and they are both passed in the form “<value> = <expression>”. What happened with the “fex” commands, and what is their purpose? Now we are ready to discuss this important question.

2.6 Trying it Out

As we mentioned, *IdBuild* usually works with a special input file called ‘master’. Although *IdBuild* gets called automatically when you choose the *G7* menu item Model | Run IdBuild and Compile Model, it can also be run from the command line with:

```
idbuild master
```

You can leave G7 running, but go back to the command prompt by clicking on the Rocket icon on the task bar. You will see the lines of the master file and the save files that are included with the “iadd” command.

IdBuild has just created a new hist.bnk, heart.cpp, tseries.inc, and several other files.

2.7 The Purpose of “f”, “fex” and “id” Statements

To fully understand the behaviors of these commands listed in table 2.2, we should explain a bit more about what *IdBuild* does. Item 1 on the list we have already described above:

1. Processes the master file, writing C++ function code for each .sav file called with “iadd”. These functions, and some other code, are written to the file heart.cpp. Each function has the first part of the name of the .sav file, with an ‘f’ appended.
2. Builds a *G7* databank, named hist.bnk. This bank includes all endogenous and exogenous macrovariables required in the model.
3. Writes a special type of file, called an **include file**, named tseries.inc, which contains C++ declarations for each of the macrovariables included in hist.bnk.

When the “f” command is used, *IdBuild* calculates the value of the variable on the left hand side, places that calculated value in the hist.bnk, and also adds the variables on the right hand side to hist.bnk, if they are not already in there. A line of code representing the calculation is also written to heart.cpp. In the code above, an example is the calculation of pintdiv (interest and dividend income):

```
pintdiv[t]= pintdivrat[t]* capinc[t];
```

This is an example of an **endogenous** variable, since it is found on the left hand side of the ‘=’ sign.

When the “fex” command is used, *IdBuild* calculates the value of the variable and puts it into hist.bnk, but does not write a line of code for it. So, “fex” is a tool for creating **exogenous** variables, and the variable definition is **ex**cluded from the C++ code. (You can remember the “ex” in “fex” can stand for exogenous, or excluded.) Since this variable is exogenous, a value for it must be specified in the model forecast. There are several different ways in which “fex” may be used. In the pi.reg file, we have the case where it is used to create a ratio, called a **behavioral ratio**. The ratio relates one variable to another. The ratio is calculated and saved to the databank, but the ratio’s definition is not part of the model. The model will include a line of code to calculate an endogenous variable using the ratio times the variable it is related to.

The first two lines of pi.sav are:

```
fex pintdivrat = pintdiv/capinc
f pintdiv = pintdivrat*capinc
```

The first line creates the exogenous behavioral ratio *pintdivrat*, which relates interest and dividend income (*pintdiv*) to total capital value added *capinc*. There will be a time series of values for this ratio in the databank, and it should be projected when making a forecast. The second line is used in the model to calculate *pintdiv* from the specified value of *pintdivrat* multiplied by the model-calculated value of *capinc*.

This type of modeling, relating variables by ratios, is quite a good method for many variables, which include calculating taxes using tax rates, calculating savings using savings rates, or calculating contributions using contribution rates. It is also useful for relating values in real monetary terms to values in quantity terms. Examples of the latter include bushels of wheat, barrels of oil, or petajoules of energy used.

The “id” statement is used only once in pi.sav, for the calculation of *pi* itself. The line is:

```
id pi = labinc + pintdiv + piprop + piren + pigsb + pibtp – picsi
```

The behavior of “id” is almost like that of “f”, except that the calculated variable is not written to the hist.bnk. Instead, the actual value of *pi* from the source databank is written. A common reason to use “id” is that adding the components on the right hand side will generate values of *pi* that are slightly different from the published values, due to rounding error. We may prefer to preserve the published values in the model databank hist.bnk.

2.8 Making Tables of Results

A simple feature in *G7* that can be used to make tables in Excel is the “(gridty)pe” command. This will type out one or more variables into a rectangular grid, which can then be copied and pasted into *Excel* or *Calc*. Here is an example, using some of the important variables you have met so far:

```
vam base
tdates 2010 2025
gridty c vfix g x m pi pidis
```

Figure 2.3 Results of "gridty" Command

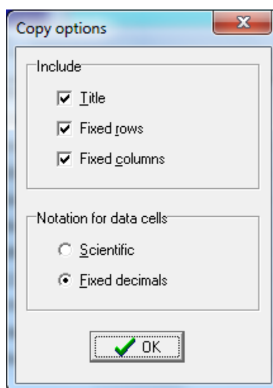
Mattype Display

Close Copy Dates

Dates	c	vfix	g	x	m	pi	pidis
2015.000	12332.257	2981.637	3218.919	2264.916	2788.958	15552.968	13615.033
2016.000	12820.694	3022.148	3267.795	2214.566	2735.805	15928.727	13968.577
2017.000	12836.316	2978.207	3310.554	2255.729	2788.029	15947.453	13985.000
2018.000	12591.659	2748.174	3353.872	2297.765	2841.362	15654.176	13727.812
2019.000	12364.264	2495.203	3397.757	2340.693	2895.827	15381.590	13488.771
2020.000	12465.054	2437.519	3442.216	2384.532	2951.447	15502.410	13594.723
2021.000	13000.095	2675.406	3487.257	2429.302	3008.248	16143.779	14157.166
2022.000	13744.222	3090.418	3532.887	2475.022	3066.254	17035.785	14939.404
2023.000	14264.376	3398.573	3579.115	2521.712	3125.491	17659.309	15486.198
2024.000	14235.196	3352.531	3625.947	2569.393	3185.986	17624.330	15455.524
2025.000	13719.354	2950.047	3673.392	2618.085	3247.764	17005.977	14913.264

Click at the bottom right corner of this grid (*pidis* in 2025) and pick the Copy menu option. You'll see the following dialog box appear:

Figure 2.4 Copying from a Grid



Click the OK button.

Now, start Excel, and create a blank worksheet. Click in the sheet where you would like to paste the data. Use the *Excel* paste function (Shift+Ins, or Ctrl+V). This is one quick method to export historical data or results of a scenario to Excel or another program.

Another way to make tables is to use the *Compare* program. This program can make tables in text, Excel (.xls) and other formats. It can show the results of historical and/or forecast data. Results for one databank can be displayed, or results from multiple databanks (scenarios) can be compared, hence the name of the program.

Compare uses table definition files, called "stub" files, usually with the file extension ".stb". Below is a stub file for *Tux* to make a table of Personal income and its components.

pi.stb

```
# pi.stb - Components of Personal Income, U.S.
\dates 2010 2012 2013 2014 2015 2016 2010-2016
\8 0
```

```
*
&
pi         ; Personal income
labinc     ;   Compensation of employees
piprop     ;   Proprietors' income
piren      ;   Rental income
pintdiv    ;   Dividend and interest income
pigsb      ;   Government social benefits
pibtp      ;   Business transfer payments to persons
picsi      ;  Less: Contributions for social insurance
\f pichk = labinc+piprop+piren+pintdiv+pigsb+pibtp-picsi
pichk      ; Personal income check
```

A table has rows and columns. The rows are either values of variables or expressions, and the columns are dates, or date expressions. An example of a date expression would be the specification of the average growth rate (two dates joined by a hypen). Table lines are typically in the format:

<variable name or expression> ; <variable text>

The variable name is the name of the variable in the model databank. For a macrovariable, this is simply the macrovariable name. For a vector, use the vector name followed by the sector number (i.e., *emp1*). For a matrix element, the row and column are separated by a period (i.e., *am10.2*).

Comments in the stub file are preceded by the '#' character. Most *Compare* commands start with the '\' character. The "\dates" command specifies the columns of the table. These may individual dates, growth rates, or other expressions which can be used to specify sums or averages. The stub file above specifies 6 columns of individual years, and one growth rate.

The command "\8 0" tells *Compare* that the fields of the table should have width 8 and 0 decimal places. The '*' command indicates to start a new page, or new worksheet if in an Excel file. The '&' command tells the program to print a line of dates.

The second line from the bottom shows one of the commands available in *Compare*. This is the "\f" command, similar to the "f" command in *G7*. In this case, it creates the variable *pichk* to provide a check of the data. The creation of this variable is local to this session of *Compare*. It is not put into the model databank. The next line uses that variable to display it.

Figure 2.5 Tables Dialog

Specify Banks, Stub, and Output File for Tables

Stub file root name (without the .stb) pi Browse

Name of output file pitable.txt

Show data from Bank 2 and above as
- Actual Values
- Differences from bank 1
- Percent differences from bank 1

Fill in a line for each bank from which you wish to draw data

	Bank type	Root name of bank		Bank type	Root name of bank
1	vam	base	6		
2			7		
3			8		
4			9		
5			10		

workspace
vam
hashed
compressed
dirfor

Load information from file | Cancel | OK

Using information from this form, the file Tables.in will be created. Clicking OK will cause TableX.bat to be run; if it does not exist, it will be created. You can use both files again with Model | Tables Express.

There are several ways to run *Compare*. From within *G7*, pick the menu option Model | Tables – Configure and Run. You'll next see the following dialog:

First specify the stub file "root name". This is the name of the file pi.stb, without the ".stb" extension, in other words, just use "pi". The second field to fill out is "Name of output file". Here I have given the name pitable.txt. The middle part of the dialog is only relevant if you are comparing two or more databanks or scenarios. This gives you the option of comparing in actual values, differences, or percentage differences. The next part of the box allows you to specify up to 10 alternative databanks. In this case, we'll just pick one. For each databank, use the drop down list at the left to specify the type of databank. For most examples in these sessions, this will be "Vam". (The "vam" option actually opens up a vam file and its accompanying macrovariables G bank as a pair.) Finally, give the root name of the databank (i.e., without the ".vam" at the end). We specify "base", for the base scenario.

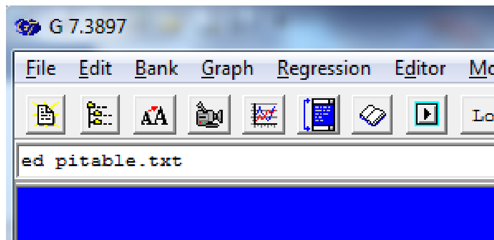
Click the OK button. You'll see the *Compare* program running in a Command Prompt window.

Figure 2.6 Run of *Compare*

```
C:\windows\system32\cmd.exe

                         COMPARE FOR WINDOWS
                            Version 6.5972
                       Copyrighted 1986 - 2016 by
                            I N F O R U M
                 Interindustry Economic Research Fund, Inc.
           P.O. Box 451, College Park, MD 20740  Tel. 301-405-4609

How many alternatives?  1
Alternative 1:
Bank type (w=workspace,c=compressed,h=hashed bank,d=dirfor,v=vam):Root name of V
am file: J:\Session2\base

With what stub? pi.stb

Name of output file: pitable.txt

Now making the table as file pitable.txt.

    Simulations being compared:
  1. Tux Model Version 1 - Base case

J:\Session2>pause
Press any key to continue . . .
```

Hit any key to continue, and you are brought back to the main G7 window. We'll use the *G7* editor to look at the table we just created. Type "ed pitable.txt" in the *G7* Command box and hit [Enter].

Figure 2.7



The editing session is shown in Figure 2.8. If the contents of your table do not line up properly, choose a font which is fixed width. In the editor menu, pick File | Font, then Courier New (9 pt).

The variable descriptions (after the ';' character) are displayed at the left margin. The body of the table shows the values, and/or growth rates of the variables and/or expressions specified. Over the period 2010-2016, which component of personal income in the U.S. grew the fastest? Which grew the slowest?

Figure 2.8 Viewing a Table in the *G7* Editor

```
J:\Session2\pitable.txt
File  Open  Save  Edit  Find  Replace  Run
+&100

                                     2010    2012    2013    2014    2015    2016   10-16
 Personal income                    12477   13915   14074   14818   15553   15929     4.1
   Compensation of employees         7961    8610    8842    9256    9708    9979     3.8
   Proprietors' income               1033    1241    1285    1316    1319    1342     4.4
   Rental income                      403     525     567     612     662     707     9.4
   Dividend and interest income      1740    2124    2056    2245    2387    2378     5.2
   Government social benefits        2282    2324    2387    2499    2631    2711     2.9
   Business transfer payments to       43      43      41      46      53      57     4.8
  Less: Contributions for social      984     952    1105    1155    1208    1245     3.9
 Personal income check              12477   13915   14074   14818   15553   15929     4.1
```

Stub files can be created for elements of vectors, matrices, or macrovariables. Quite large tables are commonly made, which will make a small book!

The command "\xls" specifies that output will go to an *Excel* file. Let's now modify pi.stb slightly, and practice some more with *Compare*. In the G7 Command box, type "ed pi.stb".

After the line "\8 0" click and type the [Enter] key to open up a new line. Type "\xls" on the new line. Now save the file by clicking the "Save" menu item, and close the *G7* editor.

Figure 2.9 Editing pi.stb

```
J:\Session2\pi.stb
File  Open  Save  Edit  Find  Replace  Run
# pi.stb - Components of Personal Income, U.S.
\dates 2010 2012 2013 2014 2015 2016 2010-2016
\8 0
\xls
*
&
pi          ; Personal income
labinc      ;   Compensation of employees
piprop      ;   Proprietors' income
piren       ;   Rental income
pintdiv     ;   Dividend and interest income
pigsb       ;   Government social benefits
pibtp       ;   Business transfer payments to persons
picsi       ;  Less: Contributions for social insurance
\f pichk = labinc+piprop+piren+pintdiv+pigsb+pibtp-picsi
pichk       ; Personal income check
```

We'll make the table again. Since all of the arguments to the dialog box will be the same (only the pi.stb file changed), use the menu option Model | Express Tables. If Excel is installed on your computer, you should see the text:

```
Writing Excel XLS spreadsheet to pitable.xls.
```

in the output from *Compare*, and an Excel file should briefly flash up in the background. Next, we'll open the Excel file. From Windows explorer, navigate to J:\Session2, and look for the file pitable.xls. Double-click on that file, and you should see the following sheet appear:

Figure 2.10 ***Compare*** **Table in Excel File**

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5								
6		2010	2012	2013	2014	2015	2016	10-16
7	Personal income	12477	13915	14074	14818	15553	15929	4.1
8	Compensation of employees	7961	8610	8842	9256	9708	9979	3.8
9	Proprietors' income	1033	1241	1285	1316	1319	1342	4.4
10	Rental income	403	525	567	612	662	707	9.4
11	Dividend and interest income	1740	2124	2056	2245	2387	2378	5.2
12	Government social benefits	2282	2324	2387	2499	2631	2711	2.9
13	Business transfer payments to pers	43	43	41	46	53	57	4.8
14	Less: Contributions for social insura	984	952	1105	1155	1208	1245	3.9
15	Personal income check	12477	13915	14074	14818	15553	15929	4.1
16								

C++ Interlude 2: Recipes

A program has much in common with a recipe. Some programs consist merely of sequential steps, like the program in the first C++ Interlude. Some recipes include the idea of a loop. "Add five cups of sugar", can be interpreted to mean "Add one cup of sugar, then another, and repeat until we have five. In C++ this might be written:

```
int i;
for(i=1;i<=5;i++)
    AddOneCupSugar();
```

The **for** loop has 3 parts, separated by semicolons. The first one is usually called *initialization*. Here, we set the integer counter *i* equal to 1. The second is called *test*. Here we check that *i* is still less than or equal to 5. The third is an *action*. The symbol '++' means "increment *i* by one". This type of loop, which is very common in C and C++, loops from 1 to 5. The value of *i* may be used or ignored by the statements in the body of the loop. If the body of the loop has more than one statement, we need to group the statements with curly brackets ({}). If we are making a cake, we may also need flour:

```
int i;
for(i=1;i<=5;i++) {
    AddOneCupSugar();
    AddOneCupFlour();
    }
```

Both *AddOneCupSugar()* and *AddOneCupFlour()* are functions, which must be defined elsewhere in the program. The indentation in these examples is simply a matter of preference of style. However, it is good to follow one indenting style consistently.

Recipes may also be conditional. If you're making American biscuits, they may be standard, buttermilk, peppery pork, strawberry or one of about 30 others! In each case, some of the ingredients are the same, some are different.

```
AddStandardDryIngredients();
if(Standard)
    AddMilk();
else if(Buttermilk) {
    AddBakingSoda();
    AddButterMilk();
    }
else if(PepperyPork) {
    AddMilk();
    AddPepperyPork();
    }
else if(Strawberry) {
    AddMilk();
    AddStrawberries();
    }
```

The conditions inside the parentheses after the if statements are examples of Boolean (true or false) expressions. If they are true, the next statement or body of statements is executed. Other looping methods are do { body of loop } while (condition); and while (condition) { body of loop}; This little program (tenfactorial.cpp) calculates the product of the first 10 integers. Remember, type 'c' to go to the \CPP folder. The program can be compiled and linked with 'bc tenfactorial'. You can then run it by typing 'tenfactorial'.

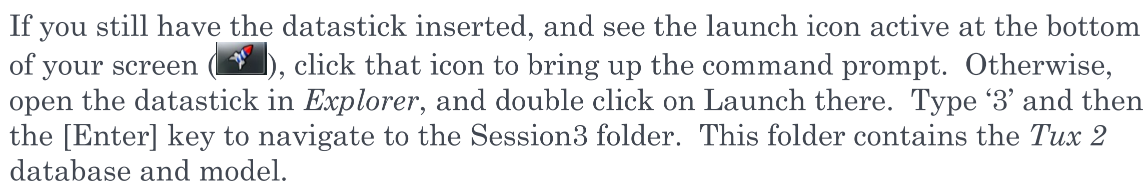
```
#include <stdio.h>
int main(void) {
    int i=0, factorial=1;
    do {
        i++;
        factorial = i*factorial;
        } while (i<10);
    printf(" 10 factorial is %d\n", factorial);
}
```

Session 3. Estimation of Sectoral Equations

Once you have mastered a technique, you hardly need look at a recipe again and can take off on your own.

In this session, we'll start to explore the creation of the sectoral equations that make up the meat of the interindustry model. The first version of *Tux* introduced in Sessions 1 and 2 did not have any sectoral equations, but rather shared down from macro totals. This is an example of a *top down* model. We will start converting *Tux* to a *bottom up* model, which we'll call version 2.

Version 1 had variables only in current prices. In this session we'll introduce IO accounts and modeling in constant prices, and skim some knowledge about solving for these prices themselves using the IO table.



If you still have the datastick inserted, and see the launch icon active at the bottom of your screen (), click that icon to bring up the command prompt. Otherwise, open the datastick in *Explorer*, and double click on Launch there. Type '3' and then the [Enter] key to navigate to the Session3 folder. This folder contains the *Tux 2* database and model.

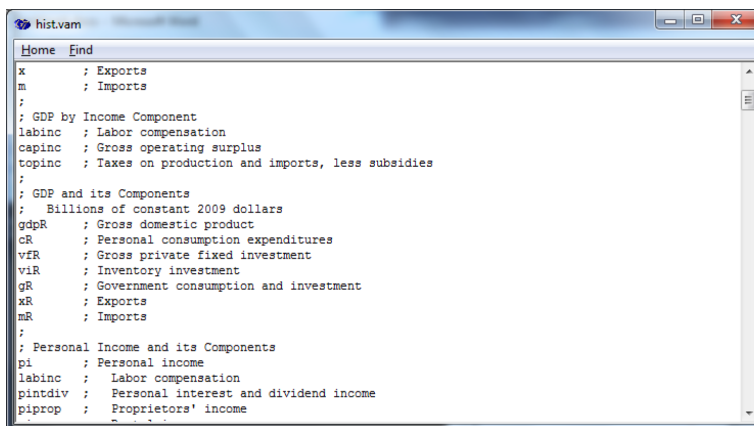
Start *G7* as before by typing 'g7' and then the [Enter] key.

Type:

```
look a
```

to see the contents of the *Tux 2* database.

Figure 3.1 The "look" Window of *Tux 2*



```
hist.vam
Home  Find
x        ; Exports
m        ; Imports
;
; GDP by Income Component
labinc   ; Labor compensation
capinc   ; Gross operating surplus
topinc   ; Taxes on production and imports, less subsidies
;
; GDP and its Components
;   Billions of constant 2009 dollars
gdpR     ; Gross domestic product
cR       ; Personal consumption expenditures
vfR      ; Gross private fixed investment
viR      ; Inventory investment
gR       ; Government consumption and investment
xR       ; Exports
mR       ; Imports
;
; Personal Income and its Components
pi       ; Personal income
labinc   ;   Labor compensation
pintdiv  ;   Personal interest and dividend income
piprop   ;   Proprietors' income
```

GDP and its components in constant prices are indicated with a capital 'R' at the end of the name. Start a new text file in the *G7* editor with the command:

```
ed real.sh
```

Add the following text to that file:

Figure 3.2 Showing Real and Nominal Variables with real.sh

```
real.sh
File Open Save Edit Find Replace Run
# real.sh - Compare some real and nominal variables

gdates 1997 2016
ti Real and Nominal GDP
gr gdpR gdp

ti Real and Nominal Consumption
gr cR c

ti Real and Nominal Fixed Investment
gr vfR vfix

# Derive GDP Deflator
f gdpP = gdp/gdpR
ti GDP Deflator
gr gdpP
```

Click on the 'Run' menu to display the 3 graphs of real and nominal GDP, and the GDP deflator. In which year do the lines cross? Which is rising faster, real or nominal?

The next section reviews the structure of the *Tux 2* database and the naming conventions.

3.1 Inventory of Ingredients

As we incorporate constant price IO data and sectoral prices into *Tux* version 2, we add new matrices, vectors and macrovariables to the model. These are like the yeast we neglected to add to the bread in *Tux 1*. Table 3.1 lists the additional matrices and vectors. This table, combined with table 1.2 provides the entire list of matrices and vectors. The names of the constant price vector variables are almost the same as the current price variables, but with the 'z' at the end removed.

The database has also been filled with data on domestic output prices (*outp*), import prices (*impp*) and a weighted price (*wtp*) which is a mixture of domestic and import prices. These are different prices for each sector.

You can view the vector of output prices with the command:

```
sh outp
```

You will notice that all of the prices are 1.0 in 2009. This is the *base year* for prices. Constant price variables are said to be in *2009 constant dollars*, or just *2009$* for short.

The constant price A-matrix in flows is *amf*. Show this matrix for 2016 with

```
sh amf y 2016
```

The vector *fd* is total final demand by commodity in millions of 2009$. The following calculation yields the sum of real final demand for the 9 sectors, in billions of 2009$.

```
f sum = @csum(fd,1-9)/1000.
```

What do you think this sum is?

Table 3.1 – Additional Matrices and Vectors in *Tux*, Version 2

Name	Description
A-Matrix, Constant Dollars	
amf	A-matrix in flows, Constant dollars
am	A-matrix coefficients
linv	Leontief inverse
Output, Final Demands and Related Vectors, Constant Dollars	
out	Output
pceio	Personal consumption expenditure
gov	Government spending
gpfi	Gross Private Fixed Investment
ven	Inventory change
ex	Exports
im	Imports
fd	Total final demand
dd	Domestic demand = out+im-ex
impshr	Imports share = im/dd
inter	Total row intermediate
intcol	Total column intermediate
Prices	
outp	Domestic output prices
impp	Import prices
wtp	Weighted domestic and import prices

You are correct! It is real GDP. This can be verified by:

```
ty sum 2012 2016
ty gdpR
```

Table 3.2 shows several new macrovariables that have been added to version 2. These are formed as the simple sums of the elements of the constant price vectors comprising the expenditure side components of GDP. As mentioned above, the real macrovariables have an 'R' at the end. The other variables are the same as in version 1.

Table 3.2 – Additional Macrovariables in *Tux*, Version 2

Name	Description
GDP and its Components, Constant Prices	
gdpR	Gross domestic product
cR	Personal consumption expenditures
vfR	Gross private fixed investment
viR	Inventory investment
gR	Government consumption and investment
xR	Exports
mR	Imports

3.2 Adding Some Spice: Investment Equations

In session 2, we learned something about the use of the *IdBuild* program for incorporating macroeconomic equations and identities into the model. To review, we started with files with the extension ".reg", which were run in *G7*. This process creates files with the extension ".sav", which will be the input for *IdBuild*. The "master" file that is read by *IdBuild* contains a sequence of "iadd" and other statements. Each "iadd" statement starts with a .sav file and then generates a callable function in "heart.cpp". The name of this function is the first part of the .sav file name, with an 'f' appended.

Remember that we used pi.reg to create pi.sav. The statement "iadd pi.sav" in master generated the heart.cpp function pif().

IdBuild can also generate equations for vector variables, such as personal consumption, investment, labor productivity and wages. We'll introduce this topic by showing how to estimate a simple set of investment equations. These will certainly not be the final version of equipment investment equations we will use, but they are a start. The vector *gpfi* is gross private fixed investment, which includes equipment, structures and intellectual property investment. If you do the command "sh gpfi", you can look at the data first, and check which sectors actually have data.

Here is gpfi.reg, with comments added to help understand the general pattern.

gpfi.reg

```
# gpfi.reg - Regressions for gross private fixed investment by sector.
# Nonzero sectors are 2,4-8
fdates 1997 2016   # "fdates" determine the dates of computations
lim 2001 2016      # "lim" sets the regression interval
gtfile sec9.gtf    # "gtf" reads a file of sectoral titles
save gpfi.sav      # "save" starts the .sav file
catch gpfi.cat     # "catch" starts the catch (.cat) file
f gppR = gdpR - gR # private real GDP
f d = gppR-gppR[1] # first difference in private real GDP
do {               # "do" command loops over sectors
  gti %1           # "gti" gets a title for this sector
  r gpfi%1 = gppR,d,d[1],d[2],d[3]   # "r" is regression
  gr *             # "gr *" plot fitted and actual values
  } (2,4-8)        # the numbers in parenthesis control the do loop
catch off          # close the catch file
save off           # close the save file
```

It's good to begin every file with a comment (started by '#') that tells the filename and what it does. The next statements set the "fdates" and limits ("lim"). If you use lagged variables (which this equation does), you generally want the "fdates" to start earlier than "lim".

Next come the "save" and "catch" statements. The save file will be the input to *IdBuild*. The catch file saves the commands and some of the output which appears on the output window, including the tables of regression results.

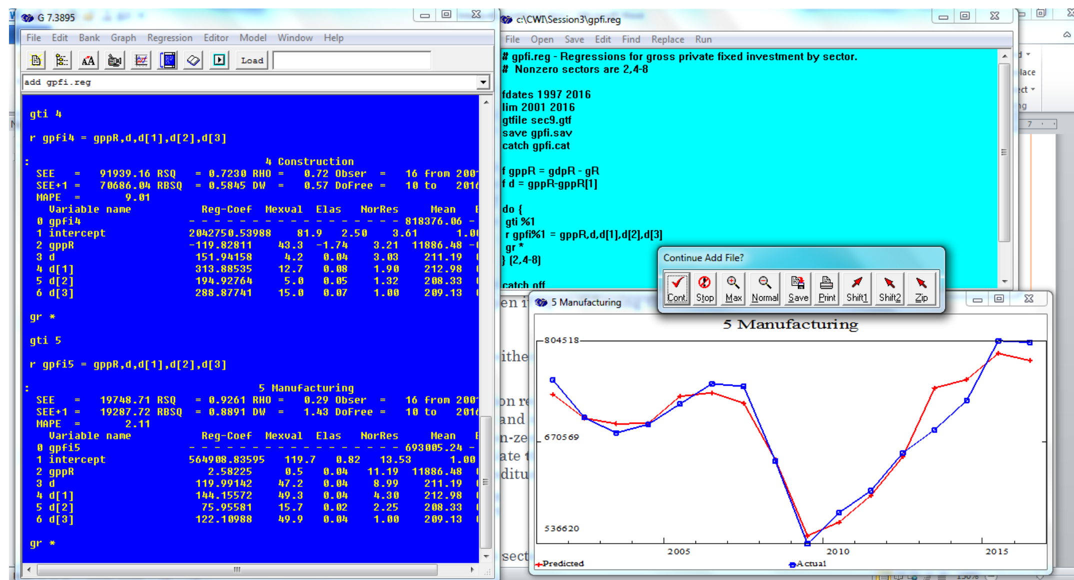
There are several new *G7* commands in this file. The first two new commands which should be explained are “gtfile” and “gti”, which work together. The “gtfile” command opens a “G7 titles file”, which we often name with the extension “.gtf”. The “gti” command, with a number, will set the title to that particular title in the titles file. This will become clear when you watch the file run.

Since you are still an apprentice chef, this file has already been created for you in the Session3 folder. Open it up in *G7* using the command:

```
ed gpfi.reg
```

Now you can run it by either clicking on the Run menu item, or pressing the F9 function key.

Figure 3.3 Running the gpfi.reg File



This investment equation relates total investment by producing sector to real gross private product (*gppR*) and its first differences (*d*).

The “do” command in *G7* runs a loop. The statements between the ‘{‘ and the ‘}’ characters are repeated. The numbers in parenthesis “(2,4-8)” indicate to run the loop over sector 2, and then sectors 4 to 8 inclusive, which are the sectors which have investment data. In each pass through the loop, the code ‘%1’ is replaced by the current sector.

Since the right-hand side variables are used by all sectors, we calculate them before the loop. Gross private product is just GDP with government expenditures (*gR*) removed. A first difference is defined as a variable minus the lagged value of that variable. Remember ‘[1]’ means the variable lagged once, ‘[2]’ lagged twice, and so on.

```
f gppR = gdpR - gR
f d = gppR-gppR[1]
```

The regression for each sector uses the same right hand side variables:

```
r gpfi%1 = gppR,d,d[1],d[2],d[3]
```

There are 6 right-hand side variables, including the constant term (intercept). Here are the regression results for sector 2:

gpfi.cat (extract)

```
r gpfi2 = gppR,d,d[1],d[2],d[3]

:                                 2 Mining and quarrying
  SEE   =   12750.89 RSQ   = 0.6348 RHO =   0.18 Obser  =   16 from 2001.000
  SEE+1 =   13049.58 RBSQ  = 0.4521 DW  =   1.65 DoFree =   10 to   2016.000
  MAPE  =      10.26
    Variable name           Reg-Coef  Mexval  Elas   NorRes     Mean   Beta
  0 gpfi2                 - - - - - - - - - - - - - - - - -  85982.05 - - -
  1 intercept             -118645.38166    18.5  -1.38    2.74      1.00
  2 gppR                    16.94981    44.8   2.34    1.09  11886.48  0.729
  3 d                       11.97100     1.4   0.03    1.07    211.19  0.132
  4 d[1]                   -12.47890     1.1  -0.03    1.07    212.98 -0.138
  5 d[2]                    22.16269     3.4   0.05    1.01    208.33  0.240
  6 d[3]                    -6.37641     0.4  -0.02    1.00    209.13 -0.069
```

The area in the top of this table includes some general descriptive regression statistics. SEE is the standard error of the estimate, a measure of the average size of the errors. RSQ is the R-squared, or general measure of fit, which varies between 0 (no fit) and 1 (perfect fit). RHO is a measure of the *autocorrelation* of the error terms. If the equation tends to miss on the same side for many periods in a row, RHO will be higher. MAPE is the *mean absolute percentage error*, which measure the average size of the error term in comparison with the left hand side variable.

Following the general statistics is one row listing the name and the mean of the left-hand side (*dependent*) variable, and six lines for the right-hand side (*independent*) variables. The first column is the variable name, and the second column is the *regression coefficient*, which tells how much the left-hand side variable will change if that right hand side variable were to increase by one unit. The column labeled 'Mexval' is the *marginal explanatory value*. This is a measure of the contribution of each variable to the overall fit of the equation. t is defined as the percentage by which the SEE would increase if that variable were to be removed from the equation. We will discuss 'Elas', 'NorRes' and 'Beta' in a future session. The column labeled 'Mean' tells the mean or average value of each variable.

Figure 3.3 Regression Plot for gpfi2

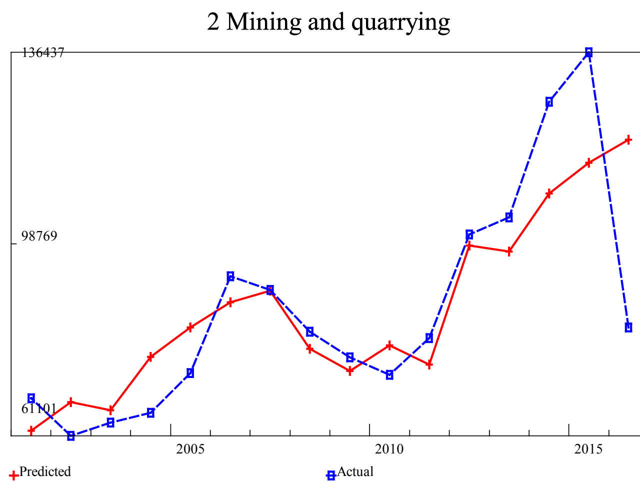


Figure 3.3 shows the regression plot, which compares the actual and fitted values of the regression.

Here is part of the gpfi.sav file created from running gpfi.reg in *G7*.

gpfi.sav

```
f gppR = gdpR - gR
f d = gppR-gppR[1]
ti 2 Mining and quarrying
r gpfi2 = -118645.381660*intercept +
          16.949807*gppR +
          11.971004*d -
          12.478899*d[1] +
          22.162694*d[2] -
           6.376413*d[3]
d
ti 4 Construction
r gpfi4 = 2042750.539877*intercept -
         119.828107*gppR +
         151.941579*d +
         313.885348*d[1] +
         194.927636*d[2] +
         288.877405*d[3]
d
< repeated for sectors 5-8 >
```

Next, we will add some lines to the master file which is used by *IdBuild*. (Note that this file has already been slightly modified from the version you saw in section 2.5.) These are highlighted in bold below:

master

```
#  Master File for Tux9: Model 2
iadd pseudo.sav
iadd RealGDP.sav
iadd pi.sav
iadd account.sav
iadd vfR.sav
ba exim
```

```
iadd exim.sav
isvector gpfi
iadd gpfi.sav
isvector clear
iadd Fixes.sav
end
```

The “isvector” command tells *IdBuild* that *gpfi* is a vector. The code that is written in heart.cpp for vectors and for macrovariables is different, as we shall see. After calling “iadd gpfi.sav”, the “isvector” command is given again, this time with the option “clear”.

After *IdBuild* is run (‘IdBuild master’) a new heart.cpp file is created. (Feel free to look at the code in heart.cpp using *np*, *ed*, *wp* or *G7* editor.) Here is the code for the investment function written by *IdBuild*.

Extract from Heart.cpp

```
void gpfif(Vector& gpfi)
{
 gppR[t]= gdpR[t]- gR[t];
 d[t]= gppR[t]- gppR[t-1];
/*  2 Mining and quarrying */
 gpfi[2] = +coef[1][0]+coef[1][1]*gppR[t]+coef[1][2]*d[t]+coef[1][3]*d[t-1]+
coef[1][4]*d[t-2]+coef[1][5]*d[t-3];
/*  4 Construction */
 gpfi[4] = coef[2][0]+coef[2][1]*gppR[t]+coef[2][2]*d[t]+coef[2][3]*d[t-1]+
coef[2][4]*d[t-2]+coef[2][5]*d[t-3];
/*  5 Manufacturing */
 gpfi[5] = coef[3][0]+coef[3][1]*gppR[t]+coef[3][2]*d[t]+
coef[3][3]*d[t-1]+coef[3][4]*d[t-2]+coef[3][5]*d[t-3];
/*  6 Trade */
 gpfi[6] = +coef[4][0]+coef[4][1]*gppR[t]+coef[4][2]*d[t]+
coef[4][3]*d[t-1]+coef[4][4]*d[t-2]+coef[4][5]*d[t-3];
/*  7 Transportation */
 gpfi[7] = coef[5][0]+coef[5][1]*gppR[t]+coef[5][2]*d[t]+coef[5][3]*d[t-1]+
coef[5][4]*d[t-2]+coef[5][5]*d[t-3];
/*  8 Services & other */
 gpfi[8] = coef[6][0]+coef[6][1]*gppR[t]+coef[6][2]*d[t]+
coef[6][3]*d[t-1]+coef[6][4]*d[t-2]+coef[6][5]*d[t-3];
}
```

Note that *macrovariables* are referenced using the variable name, with a date (t, t-1, t-2, etc.) in brackets. So ‘d[t-3]’ means the value of the variable *d* lagged three years. *Vector elements* are referenced using the vector name followed by the sector or element number in brackets. The expression ‘gpfi[7]’ means *gpfi* for sector 7.

Before looking at how the *gpfif()* function is added to the model, let’s take a peak into the model code for the first time.

3.3 A Peak Into the *Tux* Model

We promised in session 1 (“What is the *Tux* Model?”) to have a look into the computer code for the model. Now it’s time to open the oven to see what’s cooking! First, here is a very simple (‘do nothing’) model in the file Skinny.cpp that shows the main structure of the code:

Skinny.cpp

```
extern char *Version;     // Interdyme Version
#include "dymesys.h"  // All includes for general Interdyme
#include "tseries.inc" // Re-written each time by IDBUILD.
#include "heart.h"
#include "user.h"     // All global variables for user model.

// Function Prototypes go here:

void loop(void)
{
    clrscr();
    gotoxy(20,1);
    cprintf("Generic Interdyme Model");
    gotoxy(31,2);
    cprintf(Version);
    gotoxy(30,3);
    cprintf("June, 2018\r\n");

    // Matrices and Vectors:
    Matrix B("bm"), A("am"), C("cm");
    Vector out("out"), emp("emp");
    for (t = godate; t<= stopdate; t++) {
        cprintf("\r\n%5d",t);
        // Load all vectors and matrices.
        load(t);
        A = B*C;
        store(t);
        }
    printf("\nThe InterDyme run has finished.  Use G7 or Compare to view results.\n");
}
```

At the top of skinny.cpp are several #include statements, which include function declarations and global variables. The main() function for *Interdyme* is in a different file, named dyme.cpp. The file model.cpp will define the function loop() and some other functions, which implement the model. The loop() function above first prints some messages to the screen. Next come declarations for 3 matrices and two vectors. We'll delve into the mechanics of these declarations soon. For now, you should observe that the declaration

```
Matrix B("bm",'y','n'),
```

means that the matrix will be known as *B* in the program, but that it will be tied to the matrix declared as "bm" in the vam.cfg file. In other words, the declaration *links* the vector or matrix used in the program with the corresponding vector or matrix in the vam file. The vectors *out* and *emp* have the same name in the program as they have in the vam file.

Next comes the main loop of the model, over the years of the scenario. Note that although these are normally the years of a forecast, they may also be used to specify a historical simulation. The parameters *godate* and *stopdate* are usually specified in the configuration file dyme.cfg. For each year *t* a message is printed to the screen to indicate the model is solving for that year. Then all matrices and vectors declared above are loaded into memory for period *t* with the *load(t)* function call. Next follow the main calculations of the model, represented by the line 'A=B*C;'. Finally, the current set of matrices and vectors is written back to the vam file with the function call *store(t)*. The loop then proceeds to the calculations for the next year, until all years specified have been completed.

With this general structure as an introduction, let's now look at the code in the *Tux 2* model.cpp. Note how the overall structure is the same as the small model we just looked at. Everything between the *load(t)* and *store(t)* statements contains the body of the model calculations, and can be considered to replace the 'A=B*C' statement in the small model.

Core of *Tux 2* Model.cpp

```
// Loop through the years of the simulation
for (t = godate; t<= stopdate; t++) {
    // Load vectors and matrices for the current year.
    load(t);
    Iteration = 0;
    // Initialize convergence variables: consumption and investment
    oldinvtot = vfR[t]; oldpcetot = cR[t];
    while(Iteration < 20){
        Iteration++;
        // Apply exogenous macrofixes
        Fixesf();
        // Forecast final demand vectors
        vfRf(); // Aggregate investment function
        gpfi = vfR[t]*1000.*gpfic;  // billions to millions
        ven = viR[t]*1000.*venc;
        gov = gR[t]*1000.*govc;
        ex  = xR[t]*1000.*exc;
        pceio = cR[t]*1000.*pceioc;
        impshr.fix(t);
        fd = pceio + gov + gpfi + ven + ex;
        // Calculate the IO Solution
        Seidel(am, out, fd, im, ex, dd, impshr, triang, toler);
        // Recalculate fd, with imports subtracted
        fd = fd - im;
        // Forecast value added vectors, calculate prices
        labc.fix(t);
        topic.fix(t);
        gosc.fix(t);
        lab = ebemul(labc,out);
        topi = ebemul(topic,out);
        gos = ebemul(gosc,out);
        va = lab+gos+topi;
        uva = ebediv(va,out);
        outp = linv*~uva;  // Price solution
        outz = ebemul(out,outp);
        impp.fix(t);  // Import price
        imz = ebemul(impp,im);
        wtp = ebediv(outz+imz, out+im);
        // Calculate macro identities
        gdpR[t] = fd.sum()/1000.;  // millions to billions
        if(t>MacEqStartDate){
            labinc[t] = lab.sum()/1000.;
            capinc[t] = gos.sum()/1000.;
            topinc[t] = topi.sum()/1000.;
            pif();
            accountf();
            eximf();
            }
        // Check for convergence
        invdif = fabs(vfR[t] - oldinvtot);
        pcedif = fabs(cR[t]- oldpcetot);
        cprintf("Iter %2d pce = %7.1f pcedif = %6.2f invdif = %6.2f\r\n",Iteration,
            c[t],pcedif,invdif);
        oldinvtot = vfR[t]; oldpcetot = cR[t];
        if(invdif < .5 && pcedif < .5) break;
        }
    // Here when both Investment and PCE have converged
    setrho = 'y';
    vfRf();
    setrho = 'n';
    // Store vectors and matrices for the current year.
```

```
    store(t);
    }  // end t
```

The model is *simultaneous* not *recursive*, since income generated by industry production leads to the calculation of total consumption expenditures. For this reason, there is an *iteration loop*, starting with setting the iteration counter to zero. Two variables are used to check convergence: real consumption *cR* and real investment *vfR*. Each iteration, the total of consumption and investment is saved in the variables *oldpcetot* and *oldinvtot*. At the end of the iteration loop, the current values of *cR* and *vfR* are compared with the previous values, and if the difference is bigger than some predefined tolerance (0.5 in our case), the loop continues. (In this program, the maximum number of iterations is set at 20, which doesn't allow full convergence, but this can be increased.)

The first section of code calculates all final demand variables except for imports.

```
Fixesf();
// Forecast final demand vectors
vfRf(); // Aggregate investment function
gpfi = vfR[t]*1000.*gpfic;  // billions to millions
ven = viR[t]*1000.*venc;
gov = gR[t]*1000.*govc;
ex  = xR[t]*1000.*exc;
pceio = cR[t]*1000.*pceioc;
impshr.fix(t);
fd = pceio + gov + gpfi + ven + ex;
```

The line *Fixesf()* is a function that applies macrofixes to variables such as *vfR*, *viR, gR*, etc. The next line calls *vfRf()*, which is an aggregate investment function (estimated by *G7* and run through *IdBuild*). The next 5 lines use the share vectors *gpfic*, *venc*, *govc*, etc. to simply share out the macro totals. (we need to multiply the macro totals by 1000. since they are in billions and the vector data are in millions.

The vector *impshr* is defined as the share of imports in domestic demand *dd*, or *im/dd*. The vector *dd* can be defined as either:

$$dd = out + im - ex \qquad (3.1)$$

or

$$dd = inter + pceio + gpfi + ven + gov \qquad (3.2)$$

since

$$out = inter + pceio + gpfi + ven + gov + ex - im \qquad (3.3)$$

The line

```
impshr.fix(t);
```

is where fixes (if any) are applied to the *impshr* vector. *fix(t)* is a member function of the Vector class. It is called by putting a dot '.' after the vector name, and then the function call.

The next line:

```
fd = pceio + gov + gpfi + ven + ex;
```

is an example of *vector addition*. In *Interdyme*, the Vector class has several functions and operators defined. We'll see a few more of these functions and operators in the rest of the code.

In the version of *Tux* shown, the IO solution for output is next calculated with the *Seidel()* function:

```
Seidel(am, out, fd, im, ex, dd, impshr, triang, toler);
// Recalculate fd, with imports subtracted
fd = fd - im;
```

This function basically calculates the IO solution (shown below as equation 3.6), but instead of using the Leontief inverse, it uses an iterative operation known as *Gauss-Seidel*. As the iteration proceeds, imports of each commodity are calculated based on the current estimate of *dd* within *Seidel*. The vector *fd* that was used as input to *Seidel* was actually final demand before subtracting imports. After the *Seidel()* function finishes, imports are subtracted so that *fd* has the correct values.

Next we turn to the calculation of the value added vectors *lab*, *topi* and *gos*. Each value added vector has a vector of coefficients that relate that value added category to real output. For example, labc1 = lab1/out1. These can be fixed in the base.vfx file, or left constant. Next, the line 'lab=ebemul(labc,out)' does an *element-by-element multiply* of these coefficients by real output to obtain the *lab* vector. The calculations for *topi* and *gos* are similar. Finally, total value added *va* is formed as the sum, and unit value added is formed by *element-by-element division*.

```
// Forecast value added vectors, calculate prices
labc.fix(t);
topic.fix(t);
gosc.fix(t);
lab = ebemul(labc,out);
topi = ebemul(topic,out);
gos = ebemul(gosc,out);
va = lab+gos+topi;
uva = ebediv(va,out);
```

The next line

```
outp = linv*~uva;  // Price solution
```

does the IO price solution. In the line above, *outp* is the domestic output price, the matrix *linv* is the Leontief inverse, *uva* is unit value added, and the character '~' is the *transpose operator* in *Interdyme*.

The price calculations are completed with the following lines:

```
outz = ebemul(out,outp);
impp.fix(t);  // Import price
imz = ebemul(impp,im);
wtp = ebediv(outz+imz, out+im);
```

The prices calculated from the IO price solution are multiplied by real output *out* to obtain nominal output *outz*. Import prices are exogenous, and so are fixed. Nominal imports is calculated, and the weighted price *wtp* is defined as the ratio of nominal *outz+imz* and real *out+im*.

That completes the price calculation and we move on to some macroeconomic identities. Here, the very useful *sum()* function comes into play. The line:

```
gdpR[t] = fd.sum()/1000.;  // millions to billions
```

forms the macrovariable real GDP *gdpR[t]* as the sum of all elements of real final demand *fd,* dividing by 1000. to obtain billions.

The lines:]

```
if(t>=MacEqStartDate){
    labinc[t] = lab.sum()/1000.;
    capinc[t] = gos.sum()/1000.;
    topinc[t] = topi.sum()/1000.;
    pif();
    accountf();
    eximf();
    }
```

are executed only when *t* is greater than or equal to the variable *MacEqStartDate* (this is generally the last year of data availability for macrovariables, and is set in dyme.cfg.)

The macrovariables *labinc*, *capinc* and *topinc* are formed as sums of the value added vectors. The functions *pif()*, *accountf()* and *eximf()* are functions supplied using *IdBuild*.

Finally comes the check for convergence:

```
// Check for convergence
invdif = fabs(vfR[t] - oldinvtot);
pcedif = fabs(cR[t]- oldpcetot);
cprintf("Iter %2d pce = %7.1f pcedif = %6.2f invdif = %6.2f\r\n",Iteration,
    c[t],pcedif,invdif);
oldinvtot = vfR[t]; oldpcetot = cR[t];
if(invdif < .5 && pcedif < .5) break;
}
```

The *cprintf()* statement is the source of the output we see on the screen, while running the model.

We'll discuss the function of the *setrho* variable in Session 5, but for now observe that some functions are called again with *setrho* equal to 'y'. The last statement in the year loop is the *store(t)* which writes calculated values of vectors and matrices back to the vam file.

```
// Here when both Investment and PCE have converged
    setrho = 'y';
    vfRf();
    setrho = 'n';
    // Store vectors and matrices for the current year.
    store(t);
    }  // end t
```

3.4 Putting the New Equations into the Model

You have already estimated the new *gpfi* equations in G7 and modified the master file to have them written to heart.cpp by *IdBuild*. Now we'll make some slight changes to model.cpp to put the equations into the model and test them.

Close *G7* by typing the "q" command, pressing Alt+F4, or clicking on the red 'X' in the title bar. You will be returned to the command prompt.

Now type

```
np model.cpp
```

to edit the code for the model in *Notepad++*.

The model already has an equation for the aggregate investment variable *vfR*. This code is shown in bold below, starting at about line 142:

```
Fixesf();
if(t>= MacEqStartDate) {
    vfRf();
    }
gpfi = vfR[t]*1000.*gpfic;  // billions to millions
ven = viR[t]*1000.*venc;
```

We'll change this code to comment out (turn off) the aggregate equation (anything after the characters '//' is a comment, and not used in the program), and use the sectoral equations we just estimated (see the new code in bold).

```
Fixesf();
//if(t>= MacEqStartDate) {
//  vfRf();
//  vfRind[t] = vfR[t];
//  }
//gpfi = vfR[t]*1000.*gpfic;  // billions to millions
if (t>=gpfi.LastData() ) {
    gpfif(gpfi);
    vfR[t] = gpfi.sum()/1000.;
    }
```

After you have made these changes, save model.cpp with Ctrl+S, and exit with Alt+F4. Now, we will run *IdBuild* again, but this time from the command line:

```
idbuild master
```

Next, we will recompile and link the model with the statement:

```
make
```

You should see the following on the screen:

Figure 3.4 Output from 'make', When All Goes Well

```
J:\Session3>make
MAKE Version 5.2  Copyright (c) 1987, 2000 Borland
        bcc32 -c -v -a- -tWC -Od -w- model.cpp
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
model.cpp:
        bcc32 -c -v -a- -tWC -Od -w- dyme.cpp
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
Dyme.cpp:
        bcc32 -c -v -a- -tWC -Od -w- heart.cpp
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
HEART.cpp:
        bcc32 -c -v -a- -tWC -Od -w- callall.cpp
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
callall.cpp:
        ilink32 /ap/Gn/x/c/v c0x32 model.obj dyme.obj heart.obj callall.obj conf
ig.obj seidel.obj dyme.lib,dyme,, import32 cw32mt0 Borland
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
```

If the program compiles successfully, you will have a new version of dyme.exe, with your new investment equations included.

These two commands:

```
idbuild master
make
```

can also be run together using the batch file idmodel.bat, by just typing

```
idmodel
```

3.5 Tasting (Testing) the New Equations

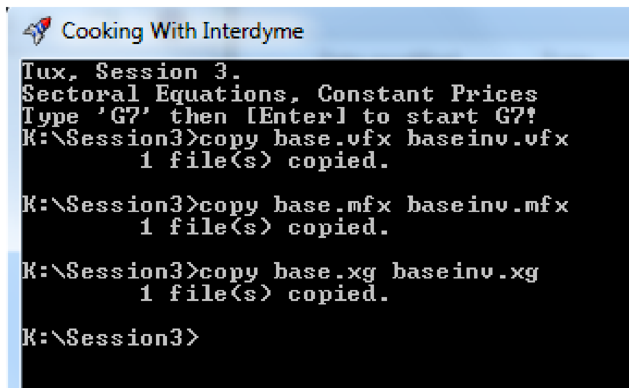
As a good chef should periodically taste her broth, a good modeler should test her model after adding new ingredients. In this section, we'll run the model with the new investment equations, and have a look at what happens.

First, double-check that you made the suggested changes to the master file and to model.cpp. Did you run 'idbuild master'? Were you able to successfully compile and link the revised model.cpp? Check the date and time of dyme.exe. If in doubt, run 'make' again.

Next, we'll create a new base, called 'baseinv' which we can compare against the previous base. I'll review the sequence for running the model.

First, copy the fixes and exogenous variable files. In the command prompt window (click on rocket icon):

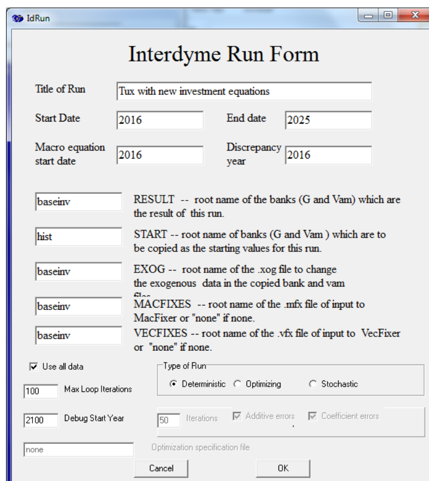
Figure 3.5 Copying the Fixes Files for New Scenario



If you are more comfortable using *Explorer* to make the copies, go right ahead.

We won't make any changes to the contents of these files, but just keep a separate copy for the new scenario. Now return to *G7*. We'll be following the same steps used in section 1.6.

Figure 3.6 Interdyme Run Form for "baseinv"



Choose the menu item Model | Run Dyme Model, or simply press the F8 key.

Fill in the Run Form as shown in Figure 3.6. If all goes well, the appearance of the model run should be quite similar to Figure 3.7.

Figure 3.7 Output of Model Run for "baseinv"

```
C:\windows\system32\cmd.exe
Seidel iterations: 5 Iter  2 pce = 16005.7 pcedif =   69.17 invdif =   2.11
Seidel iterations: 5 Iter  3 pce = 16055.1 pcedif =   38.75 invdif =   0.81
Seidel iterations: 5 Iter  4 pce = 16083.3 pcedif =   22.11 invdif =   1.21
Seidel iterations: 4 Iter  5 pce = 16099.4 pcedif =   12.62 invdif =   0.69
Seidel iterations: 4 Iter  6 pce = 16108.6 pcedif =    7.20 invdif =   0.40
Seidel iterations: 4 Iter  7 pce = 16113.8 pcedif =    4.11 invdif =   0.23
Seidel iterations: 4 Iter  8 pce = 16116.8 pcedif =    2.35 invdif =   0.13
Seidel iterations: 3 Iter  9 pce = 16118.5 pcedif =    1.34 invdif =   0.07
Seidel iterations: 3 Iter 10 pce = 16119.5 pcedif =    0.76 invdif =   0.04
Seidel iterations: 3 Iter 11 pce = 16120.0 pcedif =    0.44 invdif =   0.02
2024
Seidel iterations: 8 Iter  1 pce = 16471.2 pcedif = 3455.75 invdif = 329.21
Seidel iterations: 4 Iter  2 pce = 16543.3 pcedif =   55.39 invdif =   1.20
Seidel iterations: 5 Iter  3 pce = 16583.5 pcedif =   30.89 invdif =   0.36
Seidel iterations: 4 Iter  4 pce = 16606.4 pcedif =   17.62 invdif =   0.97
Seidel iterations: 4 Iter  5 pce = 16619.5 pcedif =   10.06 invdif =   0.55
Seidel iterations: 4 Iter  6 pce = 16626.9 pcedif =    5.74 invdif =   0.31
Seidel iterations: 4 Iter  7 pce = 16631.2 pcedif =    3.27 invdif =   0.18
Seidel iterations: 3 Iter  8 pce = 16633.6 pcedif =    1.87 invdif =   0.10
Seidel iterations: 3 Iter  9 pce = 16635.0 pcedif =    1.07 invdif =   0.06
Seidel iterations: 3 Iter 10 pce = 16635.8 pcedif =    0.61 invdif =   0.03
Seidel iterations: 3 Iter 11 pce = 16636.2 pcedif =    0.35 invdif =   0.02
2025
Seidel iterations: 8 Iter  1 pce = 16994.0 pcedif = 3828.73 invdif = 361.49
Seidel iterations: 4 Iter  2 pce = 17065.7 pcedif =   54.02 invdif =   0.97
Seidel iterations: 5 Iter  3 pce = 17105.6 pcedif =   30.09 invdif =   0.30
Seidel iterations: 4 Iter  4 pce = 17128.4 pcedif =   17.17 invdif =   0.94
Seidel iterations: 4 Iter  5 pce = 17141.4 pcedif =    9.80 invdif =   0.54
Seidel iterations: 4 Iter  6 pce = 17148.8 pcedif =    5.59 invdif =   0.31
Seidel iterations: 4 Iter  7 pce = 17153.0 pcedif =    3.19 invdif =   0.18
Seidel iterations: 3 Iter  8 pce = 17155.5 pcedif =    1.82 invdif =   0.10
Seidel iterations: 3 Iter  9 pce = 17156.8 pcedif =    1.04 invdif =   0.06
Seidel iterations: 3 Iter 10 pce = 17157.6 pcedif =    0.59 invdif =   0.03
Seidel iterations: 3 Iter 11 pce = 17158.1 pcedif =    0.34 invdif =   0.02
```

Remember to hit the [Enter] key once more when this screen appears.

Create the following file in the *G7* editor: (ed testgpfi.add).

testgpfi.add

```
# testgpfi.add - Check out the gpfi equations

vam base a
vam baseinv b
lb

t a.vfR 2014 2020
t b.vfR
gtfile sec9.gtf
sh b.gpfi

ti Compare Total Investment
gr a.vfR b.vfR 2010 2016 2025

subti New Investment Equations (blue) / Base (red)
do {
 gti %1
 gr a.gpfi%1 b.gpfi%1
} (2,4-8)
```

This file compares aggregate and sectoral investment in the original base case with the revised version including the sectoral equations. In the original base case, aggregate investment was determined by an aggregate equation, and sectoral investment was shared out by constant shares. In the new scenario, we've used the sectoral equations, and formed aggregate investment by adding them up.

What do you observe about the comparison of aggregate investment? What about sectoral investment?

If you succeeded in getting your equations to forecast, then congratulations! This is the modeling equivalent of making your first loaf of bread and actually getting it to rise. If not, don't worry, the errors are usually easy to find and fix[11].

The next section delves into some of the theory of prices in an IO model, using a small 3-sector IO database. It will be good preparation to read this before tackling session 4.

3.6 The IO Table in Constant Prices

What is an IO table in constant prices? As we will see, the answer to this is not always clear. In this session, we will follow the procedure of deriving a price for each row of the table which can be used to deflate each element of that row. That same price will be used to deflate output of each sector.

Why use an IO table in constant prices? Mainly for comparability across years. Although tables in current prices are much easier to work with (they add up nicely both by row and column), they conceal the effects of relative price change, and of general inflation. To answer questions of improvements in production, of changes in trade, and of consumer welfare, we need measures in constant prices.

The *Tux* database is already very aggregate, at 9 sectors. Let's crunch it down a bit more to 3 sectors, which is easier to gnaw on. Table 3.3 shows the data for the U.S. in 2016.

Table 3.3 Simple IO Table for the U.S., 2016 (Billions of $)

	AgMin	Industry	ServGovtM	Final Demand	Output
Agriculture and Mining	122	539	42	57	759
Industry	105	2,234	1,692	3,540	7,571
Services, Government & Misc.	122	1,515	7,090	15,028	23,755
Value Added	410	3,284	14,930	**18,624**	
Output	759	7,571	23,755		

GDP = 18,624

Source: U.S. Annual IO Tables, Bureau of Economic Analysis

As you can see, the disaggregation is lopsided, with most of the output in Services, government and miscellaneous. However, it's still an IO table. Output of each sector is equal to final demand plus total intermediate demand across the row. Output of each sector is also equal to total intermediate down the column, plus value added. GDP can be formed as either the sum of final demand or of value added. For 2016, it is $18,624 billion dollars.

The intermediate and value added elements of table 3.3 can be divided by the corresponding column output to create a coefficient matrix. Since output is the sum of intermediate and value added, these coefficients will sum to 1.0 down the column.

The base year of the U.S. database is 2009, which means that all price are equal to 1.0 in that year. Table 3.4 shows aggregate price indices for selected years.

[11] If you are still at your wits end, there are files model2b.cpp and master2b which you can copy to model.cpp and master, respectively, and the model should then work.

Table 3.4 Domestic Output Prices in the U.S.

	1997	2000	2005	2009	2013	2016
Agriculture and Mining	0.66	0.68	0.94	1.00	1.31	0.97
Industry	0.75	0.78	0.91	1.00	1.14	1.08
Services, Government & Misc.	0.75	0.81	0.91	1.00	1.07	1.12

Note that from 1997 to 2009, the price of Agriculture and mining increased the most, but it has declined since then, while the prices of the other sectors have risen.

Table 3.5 shows the IO table deflated to constant prices. All elements of the first row have been divided by 0.97, the price of sector 1 in 2016. The other rows have been divided by their prices. Real output of each sector has been divided by its corresponding price.

Table 3.5 IO Table for the U.S., 2016 (Billions of 2009$)

	AgMin	Industry	ServGovtM	Final Demand	Output
Agriculture and Mining	125	553	43	58	779
Industry	97	2,062	1,562	3,268	6,989
Services, Government & Misc.	109	1,352	6,328	13,414	21,203
Output	779	6,989	21,203		

Source: U.S. Annual IO Tables, Bureau of Economic Analysis, Inforum Deflators

Table 3.6 shows the constant price coefficients, formed by dividing the intermediate flows in constant prices by output in constant prices. We will call this matrix A. Element A(1,1) = .160, which is formed as 125/779.

Table 3.6 IO Coefficient Matrix (*A*-matrix), Constant Prices

	AgMin	Industry	ServGovtM
Agriculture and Mining	0.160	0.079	0.002
Industry	0.125	0.295	0.074
Services, Government & Misc.	0.139	0.193	0.298

If there were a reasonable definition of the "price" of value added, we could also create real value added coefficients. However, in the presence of relative price change, there would be no guarantee that the constant price IO coefficients sum to 1.0 down the column. In fact, "real value added" is usually formed as a residual, and no cogent interpretation is supplied other than that it is "needed for adding up". We can ignore this concept, as it is not a necessary or useful tool for modeling.

In matrix algebra, the expression that intermediate plus final demand is equal to output can be written:

$$\boldsymbol{q} = \boldsymbol{A}\boldsymbol{q} + \boldsymbol{f} \tag{3.4}$$

where $\boldsymbol{q}$ is output, $\boldsymbol{A}$ is the coefficient matrix we just presented, and $\boldsymbol{f}$ is the vector of final demand. Regrouping:

$$(\boldsymbol{I} - \boldsymbol{A})\boldsymbol{q} = \boldsymbol{f} \tag{3.5}$$

where I is the identity matrix. We can pre-multiply both sides by $(\boldsymbol{I} - \boldsymbol{A})^{-1}$.

$$\mathrm{q} = (\mathrm{I} - \mathrm{A})^{-1}\mathrm{f} \tag{3.6}$$

The matrix denoted by $(\mathbf{I} - \mathbf{A})^{-1}$ is called the *Leontief inverse*.

3.6 The Price Solution

The calculation of domestic output prices makes use of the fact that the price of any sector should be related to the prices of its inputs, and the amount of *value added* paid to factors. In the *Tux* model, there are three components of value added:

- *lab* – Labor compensation
- *gos* – Gross operating surplus
- *topi* - Taxes on production and imports less subsidies

Total value added is *va*, and is equal to *lab+gos+topi*, by sector. We can related value added to real output *out* in each sector *j* by the ratio:

$$uva_j = va_j / out_j \tag{3.7}$$

Think of this ratio is "value added per *unit* of output", or simply *unit value added*. The components of input cost, or *intermediate cost* of sector *j*, can be expressed as

$$ic_j = \sum_{i=1}^{n} p_i a_{ij} q_j \tag{3.8}$$

where *ic* is total intermediate cost, *p* is the domestic output price, a_{ij} is the direct IO coefficient, and *q* is real output.

Total intermediate cost per unit of output is *uic*.

$$uic_j = \frac{ic_j}{q_j} = \sum_{i=1}^{n} p_i a_{ij} \tag{3.9}$$

Total unit cost, or price, is simply unit intermediate cost plus unit value added = *uic* + *uva*

$$p_j = \sum_{i=1}^{n} p_i a_{ij} + uva_j \tag{3.10}$$

In matrix notation, this is equivalent to:

$$\boldsymbol{p}' = \boldsymbol{p}'\boldsymbol{A} + \boldsymbol{v}' \tag{3.11}$$

where $\boldsymbol{p}'$ is the row vector of domestic prices and $\boldsymbol{v}'$ is the row vector of unit value added. This identity provides a method for solving for $\boldsymbol{p}'$. Combine the two terms involving $\boldsymbol{p}'$ on the left hand side, pull out $\boldsymbol{p}'$,and then post-multiply both sides by $(\boldsymbol{I} - \boldsymbol{A})^{-1}$.

$$\boldsymbol{p}' - \boldsymbol{p}'\boldsymbol{A} = \boldsymbol{v}' \tag{3.12}$$

$$\boldsymbol{p}'(\boldsymbol{I} - \boldsymbol{A}) = \boldsymbol{v}' \quad 3.13$$

$$\boldsymbol{p}' = \boldsymbol{v}'(\boldsymbol{I} - \boldsymbol{A})^{-1} \quad 3.14$$

The *Leontief Inverse* appears again in this equation. It is the special ingredient for performing both the quantity and price IO solution!

One other identity is extremely helpful to understand. Pre-multiply equation 3.4 by $\boldsymbol{p}'$:

$$\boldsymbol{p}'\boldsymbol{q} = \boldsymbol{p}'\boldsymbol{A}\boldsymbol{q} + \boldsymbol{p}'\boldsymbol{f} \quad 3.15$$

The value of the equation is a scalar, as we are multiplying a 1xn vector by an nx1 vector, yielding a 1x1. Can you provide an interpretation for the meaning of each of the terms?

Now post-multiply equation 3.11 by $\boldsymbol{q}$.

$$\boldsymbol{p}'\boldsymbol{q} = \boldsymbol{p}'\boldsymbol{A}\boldsymbol{q} + \boldsymbol{v}'\boldsymbol{q} \quad 3.16$$

The two equations 3.15 and 3.16 together imply that:

$$\boldsymbol{p}'\boldsymbol{f} = \boldsymbol{v}'\boldsymbol{q} \quad 3.17$$

The left-hand side of the equation is simply the sum of $\boldsymbol{f}$ in current prices. From the definition of $\boldsymbol{v}$, the right-hand side is the sum of the value added vector. Both are equal to GDP.

G7 Cheat Sheet

Command	Description	Example
fex	Used for model building, and passed to .sav file. Creates an **ex**ogenous variable, which is **ex**cluded from the flow of model calculations. The exogenous variable may be used as a control or "lever" for scenario development	fex pisavrat = pisav/pidis
id	Used for model building, passed to .sav file. Create a line of code in the model showing an equation to be calculated, but does not create or put the left hand side variable in the hist.bnk databank.	id gdp=c+gpfi+vi+g+x-m
save <filename> **save off**	Opens up a "save" file (.sav) for writing. The save file is used by *IdBuild* as one of the components of the macrovariable part of an *Interdyme* model. A function is created in heart.cpp with the name of the save file with an 'f' added. (pi.sav -> pif()) The save file is closed when the "save off" command is encountered.	save fedgov.sav
catch <filename> **catch off**	Opens up a "catch" file (.cat) for writing. Output from a session of *G7* is captured to this file, until a "catch off" command is encountered.	catch pi.cat
(gtf)ile <filename>	Specifies a *G7* titles file (.gtf) to use for getting the title associated with a given sector or category number.	gtf sec9.gtf
(gti)tle <num>	Sets the title to that associated with the given sector or category number.	gti 2
do	Used for looping over a list of sectors or numbers. The body of the loop is enclosed in braces. After the closing brace, the range of the loop is specified by a group expression. Within the body, each index replaces the string "%1".	do { ty outz%1 2010 2016 } (1-9)

C++ Interlude 3: Eggs Are Objects!

C++ is one of those 'object-oriented programming' (OOP) languages. What is an object, you ask? I would say: something more complicated than just a variable or an array. Inside the object, there are pieces of data, some *private*, some *public*. The object also has behavior, which is defined by functions in the object (also called *methods*). Without further ado, let's introduce The Egg:

Egg.cpp

```
// Egg.cpp - How eggstravagant to be an object of desire!
#include <stdio.h>
class Egg {
    int shell, white, yolk;
 public:
    void Speak(void) { printf("Humpty says:\n\n"); }
    void Break(void) { printf(" I am broken.\n"); }
    void Scramble(void) { printf(" I am being scrambled!\n");}
    void Fry(void) { printf(" I am fried!\n");}
    void Boil(void) { printf(" Boil me 7 minutes.\n");}
    void Poach(void) { printf(" I\'m really good poached!\n");}
};

int main(void) {
    Egg Humpty;
    Humpty.Speak();
    Humpty.Break();
    Humpty.Fry();
    Humpty.Poach();
    Humpty.Boil();
    return 0;
}
```

The first part of the program, from 'Class Egg {' to the matching '};', is the *class definition*. *Egg* is now a new data type, along with *int*, *float*, *char* and *bool*. *Egg* has three member elements, all integer, named shell, white and yolk. Although we don't modify them or do anything with them in this simple program, they are carried around in the *Egg*. By default, unless preceded by the **public** specifier, all data elements and functions of a class are **private**, which means they can't be accessed directly by outsiders. *Egg* has 6 public functions, which simply print some text to the screen.

In the main function, we *declare* an Egg named 'Humpty'. Data elements and functions inside of Humpty the Egg are accessed by using the name of the variable, followed by a period ('.'), followed by the name of the data element or function. So, 'Humpty.Boil();' calls the boil function.

This program is in your \CPP folder. Go to that folder by typing 'c'. You can view or modify the program using 'np Egg.cpp'. Compile and link the program with 'bc egg' (command prompt does not care about small or capital letters). Run the program with 'egg'.

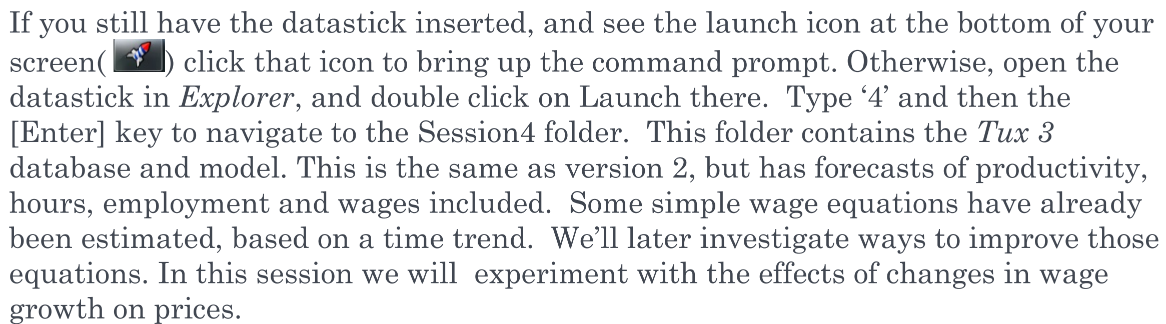
Don't get too eggcited. There is more to come!

Session 4. Wages and Prices, Government Accounts

This is my invariable advice to people: Learn how to cook- try new recipes, learn from your mistakes, be fearless, and above all have fun!

Trying a few new recipes will be our focus in this session. We'll start with an already-modified version of *Tux*, which is now at version 3. The additional changes include sectoral equations for labor productivity and annual hours worked. Data have been added for wages per hour by industry, and also some information about the ratio of *gos* (gross operating surplus) to total intermediate cost plus total labor cost, which is called *markup*. We'll focus now on three topics:

1. The estimation of simple time trend wage equations.
2. Studying the effects of changes in wages on prices
3. Developing more detailed government accounts.



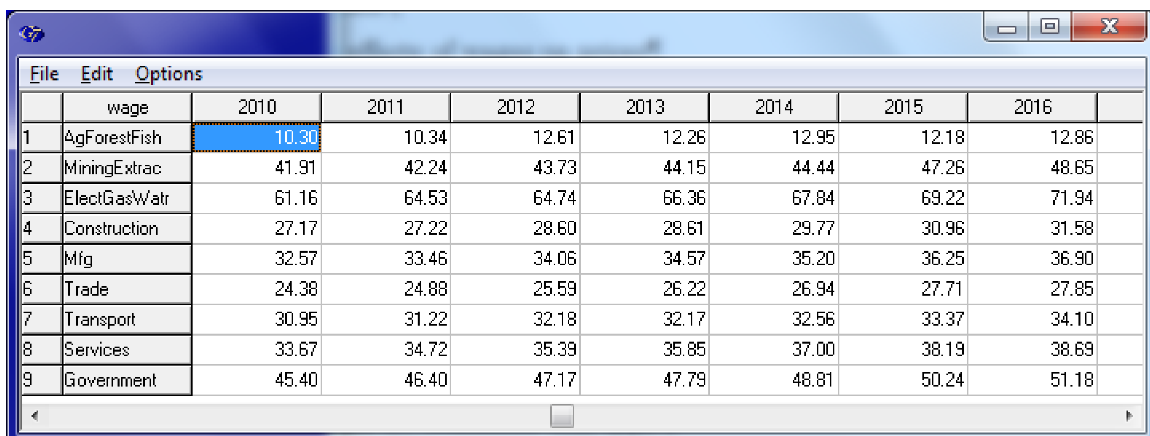
If you still have the datastick inserted, and see the launch icon at the bottom of your screen( ) click that icon to bring up the command prompt. Otherwise, open the datastick in *Explorer*, and double click on Launch there. Type '4' and then the [Enter] key to navigate to the Session4 folder. This folder contains the *Tux 3* database and model. This is the same as version 2, but has forecasts of productivity, hours, employment and wages included. Some simple wage equations have already been estimated, based on a time trend. We'll later investigate ways to improve those equations. In this session we will experiment with the effects of changes in wage growth on prices.

Start *G7* as before by typing 'g7' and then the [Enter] key. Let's have a look at wages by sector. In the *G7* command box, type:

```
sh wage 1 2010
```

We have used the "show" command with the two optional arguments, the starting sector and starting year for the display. These options are especially handy if your databank has a lot of sectors, a lot of years, or both.

Figure 4.1 Wages by Sector



File Edit Options

	wage	2010	2011	2012	2013	2014	2015	2016
1	AgForestFish	10.30	10.34	12.61	12.26	12.95	12.18	12.86
2	MiningExtrac	41.91	42.24	43.73	44.15	44.44	47.26	48.65
3	ElectGasWatr	61.16	64.53	64.74	66.36	67.84	69.22	71.94
4	Construction	27.17	27.22	28.60	28.61	29.77	30.96	31.58
5	Mfg	32.57	33.46	34.06	34.57	35.20	36.25	36.90
6	Trade	24.38	24.88	25.59	26.22	26.94	27.71	27.85
7	Transport	30.95	31.22	32.18	32.17	32.56	33.37	34.10
8	Services	33.67	34.72	35.39	35.85	37.00	38.19	38.69
9	Government	45.40	46.40	47.17	47.79	48.81	50.24	51.18

The *wage* vector is defined as labor compensation (*lab*) divided by hours worked (*hrsv*), so these figures reflect average hourly compensation.

4.1 Forecasting Wages with a Time Trend: Estimation

Forecasting a variable using a time trend can sometimes be a good first step to incorporating equations into a model. They are very stable, and won't generally cause problems in your model. However, since they don't respond to any economic influences, they are at best a straw man for comparing with more meaningful equations.

To create a time trend in *G7*, we use the @cum() function. We have encountered the @log() function earlier, when calculating the growth rate of GDP (*g_gdp*):

```
f lgdp = @log(gdp)
f g_gdp = (lgdp - lgdp[1])*100.
```

G7 includes many functions, which all begin with the '@' symbol.

The @cum() function in *G7* accumulates values of one series x into another series y, with an optional "spill rate" s. Mathematically, this can be written

$$y_t = y_{t-1} + x_t - s y_{t-1} \quad (4.1)$$

or

$$y_t = (1 - s) y_{t-1} + x_t \quad (4.2)$$

in G7, this would be written as:

```
f y = @cum(y,x,s)
```

The cum'ing starts in the first period defined by "fdates". Often this function is used to form a capital stock *K* from a series of investment expenditures *I*, with the spill rate *s* being the exponential depreciation rate of the stock.

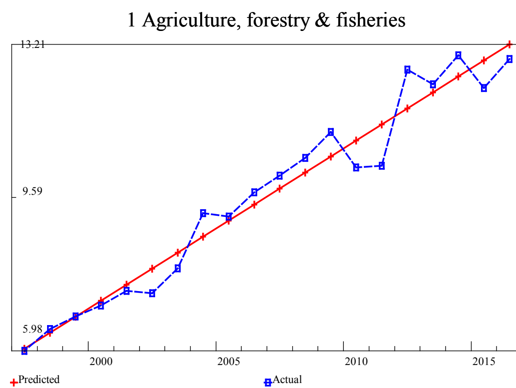
But the function has many other uses! Type the following commands in *G7*:

```
fdates 1997 2030
f timet = @cum(timet,1,0)
```

Here we are accumulating the value of 1 each period and spilling out 0. Now type the value of time. Can you see how the function works in this case? Forecasting a variable using a time trend can be as simple as regressing the variable on *timet*:

```
lim 1997 2016 2030
ti Agriculture, forestry and fisheries
r wage1 = timet
gr *
```

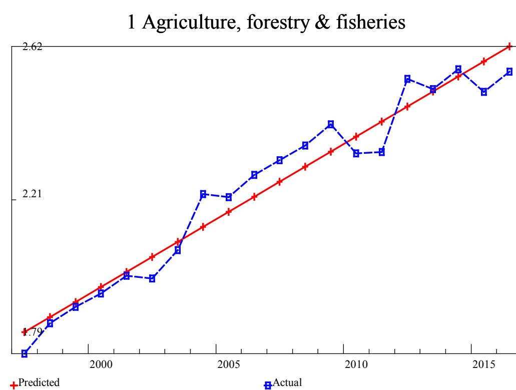
Figure 4.1 Simple Time Trend Regression



However, it is perhaps more natural to regress the *logarithm* of the variable on time:

```
f lwage1 = @log(wage1)
r lwage1 = timet
gr *
```

Figure 4.2 Log Time Trend Regression



In the regression results shown below, the coefficient on *timet* is .0409.This can be interpreted as the average annual exponential growth rate, divided by 100. The average growth rate of the wage rate in the Agriculture industry was about 4.1 percent over this period.

```
:                          1 Agriculture, forestry & fisheries
  SEE   =         0.05 RSQ   = 0.9489 RHO =   0.36 Obser  =   20 from 1997.000
  SEE+1 =         0.05 RBSQ  = 0.9460 DW  =   1.29 DoFree =   18 to   2016.000
  MAPE  =         2.11
    Variable name            Reg-Coef  Mexval  Elas   NorRes     Mean   Beta
  0 lwage1                 - - - - - - - - - - - - - - - - -     2.23 - - -
  1 intercept                1.80535  1490.7   0.81   19.56      1.00
  2 timet                    0.04090   342.2   0.19    1.00     10.50   0.974
```

Now, let’s clean off our counter, get our frying pan ready, and cook up a regression file for the wage equations.

Start a new file in the *G7* editor with the command:

```
ed wage.reg
```

A new editor screen will appear. Type in a comment on the first line, to identify the file:

```
# wage.reg - Wage equations.
```

Next, set up the "fdates" and "limits". We'll generate the right hand side variable (*time*) to run as far as 2030, to see the effects of extrapolating the regression. The three dates in the "lim" command mean to estimate the regression from 1997 to 2016, and then to forecast from 2017 to 2030.

```
fdates 1997 2030
lim 1997 2016 2030
```

The next step will be to read in the sector titles with the "gtfile" command and set up the time trend variable. We add 1996 to the time trend variable, which starts at 1 in 1997, so that it will be equal to the year:

```
gtfile sec9.gtf
f timet = @cum(timet,1,0) + 1996
```

Next, we will open up three files: a "save" file, a "catch" file and an "equation" (.eqn) file. The equation file we are seeing for the first time, and it is opened by the "punch" command. There are three parameters after the filename. The first is the maximum number of equations to save (it can be more than we have), the second is the maximum number of right hand side variables, and the third is the last year of estimation.

```
save wage.sav
catch wage.cat
punch wage.eqn 9 2 2016
```

At the bottom of the file, we need to close these files, so let's go ahead and type those lines now. All of the main code to estimate the regressions will then go in the middle, before these lines:

```
punch off
catch off
save off
```

Now here is the main code, with explanations, which goes between the two "punch" commands (you don't need to type the comments, which are text after and including the '#' character):

```
fex timet = timet       # bring timet into the databank
do {                    # start the loop
 gti %1                 # assign the title of the regression
 fex lwage%1 = @log(wage%1) # create the log wage, but don't generate code
 r lwage%1 = timet          # perform the regression
 gr *                       # make a regression plot
 ipch wage %1 a             # write into the "equation" (.eqn) file
 id wage%1 = @exp(lwage%1)  # generate code to change the solved value
                            #  to levels
} (1-9)                     # index of the loop
```

The "ipch" is a new command. It writes information into the .eqn file about the estimated values of parameters for the regression, in a format that can be read by the model.

The complete file should appear as shown below:

wage.reg

```
# wage.reg - Wage equations.

fdates 1997 2030
lim 1997 2016 2030
gtfile sec9.gtf

f timet = @cum(timet,1,0) + 1996
save wage.sav
catch wage.cat
punch wage.eqn 9 2 2016

fex timet = timet
do {
 gti %1
 fex lwage%1 = @log(wage%1)
 r lwage%1 = timet
 gr *
 ipch wage %1 a
 id wage%1 = @exp(lwage%1)
} (1-9)

punch off
catch off
save off
```

If everything looks good, go ahead and run the file in *G7* by clicking the Run menu or using the F9 function key. You can observe the regression parameters (fit, autocorrelation, regression coefficients, etc.) as well as the regression plot, for each of the 9 sectors. The output file, used by the model, is wage.eqn. The first few lines of that file are shown below:

wage.eqn

```
9 2 2016
wage 1 a 2
 1 2
     0.357478   -79.823944     0.040896
wage 2 a 2
 1 2
     0.782427   -66.177811     0.034768
wage 3 a 2
 1 2
     0.847188   -72.501709     0.038112
```

The equations we have just estimated are an example of what are called *detached coefficient equations* in *Interdyme*. In contrast to the equations estimated by *Idbuild*, which are written directly out in C++ code, the detached coefficient equations have their parameters written out to an equation (.eqn) file, which is read by the model. Some programming is necessary to implement the equations in the model. This has already been done for you in *Tux 3*. In session 5 we'll delve into the estimation of detached coefficient equations in more detail, as they are an important component of a typical model.

4.2 Effects of Wages on Prices

The IO price model enables the analysis of the effects of the change in price in one sector on the prices of all other sectors, or of the effects of changing components of value added on prices. An important component of value added is wages. Table 4.1 summarizes the time trend rates of growth from the wage rate equations estimated in section 4.1, and shows total wages, total value added, and the share of wages in value added and output for 2016.

Table 4.1 Growth Rates of Wage Rates, and Share of Wages in Value Added and Output

		2016 Values				
	Estimated Growth Rate	Wages	Total Value Added	Wage Share (%)	Total Nominal Output	Wage Share (%)
1 Agriculture, forestry, fishing and hunting	4.1	55,104	180,991	30.4	430,951	12.8
2 Mining and extraction	3.5	64,527	229,314	28.1	328,229	19.7
3 Utilities	3.8	78,423	346,182	22.7	510,288	15.4
4 Construction	3.2	540,577	883,032	61.2	1,529,777	35.3
5 Manufacturing	3.0	906,402	2,054,537	44.1	5,530,858	16.4
6 Trade	2.8	977,817	1,999,270	48.9	3,108,180	31.5
7 Transportation	2.6	333,656	579,266	57.6	1,096,268	30.4
8 Services	3.2	5,212,582	10,490,040	49.7	16,722,400	31.2
9 Government	3.8	1,823,152	1,861,842	97.9	2,827,973	64.5
Total		**9,992,240**	**18,624,475**	**53.7**	**32,084,924**	**31.1**

We'll create a new alternative scenario that increases all wage rates relative to the base case, called *HighWage*.

If you already have a command prompt window open in the \Session4 folder, you're ready. If not, click on the launch icon again on the data stick, and type '4'. Copy the 3 main input files from *Base* to *HighWage*:

Figure 4.3 Copying Fixes Files for *HighWage* Scenario

```
K:\Session4>copy base.vfx highwage.vfx
        1 file(s) copied.

K:\Session4>copy base.mfx highwage.mfx
        1 file(s) copied.

K:\Session4>copy base.xg highwage.xg
        1 file(s) copied.
```

Next, we'll edit the highwage.vfx file with *Notepad++*.

```
np highwage.vfx
```

<u>**HighWage.vfx**</u>

```
group All
 1-9

mul -s wage :All
 2016 1.0
 2030 1.2
```

Type the lines in bold above into the file. This fix is an example of a “mul” fix, where the values of a variable are multiplied by a specified factor. In our example, the factor is 1.0 in 2016 (no change), and 1.2 by 2030 (20 percent increase). The factors in the intervening years are interpolated. The first lines of the fix file set up a group called *All*, which consists of all sectors 1 to 9. The fix is an example of a *group fix* which applies to all sectors of a group. The group name is specified by placing a colon (‘:’) in front of the group name. The last item to explain is the “-s” option. Normally a group fix applies to the sum of the elements of a group. With the “-s” option, it applies the fix to each single sector. Therefore, this fix increases wage rates in all sectors by 20 percent above the base by 2030.

Now save the fixes file. We’ll run the model as we did in session 1.

Open *G7*, or switch to it, if it is already running. Pick Model | Run Dyme Model, or press the F8 function key. Fill in the dialog box as shown below.

Figure 4.4 Run Dialog for High Wage Scenario

IdRun

Interdyme Run Form

Title of Run: High Wage Scenario
Start Date: 2016 End date: 2030
Macro equation start date: 2016 Discrepancy year: 2016

highwage — RESULT -- root name of the banks (G and Vam) which are the result of this run.
hist — START -- root name of banks (G and Vam) which are to be copied as the starting values for this run.
highwage — EXOG -- root name of the .xog file to change the exogenous data in the copied bank and vam files.
highwage — MACFIXES -- root name of the .mfx file of input to MacFixer or "none" if none.
highwage — VECFIXES -- root name of the .vfx file of input to VecFixer or "none" if none.

Use all data
100 Max Loop Iterations
2100 Debug Start Year
Type of Run: Deterministic / Optimizing / Stochastic
50 Iterations / Additive errors / Coefficient errors
none Optimization specification file
Cancel OK

Click the OK button. When the Pause window appears, click OK again. Finally, when the output window showing the Seidel iterations has completed, hit any key to return to G7. Now you have a new scenario in HighWage.vam and HighWage.bnk.

The following show file, which is on your datastick already, can be used to verify the ratio of increase of wages by sector, and the resulting increase in prices. What can you conclude about the differential impacts on output prices. Which sectors’ prices increased the most, and why?

HighWage.sh

```
vam base a
vam highwage b
fdates 2000 2030
subti Compare Wage Rates
gtfile sec9.gtf
do {
 gti %1
 f ratio = b.wage%1 / a.wage%1
 ty ratio 2016 2030
 gr a.wage%1 b.wage%1 2010 2016 2030
} (1-9)

subti Compare Output Prices
gtfile sec9.gtf
do {
 gti %1
 f ratio = b.outp%1 / a.outp%1
 ty ratio 2016 2030
 gr a.outp%1 b.outp%1 2010 2016 2030
} (1-9)
```

4.3 Government Receipts and Expenditures

Questions of tax policy, public investment, transfer payments and the overall savings/investment balance in the economy are intimately related to the structure of the government accounts. In this session, we start to expand *Tux* to include accounts for Federal and State and local governments, modeling the components of receipts and expenditures to arrive at a measure of total government saving or deficit. Table 4.2 summarizes the main variables comprising federal receipts and expenditures for the U.S. from 2010 to 2016. The names in the *Variable* column will be the variable names in the databank.

Table 4.2 Federal Government Receipts and Expenditures

Title	Variable	2010	2011	2012	2013	2014	2015	2016
Current receipts	gfr	2,443	2,574	2,699	3,138	3,291	3,441	3,452
Current tax receipts	gfrt	1,353	1,554	1,661	1,824	1,995	2,127	2,100
Personal current taxes	gfrtp	942	1,129	1,165	1,302	1,403	1,529	1,541
Taxes on production and imports	gfrti	97	109	115	125	135	140	137
Taxes on corporate income	gfrtc	299	299	363	378	437	437	401
Taxes from the rest of the world	gfrtr	16	17	18	19	20	21	21
Contributions for government social insurance	gfrcsi	971	904	938	1,091	1,141	1,193	1,230
Income receipts on assets	gfra	55	56	53	163	75	49	47
Current transfer receipts	gfrct	68	67	56	71	88	77	78
Current surplus of government enterprises	gfrsurp	-3	-7	-9	-11	-8	-5	-4
Current expenditures	**gfe**	**3,772**	**3,818**	**3,789**	**3,782**	**3,901**	**4,028**	**4,149**
Consumption expenditures	gfec	1,004	1,006	1,008	961	954	960	965
Current transfer payments	gfet	2,333	2,327	2,301	2,346	2,449	2,573	2,648
Interest payments	gfeint	381	426	423	416	441	438	475
Subsidies	gfesub	54	60	58	59	58	57	61
Net Federal Government saving	**gfsav**	**-1,329**	**-1,244**	**-1,090**	**-644**	**-610**	**-587**	**-697**

Made using Compare and fedgov.stb with hist.vam

As we did earlier in Session 2, we'll build a .reg file called fedgov.reg to forecast these components, either by assumption, or by relating them to other variables in the model by ratio or regression. This file is shown below. Much of the workings of

this .reg file should be familiar from the earlier discussion of pi.reg. However, there are a few new features.

We introduce some lines of C++ code directly into heart.cpp in the fedgovf() function with the commands:

```
cc // MacFix macro:
cc #define MacFix(x) depend=x[t]; x.modify(depend);
```

The “cc” command means to pass the code verbatim into the destination .cpp file. The MacFix macro definition here enables us to specify where a macrofix can be applied to a variable. For example, several lines below we encounter:

```
# Personal taxes = gfrtp
fex gfrtprat = gfrtp / pi
cc MacFix(gfrtprat);
id gfrtp = gfrtprat * pi
```

You should make yourself familiar with the pattern of the three lines of code. The first, which is an “fex” statement, creates an *exogenous* variable, which is the ratio of personal federal taxes to personal income. The second line, with the MacFix() macro, is a C++ statement where fixes to *gfrtprat* will be applied. Finally, the “id” statement calculates personal federal taxes *gfrtp* from the assumed tax rate, and the value of personal income *pi*.

fedgov.reg

```
# fedgov.reg - Develop the components of federal government revenue and
#   expenditures.  Illustrate the federal government identities and the use
#   of macrofixes

ba macro a
vam hist b

fdates 1997 2016
tdates 1997 2016
gdates 1997 2016

f pi = pi
f topinc = b.topinc
f capinc = b.capinc

# set up time trend variable (= to year)
f timet = @cum(timet,1,0) + 1996

catch fedgov.cat
save fedgov.sav
cc // MacFix macro:
cc #define MacFix(x) depend=x[t]; x.modify(depend);
# -----------------
# Personal taxes = gfrtp
fex gfrtprat = gfrtp / pi
cc MacFix(gfrtprat);
id gfrtp = gfrtprat * pi

# Indirect taxes = gfrti
fex gfrtirat = gfrti / (topinc/1000.)
cc MacFix(gfrtirat);
id gfrti = gfrtirat * (topinc/1000.)

# Corporate taxes = gfrtc
fex gfrtcrat = gfrtc / (capinc/1000.)
cc MacFix(gfrtcrat);
id gfrtc = gfrtcrat * (capinc/1000.)
```

```
# Tax receipts from rest of world = gfrtr
ti Tax receipts from rest of world
r gfrtr = timet
gr *

# Total federal taxes = gfrt
id gfrt = gfrtp + gfrti + gfrtc + gfrtr

# Contributions for social insurance = gfrcsi
fex gfrcsirat = gfrcsi/pi
cc MacFix(gfrcsirat);
id gfrcsi = gfrcsirat * pi

# Income receipts from assets = gfra (exogenous)
cc MacFix(gfra);

# Current transfer receipts = gfrct
ti Current transfer receipts
r gfrct = timet
gr *

# Current surplus of government enterprises  = gfrsurp (exogenous)
cc MacFix(gfrsurp);

# Total receipts = gfr
id gfr = gfrt + gfrcsi + gfra + gfrct + gfrsurp
# ----------------
# Expenditures

# Consumption expenditures = gfec (exogenous)
cc MacFix(gfec);

# Current transfer payments = gfet (time trend)
ti Current transfer payments
r gfet = timet
gr *

# Interest payments = gfeint (time trend, but could be made a function of debt held by
#   the public times an average interest rate paid on that debt.
ti Interest payments
r gfeint = timet
gr *

# Subsidies = gfesub (time trend)
ti Subsidies
r gfesub = timet
gr *

id gfe = gfec + gfet + gfeint + gfesub

# Net federal government saving
id gfsav = gfr - gfe

save off
catch off
```

There is a similar .reg file slgov.reg for State and local government, which is not shown here, to save space.

By running each of these files in *G7*, we create fedgov.sav and slgov.sav. They can then be added to the master file. The new version of this file is shown below:

master (for Session 4)

```
#  Master File for Tux9: Model 3
iadd pseudo.sav
iadd RealGDP.sav
iadd pi.sav
```

```
iadd fedgov.sav
iadd slgov.sav
iadd account.sav
iadd vfR.sav
isvector gpfi
iadd gpfi.sav   # fixed investment
isvector clear
#isv prdv
#iadd prd.sav    # labor productivity
#isv clear
isv yhrv
iadd yhr.sav    # average hours worked
isv clear
ba empwag
isv empv, yhrv, prdv, hrsv, lab
iadd empwag.sav
isv clear
ba exim
iadd exim.sav
iadd Fixes.sav
end
```

You can run *IdBuild* at the command prompt by typing:

```
idbuild master
```

The current list of macrovariables in hist.bnk can now be viewed in the file TSeries.inc. There should be well over 120 macrovariables at this point. The exogenous macrovariables, which should be fixed if you want them to change, are listed in the file Run.xog. For each exogenous macrovariable, there is a line of the form

```
add <name>.xog
```

where 'name' is the name of the macrovariable. We'll cover the use of run.xog presently, but for now it is a handy way of getting a list of the exogenous macrovariables.

If you don't fix them, they will remain flat at the last known value, currently 2016.

Two new stub (.stb) files have been created for *Tux 3* to look at the government accounts. They are named fedgov.stb and slgov.stb.

Suggested Assignments and Projects

No one is born a great cook, one learns by doing.

At this point, we take a pause, reflect on what we have learned, and play with some new recipes. The assignments below have been designed to be doable in less than an hour. Projects are more extensive, and probably require guidance and communication with a more experienced chef. However, from what you have learned so far, you should be able to see the way through a more extensive project, or at least understand better what you still need to learn to get there.

Although this is not a session *per se*, a folder named Session5 has been set up on your datastick. It is essentially a copy of Session4 with the improvements in that session included. You can use this space to experiment with the assignments and projects described below.

Create Aggregate Price Ratios

1. Create a .reg file named Prices.reg. Make the first line a comment describing the purpose of the file.
2. Include the statements:
   ```
   ba hist
   fdates 1997 2016
   ```
3. Use the "save" command to open up Prices.sav for writing. Put a "save off" command at the bottom of the file.
4. Include statements to calculate *cP*, *vfP*, *viP*, *gP*, *xP*, *mP* and *gdpP*. For example "f cP=c/cR". Note that for technical reasons, the nominal value of fixed investment is *vfix*. So, the command for *vfP* is "f vfP = vfix/vfR"
5. Run the Prices.reg file in *G7*. Look at Prices.sav to make sure it looks right.
6. Add a line in the master file "iadd Prices.sav", just after "iadd Account.sav".
7. Run "idbuild master". Check tseries.inc to see if your new variables got included. Open up G7, assign the hist bank with "ba hist", and type the values of your new price variables to check them.
8. Open up heart.cpp in the Notepad++ editor ("np heart.cpp") and find the Pricesf() function. Check the code to see if it is correct.
9. In model.cpp, add the line "Pricesf();" just after "Accountf(); Don't forget the semicolon (';') at the end of the line!
10. Save the model.cpp file, and compile and link with "make".
11. Now, run the base simulation with the new model.
12. When it has finished, start *G7* assign the scenario with "vam base", and type the values of the price deflators you have added to the model. (i.e., "ty cP 2005 2030", for example)

Estimate Personal Consumption Equations

1. Create a .reg file called pceio.reg. Make the first line commenting the purpose of the file.
2. Include the statements:
```
vam hist a
fdates 1997 2016
lim 1999 2016
```
3. Add a line to set the 9-sector titles file with “gtfile sec9.gtf”.
4. Use the “save” command to open up *pceio.sav* for writing. Put a “save off command and the bottom of the file.
5. Create a Real Disposable Income variable called *pidisR* using the nominal variable and the *cP* variable created in the previous exercise. “f pidisR = pidis/cP”.
6. Create a variable measuring the change in *pidisR* from one year to the next using a lag (*pidisR[1]*). Call this variable *dipis*. (“f dipis = pidisR – pidisR[1]”).
7. Create a do loop that excludes sector four, as there is no *pceio* data for that sector. Inside the do loop, regress *pceio* against *pidisR* and *dipis*. Run the file.
8. Add the following lines to the master file, after iadd prices.sav.
```
isvector pceio
iadd pceio.sav
isvector clear
```
9. Run “idbuild master”. Open up G7, assign the hist bank with “ba hist” and type out your new variables to make sure they are included.
10. In model.cpp, pceio is currently shared out from the macrovariable cR, which is determined in the Accountant (Accountf()). Find the line:
```
pceio = cR[t]*1000.*pceioc;
```
Comment it out with ‘//’ in front of it. Replace it with your new function, and then we add a line to control it to a variable called *pceiosum*, which is essentially *cR**1000:
```
pceiof();
control(pceio,pceiosum);
```
11. Save the model.cpp file, and compile and link with “make”.
12. Run the base case again, and look at your resulting forecast of pceio in *G7*.

Estimate “Markup” Equations for Gross Operating Surplus

Total value added is the sum of labor compensation (*lab*), gross operating surplus (*gos*) and taxes on production and imports less subsidies (*topi*). We have already experimented with the simple time trend wages equations which are used in the model to determine *lab*. Gross operating surplus is composed of many different types of income, including profits, proprietors income, rental income, net interest, depreciation (capital consumption)

1. Create a .reg file named gos.reg. Make the first line a comment describing the purpose of the file.
2. Include the statements:
```
vam hist
fdates 1997 2016
lim 1997 2016
```

3. Add a line to set the 9-sector titles file with “gtfile sec9.gtf”.
4. Add a line to calculate a time trend.
5. Use the “save” command to open up gos.sav for writing. Put a “save off” command at the bottom of the file.
6. Let’s approach this problem incrementally. Pick one sector, say Manufacturing (5), and estimate a time trend regression:
```
gti 5
r gos5 = timet
gr *
```
7. Look at the plot. How well does the time trend describe this series? You may notice that the gos5 series is cyclical, with downturns in 2001 and 2009, which were both recessions in the U.S.
8. A cyclical indicator, which we may try in this regression, is the unemployment rate *un*. Modify the regression statement to add the *un* variable:
```
r gos5 = timet, un
```
9. This variable does not improve the fit very much, however.
10. Another variable is the first difference of real GDP. Just after the “save” command, add the following statement:
```
f delgdpR = gdpR - gdpR[1]
```
11. Now revise the “lim” statement near the top of the file to “lim 2000 2016”. (We’ll be introducing some lagged variables, and so need to start the regression later.)
12. Add delgdpR and delgdpR[1] to the regression:
```
r gos5 = timet, un, delgdpR, delgdpR[1]
```
13. Now experiment with setting up a do loop to estimate this equation for *gos* for all sectors.
14. Follow the typical routine for incorporating in the model. Run gos.reg. Inspect gos.sav. Put the lines:
```
isvector gos
iadd gos.sav
isvector clear
```
in the master file. Run “idbuild master”.
15. Open up model.cpp in the Notepad++ editor (“np model.cpp”). Find the line:
```
gos = ebemul(gosc,out);
```
Comment this out with ‘//’, and replace with the line:
```
gosf(gos);
```
16. Compile the model with “make” and run the base case again. Check out the results for *gos* in *G7*.

Run a Tax Cut Scenario

Since we have incorporated fedgov.reg and slgov.reg in the model, there are variables that relate federal and state and local personal taxes to personal income. Here are the relevant lines from those files:

fedgov.reg (extract)

```
fex gfrtprat = gfrtp / pi
cc MacFix(gfrtprat);
id gfrtp = gfrtprat * pi
```

slgov.reg (extract)

```
fex gsrtprat = gsrtp / pi
cc MacFix(gsrtprat);
id gsrtp = gsrtprat * pi
```

The variables *gfrtprat* and *gsrtprat* are the ratios of taxes paid to federal and state and local governments to personal income. They are *exogenous* to the model. At this point, we have not made any special assumptions about them, so they remain flat in the forecast. You can verify this by typing them out in *G7*:

```
vam base a
ty a.gfrtprat 2010 2030
ty a.gsrtprat
```

Using what you have learned about creating alternative scenarios from sections 1.7 and 4.2, create an alternative scenario called TaxCut that reduces the federal tax ratio *gfrtprat* relative to the base case. The steps are roughly:

1. make copies of base.mfx, base.vfx and base.xg to taxcut.mfx, taxcut.vfx and taxcut.xg.
2. Add lines to base.mfx to specify a fix for *gfrtprat*.
3. Run the Taxcut scenario in *G7*.
4. Open the Taxcut scenario along with the base:
   ```
   vam base a
   vam taxcut b
   ```
5. Create a "show" (.sh) file taxcut.sh to compare the values of several important variables between the base and taxcut cases. For example, the following lines compare the tax rate variable (*gfrtprat*), and the value of real personal consumption (*cR*):
   ```
   ti Ratio of Federal Taxes to Personal Income
   gr a.gfrtprat b.gfrtprat 2010 2016 2030
   ti Personal Consumption
   gr a.cR b.cR
   ```
6. Think of some other variables to compare, and add them to the show file. Do the variables change in the expected direction? Do some variables not change? If so, see if you can figure out why not.

Session 6. Labor Productivity

A cookbook is only as good as its poorest recipe.

6.1 Background

Labor productivity is one of the most important, yet most poorly understood features of the economy. Over the last 50 years, labor productivity in the U.S. has experienced several significant slowdowns and speeding up periods. How can we explain the changes in growth rates? What is the likely growth rate in the near future, for the next 10 or 20 years? Answering this question has implications for the expected standard of living for citizens, as well as the ability to pay for government programs, and to pay off accrued government debt.

Table 6.1 GDP, Productivity & Labor Force, in the U.S.: 1977-2016

	77-87	87-95	95-01	01-11	11-16	77-16
Growth Rates						
Real GDP	**3.15**	**2.80**	**3.67**	**1.69**	**2.14**	**2.65**
Private sector real GDP	3.56	3.15	4.13	1.78	2.44	2.96
Aggregate private productivity	1.66	1.58	2.60	1.99	0.46	1.72
Labor force	1.91	1.23	1.38	0.66	0.71	1.22
Total growth of labor force and productivity	**3.57**	**2.82**	**3.98**	**2.65**	**1.17**	**2.93**
Averages						
Average unemployment rate	7.36	6.17	4.77	6.36	6.79	6.38
Average Govt. value added share of GDP	19.8%	17.3%	14.4%	12.9%	12.0%	15.7%

Table 6.1 shows growth rates for selected intervals between 1977 and 2016 of some important statistics, including a measure of aggregate private sector labor productivity. If the unemployment rate were constant, the labor force was always the same share of population, and the government was a constant share of GDP, then the growth of productivity would be a good measure of the growth in living standards. Average growth in productivity over the whole period is about 1.72 percent. With this growth rate, we find a doubling every 40.2 years.[12] The late 1990s, tagged as the "New Economy", averaged a 2.6 percent rate of growth, which if maintained, would yield a doubling in 26.7 years. At the tepid rate of growth from 2011-2016 (.046 percent), doubling would take 150.7 years!

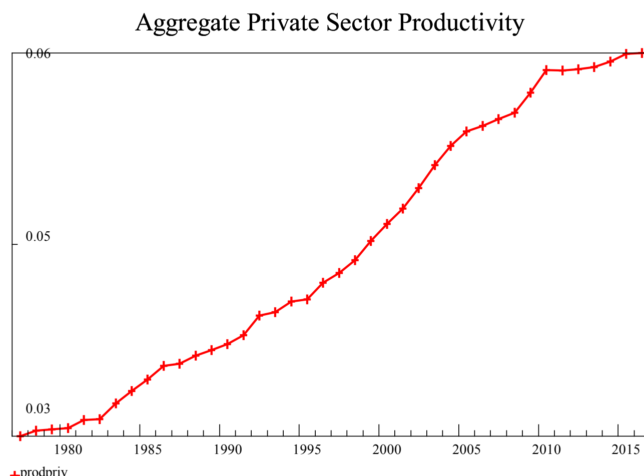
Table 6.1 also sheds light on the relationship between private labor productivity growth and real private GDP growth. Again, with the simplifying assumptions of constant unemployment rate and constant government share, then real GDP growth

[12] To find the doubling time T for any growth rate r expressed as a percent, use the formula $T = \ln(2)/(r/100)$. Since ln(2)*100=69.3, a rough rule of thumb is to divide 70 by the growth rate to obtain the approximate doubling period.

should be approximately equal to the growth rate of the labor force plus the growth rate of productivity.[13]

Figure 6.1 shows the aggregate growth rate in a continuous graph from 1977 to 2016[14]. The rapid productivity growth centering on the year 2000, and the slow growth after 2011 are both readily apparent from this graph.

Figure 6.1 Aggregate Labor Productivity



The measure of aggregate private sector labor productivity we've shown so far is calculated as private real GDP (GDP less government value added) divided by private hours worked. In an interindustry model such as *Tux*, we model sectoral labor productivity, which is calculated as real output divided by hours worked.

Table 6.2 shows sectoral productivity growth rates at the 9-sector level, for the same periods as in table 6.1.

Table 6.2 Sectoral Productivity Growth in the U.S.: 1977-2016

	77-87	87-95	95-01	01-11	11-16	77-16
Agriculture, forestry & fisheries	3.74	2.57	3.34	1.27	1.56	2.52
Mining and extraction	1.37	2.19	3.20	-1.98	4.33	1.34
Electricity, gas & water	-0.76	1.12	7.86	-2.49	0.50	0.67
Construction	0.09	-0.96	-0.45	-1.41	0.97	-0.48
Manufacturing	1.25	2.02	3.19	3.10	0.91	2.14
Trade	1.72	2.79	3.46	2.45	1.55	2.37
Transportation	0.66	1.23	0.65	1.49	-1.10	0.76
Services	0.78	0.61	2.15	1.17	0.43	1.01
Government	0.95	1.87	9.34	0.90	-0.20	2.27

The sectoral growth rates of course show a lot of variation, both across time periods and across sectors, even at this fairly aggregate level of 9 sectors. The construction sector has recorded an average of -0.48 growth over the period 1977

[13] In fact, the growth rates are not extremely close, due to changes in the unemployment rate, the government share of value added, changes in average hours worked, and changes in the spread between the industry and household measures of employment.

[14] The data in the *Tux* session directories begins in 1997. However, the datastick includes 3 historical folders at different sector levels: hist9, hist17 and hist47. These folders include IO and some other data back to 1967.

to 2016! Agriculture, Manufacturing, Trade and Government have fairly high growth rates.[15] Growth in the most recent period, 2011 to 2016 has been low, but several sectors, notably Mining and extraction, have experienced an uptick in productivity. The productivity growth of Electricity, gas and water has been low on average, but quite volatile.

6.2 Forecasting Productivity with a Time Trend

These productivity explorations will be done in the folder \Session6 on your datastick. (Just type '6', then start *G7*.)

Let's first look at fitting productivity with a time trend. We'll try a regression similar to that used for the wage rate. For the recipes in this session, we'll give each version a letter indicator. The first will be 'a'.

In this first model, we simply regress the log of productivity on time. In the *Tux* model, the predicted value from this regression will be converted back to levels. The productivity from this equation will then be used in conjunction with the output forecast to forecast hours worked by industry.

Note again the use of the "(gtf)ile" and "(gti)tle" commands to use titles from a titles (.gtf) file, the use of the catch command to capture output to a catch (.cat) file, and the saving of the equation results to a save (.sav) file. The "punch" command is used to write information about each equation to an equation (.eqn) file, which can be read in by the model as an Equation object. The type 'a', is indicated in the "ipch" command, which actually writes the parameters for each equation to the .eqn file. This type indicator is used in the model to branch to different code depending on the structure and variables in the equation. The file LabProdA.reg contains the *G7* commands to estimate these regressions and create the .eqn file.

LabProdA.reg

```
# LabProdA.reg - Regressions for labor productivity by sector, using detached
#  coefficients with rho-adjustment.  This is a simple log on time trend
#  model, so the time coefficient is the estimate of the exponential growth
#  rate.
#  Sectors are 1-9

fdates 1997 2030
#lim 1997 2016 2030
lim 1997 2016 2016
gtfile sec9.gtf
f timet = @cum(timet,1,0) + 1996
save LabProdA.sav
catch LabProdA.cat
punch LabProdA.eqn 9 2 2016
do {
 gti %1
 f lprdv%1 = @log(prdv%1)
```

[15] Government output in the U.S. NIPA is measured as real labor compensation and depreciation of capital.

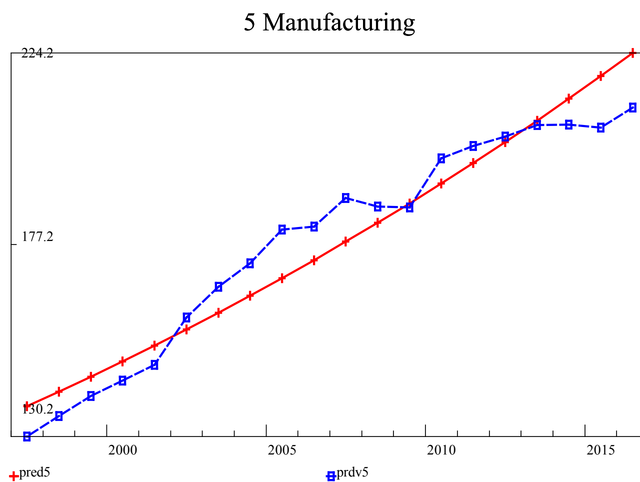
```
    r lprdv%1 = timet
    gr *
    ipch prd %1 a
   } (1-9)
```

Results for Manufacturing

```
:                                          5 Manufacturing
  SEE    =        0.04 RSQ   = 0.9309 RHO =    0.85 Obser  =   20 from 1997.000
  SEE+1  =        0.02 RBSQ  = 0.9271 DW  =    0.30 DoFree =   18 to   2016.000
  MAPE   =        0.69
    Variable name           Reg-Coef  Mexval  Elas   NorRes     Mean   Beta
  0 lprdv5                 - - - - - - - - - - - - - - - - -     5.17 - - -
  1 intercept               -46.44230   245.2  -8.99   14.48      1.00
  2 timet                     0.02572   280.5   9.99    1.00   2006.50  0.965
```

Manufacturing is one of the easier sectors to fit, with an R^2 of .9309, and an exponential trend of 2.57 percent (.02572 coefficient). (In the figure below, the actual and predicted values have been converted back to levels.) Note how labor productivity grew quickly in the expansion up to 2007, and then declined during the first years of the crisis. After a jump in 2010, growth has been slow from 2011 to 2016.

Figure 6.2 Regression Fit for Manufacturing



One of the more difficult sectors to fit in this period is Mining and quarrying, with an R^2 of .1051. Figure 6.3 shows that the productivity in this sector shows no noticeable trend, and is highly volatile. The trend is a small negative value (-0,45 percent).

Results for Mining and Quarrying

```
:                                      2 Mining and quarrying
  SEE    =        0.08 RSQ   = 0.1051 RHO =    0.81 Obser  =   20 from 1997.000
  SEE+1  =        0.05 RBSQ  = 0.0554 DW  =    0.37 DoFree =   18 to   2016.000
  MAPE   =        1.19
    Variable name           Reg-Coef  Mexval  Elas   NorRes     Mean   Beta
  0 lprdv2                 - - - - - - - - - - - - - - - - -     5.59 - - -
  1 intercept                14.60864    14.4   2.62    1.12      1.00
  2 timet                    -0.00450     5.7  -1.62    1.00   2006.50 -0.324
```

Figure 6.2 Regression Fit for Mining and Quarrying

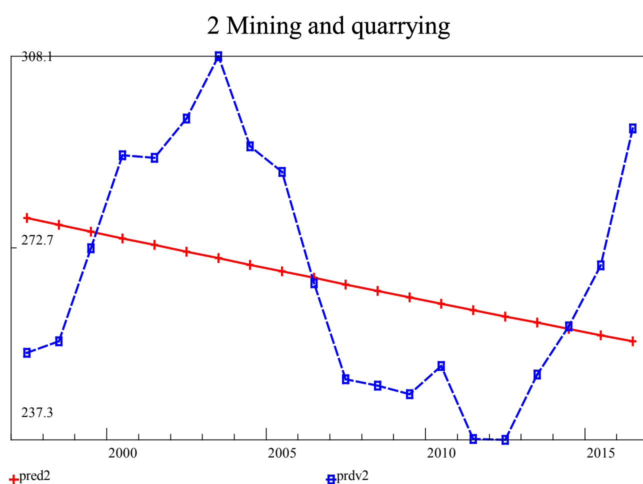


Table 6.3 summarizes the results for R^2 and the estimated time trend for each of the 9 sectors. Two sectors, Mining and extraction, and Electricity, gas and water, have a very poor fit, and no significant trend. However, these two have a small share of total employment in 2016, as shown in Table 6.4. They also have the highest *levels* of productivity. The largest shares of employment are in Services (52.3%), Trade (14.4%) and Government (14.2%). The Manufacturing share (8%) has declined from a level of 20% in 1977, due partly to rapid productivity growth, and partly to a steady increase in the import share of Manufacturing.

Table 6.3 Regression Summary for Time Trend Model 'a', 9 Sectors

#	Title	R-Squared	Trend
1	Agriculture, forestry & fisheries	0.721	1.33
2	Mining and extraction	0.105	-0.45
3	Electricity, gas & water	0.006	-0.11
4	Construction	0.703	-0.85
5	Manufacturing	0.931	2.57
6	Trade	0.900	2.21
7	Transportation	0.638	0.82
8	Services	0.885	1.16
9	Government	0.807	0.77

Table 6.4 Employment and Labor Summary, 2016

#	Title	Employment	Employment Share (%) 2016	Productivity (2009$/hr) 2016	Labor Cost Share (%) 2016
1	Agriculture, forestry & fisheries	2,298	1.5	90.6	12.8
2	Mining and extraction	633	0.4	294.8	19.7
3	Electricity, gas & water	555	0.4	463.8	15.4
4	Construction	8,538	5.4	76.4	35.3
5	Manufacturing	12,634	8.0	210.7	16.4
6	Trade	22,827	14.4	80.5	31.5
7	Transportation	5,393	3.4	98.1	30.4
8	Services	82,689	52.3	110.7	31.2
9	Government	22,435	14.2	70.1	64.5

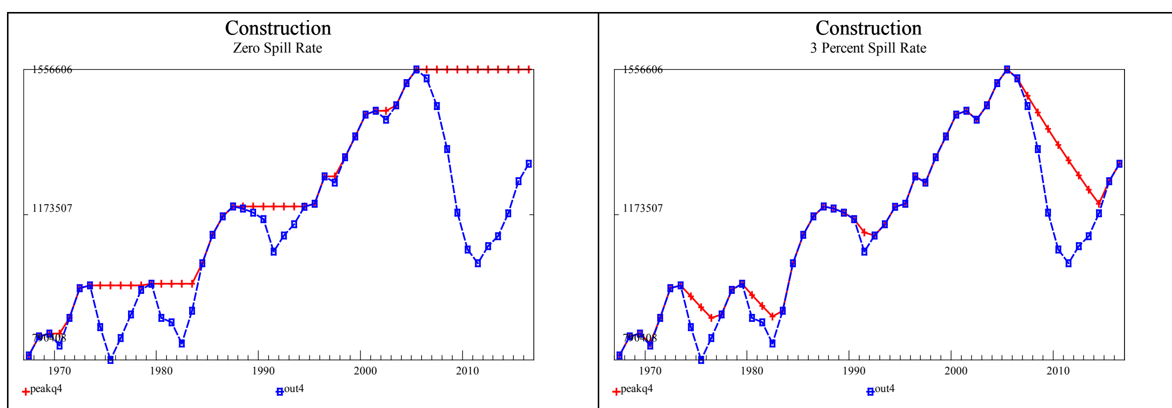
6.3 Incorporating Pro-Cyclical Labor Productivity Growth

The next model ('b') incorporates the effect of changes in output, in addition to the exponential time trend. The simplest version of such a model would be:

$$\ln(prd_j) = \mathrm{b}_0 + \mathrm{b}_1 t + \mathrm{b}_2(\ln(out_{j,t}) - \ln(out_{j,t-1})) \qquad 6.1$$

The version we present here incorporates two minor innovations to this basic setup. First, we make use of the concept of *peak output*. This is the previous peak value of output, with an option *spill rate* or depreciation rate. In *G7*, this is implemented with the @peak() function. Figure 6.3 compares actual output in the Construction sector with peak output, first with a zero spill rate (left), and then using a 3 percent spill rate (right). The concept of peak output is an attempt to measure potential capacity in place. The spill rate roughly models the depreciation of this capacity.

Figure 6.3 Illustration of the @peak() Function



The second feature is to separate positive changes from peak output from negative changes. The reasoning behind this choice is that the pro-cyclical impacts of output changes on productivity may be different when output is increasing than when it is decreasing. The positive changes are in the variable *Qup* (which is zero otherwise), and the negative changes are in *Qdown*. Equation 6.2 is the estimated equation.

$$\ln(prd_{jt}) = b_0 + b_1 t + b_2 Qup_{jt} + b_3 Qdown_{jt} \qquad 6.2$$

where:

$$Qup_{jt} = \ln(out_{jt}) - \ln(qpeak_{j,t-1}), if\ out_{jt} \geq qpeak_{j,t-1}, otherwise = 0 \qquad 6.3$$

$$Qdown_{jt} = \ln(out_{jt}) - \ln(qpeak_{j,t-1}), if\ out_{jt} < qpeak_{j,t-1}, otherwise = 0$$

The regression file for equation type 'b' is LabProdB.reg. The .reg file for each individual sector, QupQdown.reg is shown just below this.

LabProdB.reg

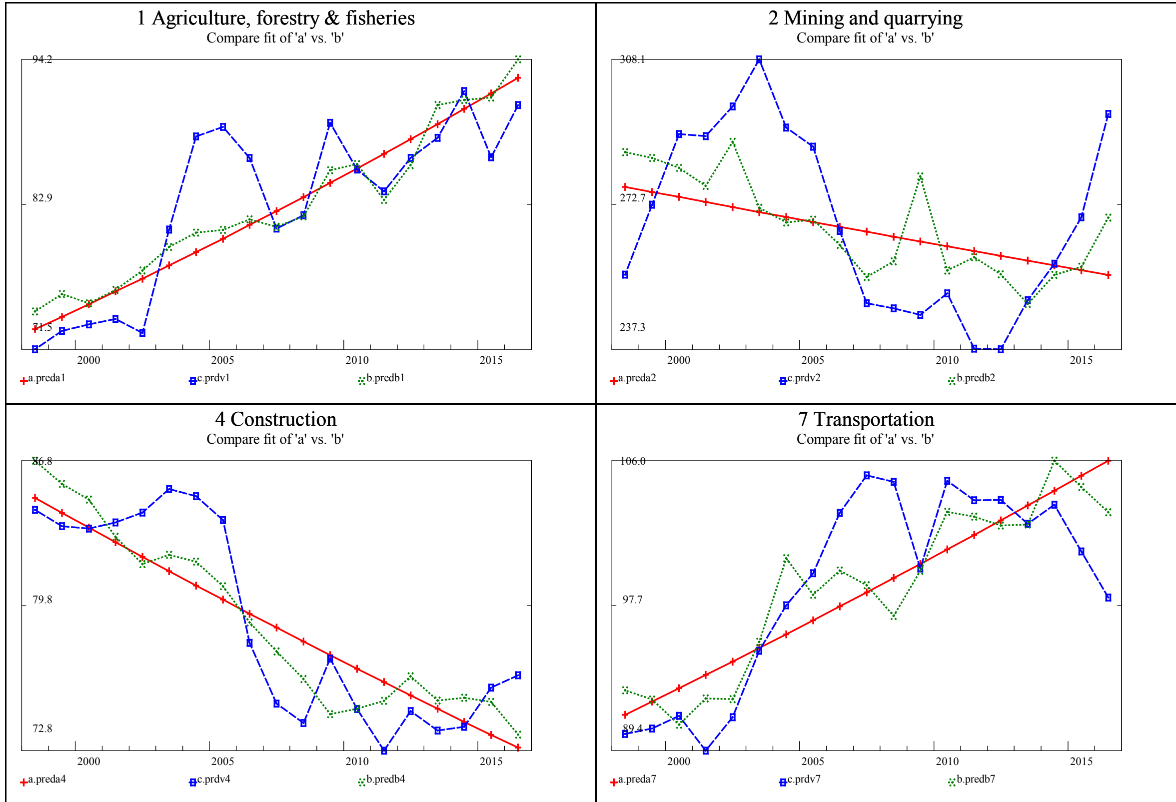
```
# LabProdB.reg - Estimate the labor productivity equations with procyclical
#   output effect for the Tux 9 model.

vam hist a
f timet = @cum(timet,1,0) + 1996
str ENDDATE = "2016"
fdates 1997 %s(ENDDATE)
gdates 1998 %s(ENDDATE)
tdates 1997 %s(ENDDATE)
lim 1998 %s(ENDDATE)

punch Labprod.eqn 9 7 %s(ENDDATE)
catch Labprod.cat
add QupQdown.reg    1  Agriculture, forestry & fisheries
add QupQdown.reg    2  Mining and quarrying
add QupQdown.reg    3  Electricity, gas & water
add QupQdown.reg    4  Construction
add QupQdown.reg    5  Manufacturing
add QupQdown.reg    6  Trade
add QupQdown.reg    7  Transportation
add QupQdown.reg    8  Services & other
add QupQdown.reg    9  Government

catch off
save off
punch off
```

QupQdown.reg

```
# QupQdown.reg - This regresses labor productivity on two time trends and
#   measures of peak output (Qup and Qdown).

ti %1 %2
f lout = @log(out%1)
f qpeak = @peak(qpeak,out%1,0.3)
f lqpeak = @log(qpeak)

f qup = lout - lqpeak[1]
f qup = @zero(qup)    # Replaces all missing observation signs in 'qup' with true zeroes
f nqdown = -1.0 * qup    # Returns negative of qup
f down = @pos(nqdown)    # Returns nqdown if nqdown > 0, otherwise 0.
f qdown%1 = -1.0 * down  # Returns negative of down
f qup%1 = @pos(qup)      # Returns qup if qup > 0, otherwise 0.
f lprod%1 = @log(prdv%1) # Log of prd (productivity)

r lprod%1 = timet, qup%1, qdown%1
ipch prdv %1 b
subti Graph in logs
gr *
```

Figure 6.4 compares the fitted values from equation 'b' (green, 'X' markers) with equation 'a' (red, '+') and the actual historical values (blue, squares), for 4 sectors. The *Qup* and *Qdown* terms help to explain part of the cyclical pattern of labor productivity, though certainly not all of it. In Mining and quarrying, the equation actually moves in the wrong direction in 2009.

Figure 6.4 Comparison of Fits ('b' vs. 'a')



Regression results are compared for two sectors below. Although type 'b' seems to follow the patterns in Agriculture better, the fit is not much different,

Comparison of Regression Results for Agriculture

Type 'a'

```
:                           1 Agriculture, forestry & fisheries
  SEE    =        0.05 RSQ   = 0.7212 RHO =    0.55 Obser  =   20 from 1997.000
  SEE+1  =        0.04 RBSQ  = 0.7057 DW  =    0.90 DoFree =   18 to   2016.000
  MAPE   =        0.86
    Variable name            Reg-Coef  Mexval  Elas    NorRes      Mean    Beta
  0 lprdv1                 - - - - - - - - - - - - - - - - -       4.40 - - -
  1 intercept              -22.27417    67.4  -5.06     3.59       1.00
  2 timet                    0.01330    89.4   6.06     1.00    2006.50   0.849
```

Type 'b'

```
:                                      1 Agriculture
  SEE    =        0.04 RSQ   = 0.7123 RHO =    0.60 Obser  =   19 from 1998.000
  SEE+1  =        0.04 RBSQ  = 0.6548 DW  =    0.80 DoFree =   15 to   2016.000
  MAPE   =        0.77
    Variable name            Reg-Coef  Mexval  Elas    NorRes      Mean    Beta
```

```
  0 lprod1                   - - - - - - - - - - - - - - - - -       4.41 - - -
  1 intercept                -20.05855    48.9  -4.54     3.48       1.00
  2 timet                      0.01219    67.5   5.54     1.13    2007.00  0.816
  3 qup1                       0.38347     0.8   0.00     1.04       0.02  0.088
  4 qdown1                     1.29691     1.9  -0.00     1.00      -0.00  0.132
```

On the other hand the fit in the construction industry improved from type ‘a’ to type ‘b’, even adjusted for degrees of freedom ($\bar{R}^2 = RBSQ$).

Comparison of Regression Results for Construction

Type ‘a’

```
:                                    4 Construction
  SEE   =        0.03 RSQ   = 0.7033 RHO =    0.69 Obser  =   20 from 1997.000
  SEE+1 =        0.02 RBSQ  = 0.6868 DW  =    0.62 DoFree =   18 to   2016.000
  MAPE  =        0.60
    Variable name            Reg-Coef  Mexval  Elas    NorRes      Mean    Beta
  0 lprdv4                 - - - - - - - - - - - - - - - - -       4.37 - - -
  1 intercept                21.41918   117.7   4.90     3.37       1.00
  2 timet                    -0.00850    83.6  -3.90     1.00    2006.50 -0.839
```

Type ‘b’

```
:                                    4 Construction
  SEE   =        0.03 RSQ   = 0.7854 RHO =    0.56 Obser  =   19 from 1998.000
  SEE+1 =        0.02 RBSQ  = 0.7424 DW  =    0.88 DoFree =   15 to   2016.000
  MAPE  =        0.57
    Variable name            Reg-Coef  Mexval  Elas    NorRes      Mean    Beta
  0 lprod4                 - - - - - - - - - - - - - - - - -       4.37 - - -
  1 intercept                22.14007   134.3   5.07     4.66       1.00
  2 timet                    -0.00886    97.1  -4.07     1.36    2007.00 -0.826
  3 qup4                      0.37407     2.5   0.00     1.07       0.02  0.140
  4 qdown4                    0.26567     3.6  -0.00     1.00      -0.02  0.169
```

6.4 Production Function Based Productivity Equation

In many of the most commonly used production functions, capital and labor are substitutable, so that capital can be increased in exchange for a reduction in labor input, given the same level of output. Fundamental to the understanding of production analysis is the Cobb-Douglas production function, generally written as

$$Q_t = \mathrm{A}_t K_t^{\alpha_K} L_t^{\alpha_L} \qquad 6.4$$

where Q is output

K is capital input

L is labor input.

If this function satisfies constant returns to scale, then

$$\alpha_K + \alpha_L = 1 \qquad 6.5$$

Taking logarithms of both sides of 6.4:

$$lnQ_t = \ln\mathrm{A}_t + \alpha_K lnK_t \ + \alpha_L lnL_t \qquad 6.6$$

Taking advantage of 6.5, this can be rearranged as

$$\ln\frac{Q_t}{L_t} = \ln A_t + \alpha_K \ln\frac{K_t}{L_t} \qquad 6.7$$

We will add the logarithm of capital over labor term to the equation, and call this version 'c'. The estimated equation is then:

$$\ln(prd_{jt}) = b_0 + b_1 t + b_2 Qup_{jt} + b_3 Qdown_{jt} + b_3 \frac{K_{jt}}{L_{jt}} \qquad 6.8$$

This equation can be interpreted as the labor productivity conditions arising out of a simple Cobb-Douglas production function, with the 'A' term expanded to include a time trend and the pro-cyclical output response. The fits of this version are slightly better than version 'b' for some sectors, and considerably better for others. Figure 6.5 compares the fits of these two versions with the actual for 4 selected sectors. In these graphs, the blue line (square markers) is actual labor productivity, the red line ('+'s) is version 'c', and the green dotted line ('X's) is the fit for version 'b'.

Figure 6.5 Comparison of Fits ('c' vs. 'b')

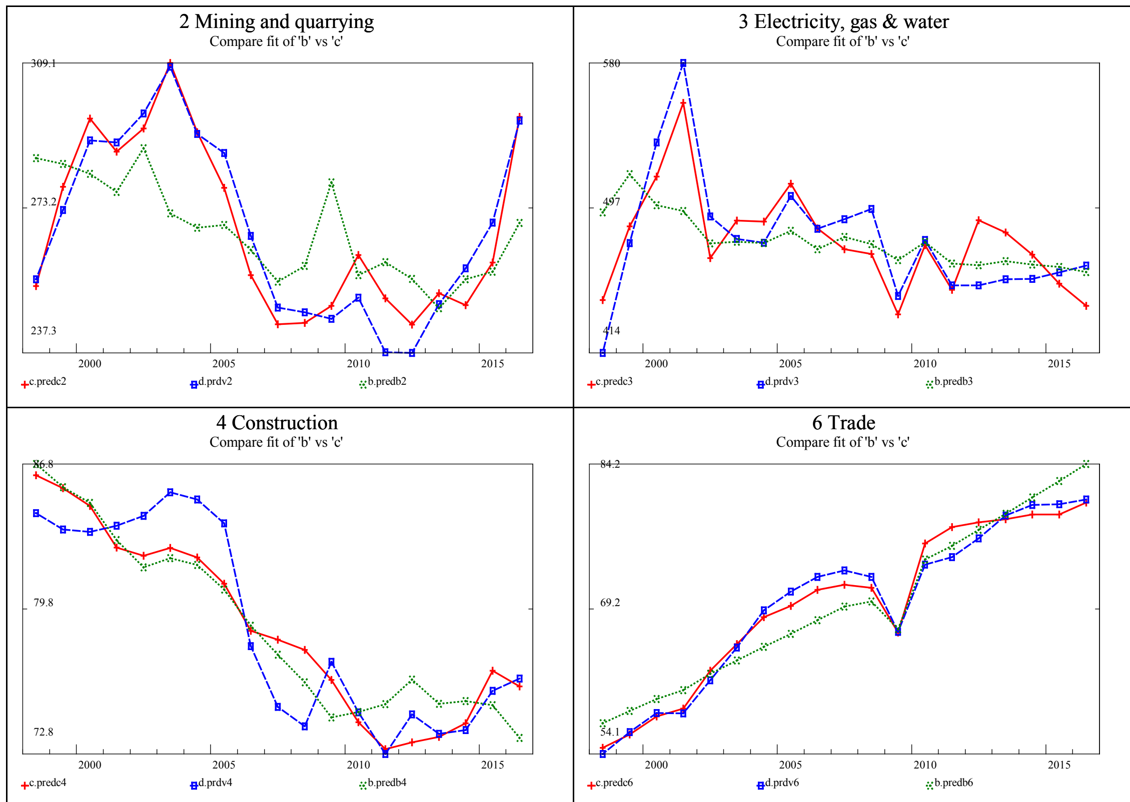


Table 6.5 compares RBSQ for the two versions.

Table 6.5 Comparison of Fit for Models 'b' and 'c'

#	Title	R-bar Squared Model 'b'	R-bar Squared Model 'c'
1	Agriculture, forestry & fisheries	0.6548	0.6691
2	Mining and extraction	0.1541	0.8735
3	Electricity, gas & water	0.0089	0.5872
4	Construction	0.7424	0.7982
5	Manufacturing	0.9154	0.9495
6	Trade	0.8901	0.9753
7	Transportation	0.6293	0.8546
8	Services	0.8785	0.9591
9	Government	0.7385	N/A