INTERDYME

A Package of Programs for Building

Interindustry Dynamic Macroeconomic Models

by

INFORUM

Version 3.10 Matching G7.31
2000 March 9

TABLE OF CONTENTS

1. OVERVIEW 1
Characteristics of Interdyme 1
Why Interindustry Models Are More Complicated than Aggregate Models 2
The Programs of Interdyme 4
Preparation Programs 4
Simulation or Forecasting Program 6
Display Programs 7
A Note on Notation 7
Installing and Checking the Software 8

2. WORKING WITH VAM FILES IN G 11
The Vam Configuration File 11
Creating, Assigning, Defaulting, and Closing a Vam File 12
Commands for Introducing Data 13
Individual series 13
Entire Vectors 17
Whole Matrices 20
Packed Matrices 21
Loading the Vam File from G banks 23
Commands for the Display of Data 24
Arguments and Looping in G 25
Commands for Sweeping Modification and Projection of Data 26
Vector Calculations 26
Groups of Sector Numbers 27
Linear Interpolation of Vectors and Matrices 28
Moving Vectors and Matrices by Indexing 28
Controlling Totals of Vectors and Subvectors 30
Matrix Balancing by the RAS Method, Coefficients and Flows 30
Commands for Writing Data to ASCII Files 32
Vam File Title and Prompting Commands 33
Vam2vam -- Selective Copying From One Vam File to Another 33
VamtoG -- Creating a G bank From Series in a Vam File 34

3. IDBUILD -- MACRO-EQUATION PROCESSOR 36
The Master File for Idbuild 36
Combining Vector and Tseries Variables in a Function With Idbuild 39

4. DETACHED-COEFFICIENT EQUATIONS 42

5. FIXES AND THE FIXER PROGRAM 47
Macro Variable Fixes 47
Vector and Matrix Fixes 53

Output Fixes....57

6. THE SIMULATION PROGRAM....58
User Written Code....58
Interdyme System Code....58
A Tutorial Example....59
The Use of Multiple Vam Files and G Banks in an Interdyme Model....68
Summary of Vector, Matrix and Tseries Functions....70
Lagged Values....73
Seidel and PSeidel....74

7. COMPARE -- TABULATING AND COMPARING RESULTS....76
The Compare Stub File....76
Matrix Listing....80

1. OVERVIEW

INTERDYME is a package of programs for building INTERindustry DYnamic MacroEconomic models such as Inforum's LIFT model or the members of its internationally-linked system of models. We speak of these models as "macroeconomic" because they deal with all the concerns of macroeconomics, such as employment and unemployment, inflation, income, interest rates, output, investment, and productivity. But they are genuine interindustry models and build up many of the aggregates from industry-level variables.

Interdyme concentrates on the extensive data management aspects of such models and leaves the user in complete freedom for development of the behavioral equations which determine the character of the model. The development of Interdyme has been strongly influenced by our experience both with a predecessor, Slimforp, and with the G-Build system for building aggregate models. The G-Build system, in fact, is an integral part of Interdyme, and this explanation will assume that the reader is already acquainted with it through the book *The Craft of Economic Modeling* Part I. Some general acquaintance with input-output ideas is also necessary. Interdyme is written in C++; a *very minimal* acquaintance with this language is needed by someone in a model building team using Interdyme. In fact, reading this manual should provider the necessary minimum. The hard part of the C++ programing has been done precisely so that the work of the model builder is extremely easy.

Characteristics of Interdyme

All Interdyme models can be built, run, and examined inside the G regression program, version 7.3 and above.

■* Vector and matrix algebra may be used in writing the program that is individual to each country model. If fd, con, inv, and gov are vectors, one can write

```
fd = con + inv + gov;
```

and get the desired result. Or if M is the import matrix and q is the vector of product outputs, one can write

```
mi = M*q;
```

to produce the vector mi of imports used in production. Since most of a model built in Interdyme is often specified in matrix algebra, coding is easy and clear. This clarity is in striking contrast to the macros which used in coaxing spreadsheets such as Excel to do model calculations.

Although, strictly speaking, this part of the coding is in C++, one needs to know very little about C++ to use tools confidently and correctly. The tricky parts of C++ are hidden well away and

need concern the user only a little more than does the fact that his web browser or word processor is written in C++.

- Interdyme is very visual. Matrices and vectors of both input and output may be viewed on the screen in a grid very like a spreadsheet. Graphs of any element of a vector or matrix or macro variable can be easily viewed and printed. Graphs and grids can be viewed simultaneously. Multiple graphs and multiple spreadsheets can be viewed.

- Equations estimated in G can be easily introduced into the model. Time series of vectors and matrices used in the model can be directly used by G in estimation of equations or in displaying and comparing results of the model graphically or in tables.

- It is common to want to run models with various modifications to its equations. These "fixes" to both macro equations and vectors are easily specified in versatile formats.

- Interdyme allows the model to begin in any year for which the necessary data are available, not just the base year of the input-output table. This capability is important for testing the model in historical simulation and for efficiency in forecasting. If a forecast has been made out to 2015 and an alternative is desired that differs from the first only after 2005, it is not necessary to rerun the model over the years prior to 2005.

- Input format is flexible, so often little pre-processing of data is necessary.

- Learning the Interdyme system is progressive. Everything learned in building a simple macro model with G applies to building interindustry models, but their are some new capabilities that need to be applied.

Why Interindustry Models Are More Complicated than Aggregate Models

The building of interindustry models is made considerably more complicated than building aggregate models by the following factors:

1. Aggregate models deal only with scalar variables; feR -- exports, for example, is one number. Interindustry models deal not only with scalars but also with numerous vectors; the scalar “exports” becomes a vector of the exports of each industry in the model. We need ways to refer both to the total vector and to its elements and groups of its elements. We must also deal with variable matrices, such as that of input-output coefficients.

2. The volumes of data to be handled require more versatile tools than were necessary for the aggregate model. Suppose, for example, that you want to increase government non-defense expenditures by two percent a year for the next ten years. In an aggregate model, you have one number to specify for each of those ten years; in an interindustry model you have as

many numbers as you have sectors, typically 50 - 100 but ranging up to over 400. Clearly, we need some wholesale way of dealing with this sort of problem.

3. As the model runs, it is often necessary to modify the results of the behavioral equations. (This modification may be necessary either to study the effects of alternative behavior or because one does not fully trust the equations for forecasting.) In aggregate models, there are usually only single, scalar variables to modify, such as exports. In interindustry models, we might, for example, wish to modify -- or "fix" -- (a) the export of a single industry (say sector 27) or (b) total exports of a group of industries (say sectors 27, 28, 30, and 35) or (c) the total exports of all industries. We need convenient ways to specify such fixes.

4. In an aggregate model, we generally expect an equation to function either in every period or (if it is "skipped") in no period. Matters are not so simple in multisectoral models because the data are not so uniformly available as in aggregate models. For example, one may have an input-output table for 1992, industry outputs through 1997, consumption expenditures through 1998, and interest rates through 1999. If we start the model in 1992, we may want to use actual historical consumption and interest data where available, so the consumption functions would not start until 1999 and the interest rate function would not start until 2000. That information must be conveyed to the model. Furthermore, there is frequently a "jump on" problem as we move from actual data to an equation. A rho-adjustment procedure is necessary. But if the last year of actual data on a variable is several years after the first year of the simulation, the initial error for the rho adjustment should be calculated in that last year, not in the first period of the simulation. Finally, in the above example, in 1993 through 1997 there will be discrepancies between the known outputs and those calculated by the model; there is a question of what to do about those discrepancies.

5. In aggregate models, each equation is pretty much *sui generis*; in a multisectoral model, many equations frequently have the same form. The employment equation for industry i may look just like the employment equation for industry j, only with different values of the parameters and different subscripts on output and employment. To keep the code to manageable proportions, it is important to be able to take advantage of this similarity. The coefficients of the equations are then stored separately from the formulas which use them. Such equations will therefore be referred to as "detached-coefficient" equations in contrast to "code-image" equations, in which the equation is written out by G with coefficients and variables just as it will appear in the C language program. The multisectoral model needs to handle both types of equations easily.

6. Numerous matrices are needed in the multisectoral model. There is not only the input-output matrix, but also numerous bridge matrices. For example, there is a matrix to convert equipment investment by investing industry to investment by product purchased, and one to convert personal consumption by budget categories to consumption by input-output industry of origin of the products. These matrices must be read and forecasted. Matrix and vector objects need to be defined and the operators =, +, -, and * "overloaded" so that equations can be written in matrix and vector notation.

The complexity of the task of building an interindustry macroeconomic model makes it necessary to lay out the whole project in broad outline before filling in details. We find it extremely important to make a very simple working version of the whole model, with all its vectors and matrices, before developing the behavioral equations for it. Once this "slim" stage of the model is working, equations can be added in small groups. If a problem arises when a group of equations is added to a working model, it is clear where the problem lies. If, on the other hand, the model is not run before all parts are finished and in place, and it then doesn't work, we are in serious trouble, for it is then not at all clear where to start looking for the problem.

The Programs of Interdyme

The programs of Interdyme fall into three broad categories:

Preparation	-- programs to prepare exogenous data, fixes, and equations.
Simulation	-- the program that makes the model run.
Display	-- programs to display the results.

We will describe the programs in this order. In practice, one program, G, serves as a framework from which the other programs can be run, often by just a mouse click.

Preparation Programs

Interdyme input and output center on a G data bank, which we will call the model G bank, and on a binary file which is called the vam file. This vam file contains all of the vectors and matrices which vary over time. Indeed, "vam" stands for "Vectors And Matrices." The vam file contains both input to the simulation program and, eventually, output from it. The model G bank contains all the individual time series which are not elements of a vector or matrix. Because these variables are often (but not always!) macroeconomic variables like the average wage rate, the interest rate, or the unemployment rate, we refer to them as the "macro" variables. In addition, there may be any number of files of equations estimated by G7. The vam file is prepared with G7, version 7.3 and above, while the model G bank is prepared with Idbuild, the adaptation of Build for Interdyme. To convey a feeling for how these programs work, we give short descriptions to be amplified below.

G7

In addition to the capacities known from building macro models, G7 version 7.3 and above has a number of features for making the vam file. The list of matrices and vectors to be put into the vam file is specified by the VAM.CFG file. This file shows the name and dimensions of every vector in the model. For vectors, it shows how many lagged values of the vector may be used in each year's calculation in the model. For each matrix, it shows whether the matrix is carried in rectangular or packed form. It gives files names for the row titles and the column titles of each vector or matrix. Finally, it has a comment to explain the economic content or other information about the matrix. Thus, this one file specifies the whole data content of the model except for the "macro" variables. Every VAM.CFG file should also contain a vector by the name of "fix", which will be used to store the numerical values associated with various "fixes".

G7 can build a vam file from data in G banks or ASCII files. Often it can read input-output tables with very minor adaptations from the way they are printed. It can then display a vector as a table with labeled rows down the side and successive years across the top. The cursor keys move the screen like a window over this table, much like a spreadsheet program with locked titles. Just as in a spreadsheet, one can edit entries in the displayed vectors. The program can also display a matrix by showing either a row or column in successive years or the matrix in a single year. It can graph an element of a matrix or a vector over time. It can interpolate elements of a vector or of a whole matrix. It can move a group of elements of a vector or matrix by an index. It can impose a control total on a group of elements. The simulation program writes the results of the forecast into a vam file, so G7 can also be used to view output in numerical grids or as graphs.

IdBuild

IdBuild is actually just the Build program familiar from macro modeling adapted for Interdyme. It puts the "code-image" or "macro" equations estimated by G7 into a C++ program. It also builds a data bank of all individual time series which are needed in the model. This is the bank we refer to as the "model G-bank". Some of the series in it are needed in the macro equations; others, such as personal disposable income, may be used in many sectoral equations. The input to Idbuild is a master file of commands of the form "iadd <filename>" where the <filename> is the name of a "save" file created with G. Once the master file and the .sav files of the equations are ready, IdBuild is executed by clicking Model | IdBuild from G7's main menu. The output includes section of code which go into the simulation program described below. This same "click," therefore, also recompiles and links the simulation program.

Fixer and MacFixer

Just as macro models sometimes required "fixes" to modify the result of the equations, so do interindustry models. Here, however, we divide them into two groups, those that are to be applied to individual macro variables and those that apply to vector variables. Because of the greater complexity here, each set is put through a preprocessor, MacFixer for the macrovariable fixes and Fixer for the vector fixes. These preprocessors may be run from the Model | MacFixer and Model | VecFixer items of the G7 main menu.

Fixer makes use of the concept of a *group* of sectors. For example, it may be desired to scale exports of manufacturing sectors to some exogenously specified total. To do so, there must be a definition of the group of "manufacturing" sectors. The "Fixer" program reads in an ASCII file of group definitions and creates a binary file (groups.bin) of these definitions for use in the forecasting program. It also creates an index of the fixes in a file with the extension .fin and it puts the numerical values of the series into the "fix" vector in the vam file.

Simulation or Forecasting Program

The simulation program -- the one which makes the model run -- carries the name "dyme". This "dyme", however, cannot be entirely written in advance, independent of the individual model. What is written in advance, however, is a framework, a schematic model, and tools for expanding it. The framework does all the housekeeping work of reading in and writing out all the individual time series and, each year, all the matrices and vectors. The tools provided are:

- matrix and vector algebra, including the use of "packed matrices" for handling large, sparse matrices, and non-standard functions such as element by element division or multiplication.
- scaling of groups of sectors in a vector to a specified total.
- automatic summation of a vector, or of parts of a vector.
- ready access to the "macro" equations.
- flexible "fixing" of the results of the macro equations with "add" and "mul" factors and rho adjustments
- Seidel solution of linear equations.
- Easy reading of detached-coefficient equations estimated with G.
- Flexible debugging printout.

The code for using the detached-coefficient equations must be provided by the model builder. Also, the economic structure of the model must be specified.

Code for handling the macro equations is written by IdBuild, so when it is executed by clicking Model | IdBuild, the simulation program is also automatically recompiled and linked.

Once the model has been built, it can be run from G7 by clicking Model | Run Dyme. A form then appears asking for the starting and stopping dates of the run, the title of the run, the base of the file name for the output files, the names of the files containing the macro fixes and the vector fixes, and certain variables for controlling the run. When they have been supplied, the model is executed by clicking the OK button.

Sometimes it has proven advantageous to construct programs that have the form of a simulation program for some steps in data preparation. A wider range of matrix and vector operations is available here than in G7 plus one has the ultimate capacity of writing directly in C++ if need be. In both the Chinese and Japanese models, programs which had the external form of a simulation model were used in preparing data.

Display Programs

G7 can be used for viewing and graphing results of all matrices and vectors. The Compare program, activated by Model | Tables from the G7 main menu, in addition to the features familiar from macromodeling, can display of all the flows in a row of the input-output table, showing the flow in each (sufficiently important) cell in selected years and the growth rates of these flows.

A Note on Notation

In the following chapters, examples of commands will be frequently given. The examples will be of two types, generic and specific. In the generic examples, items inclosed in< > are mandatory; items enclosed in [ ] are optional. Items not enclosed in either must appear exactly as written; the words inside < > and [ ] are intended to describe a class of entries. In no case would the < > or [ ] appear on the actual command. Thus, the DOS copy command might be described as

copy <source file> [destination file]

Names of files will be in all capitals. Names of programs will have the first letter capitalized. Names of commands used in the programs are surrounded by double quotes. Where special keys need to be typed, such as the arrow keys or other position keys, they will be enclosed in brackets. For example, holding down the control key and typing the page up key is indicated as "[Ctrl]-[PgUp]. We shall frequently use ... to indicate that lines from an example have been omitted because they are not necessary for the point being discussed. Most of the examples come from the 33-sector Mudan model of China. This model is available on request, or if you have contact with one of the Inforum partners, they may also make a version of the code of an existing Interdyme model available to you.

Installing and Checking the Software by Running Slimdyme

A skeleton model based on Chinese data comprises the distribution files for Interdyme. This model contains only the output and price solution, and a simple imports equation. This simple model, called "Slimdyme", should be installed and compiled, and run first to make sure that everything is working correctly. The Slimdyme model file contains comments pointing out where you need to add the code for the functions of a typical interindustry model, such as final demand equations, employment equations, value added equations, etc. To install, create a directory called \DYME, and unzip the Slimdyme files into that directory. The Slimdyme .zip file will be named DYMEVxxx.ZIP, where the "xxx" indicates the current version of InterDyme. You should also have installed a Borland Builder, which includes a 32-bit Borland C++ compiler called by "bcc32".

Here is a sequence of steps to test that everything is installed properly. It also provides an overview of the operation of a model.

1. From the G7 command line, do

   add initial

The file INITIAL has the following contents.

```
# INITIAL for SLIMDYME
# Create vam file
vamcreate vam.cfg nohist
# Assign as bank b
vam hist b
# Make b the default vam file
dvam b

# Set up a constant A-matrix, using the lint command:
add am.dat
fdates 1980 2000
f mover = 1
index 1980 mover am

# Bring into the VAM file from the Mudan
# G bank data for some vectors.
ba mudan
fadd getout.add   sec33.fad
fadd getfd.add    sec33.fad
fadd getim.add    sec33.fad
fadd getprice.add sec33.fad
fadd getva.add    sec33.fad

# Move 1990 final demand vector, fd,
# forward by 2% per year.

add indexfor.add 1990 .02 fd

# Store the loaded vector, fd.  This is
# necessary only at the end of the work
# with the vam file.
store

# Before you can do Model | VecFixer, you
# must do "close b", but you may first want
# to "show b.fd" or show other vectors or
# matrices.
```

When the program finishes, you may, as indicated by the comments at the end of the *Initial* file, either first look at some of the vectors and matrices, or proceed to build the rest of the model. Before proceeding, however, you must first give the command

```
close b
```

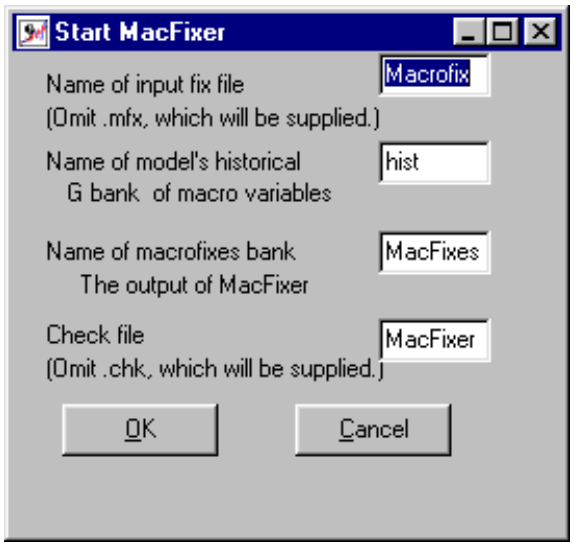
2. Build the macro part of the model by the command

```
Model | IdBuild
```

This step also compiles and links the simulation model.

3. Prepare the macro fixes:

```
Model  |  MacFixer
```

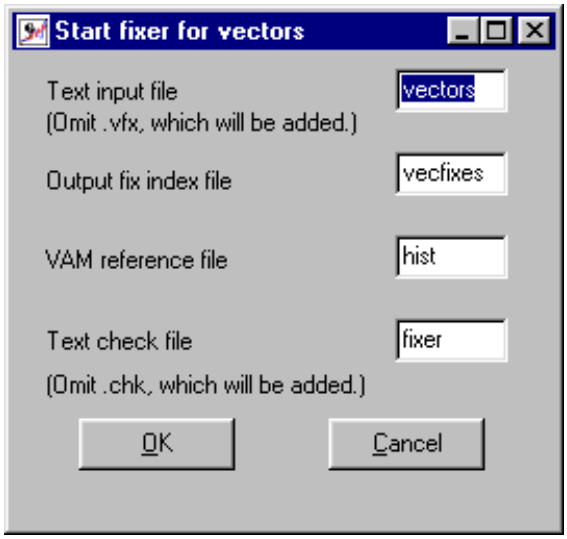


From the information on the form which this command opens, G7 will prepare the file MACFIXER.CFG and then execute the MacFixer program. For slimdyme, you should always accept the default information in the form which opens in this and the following menus.

4. Prepare the vector fixes:

```
Model  |  VecFixer
```

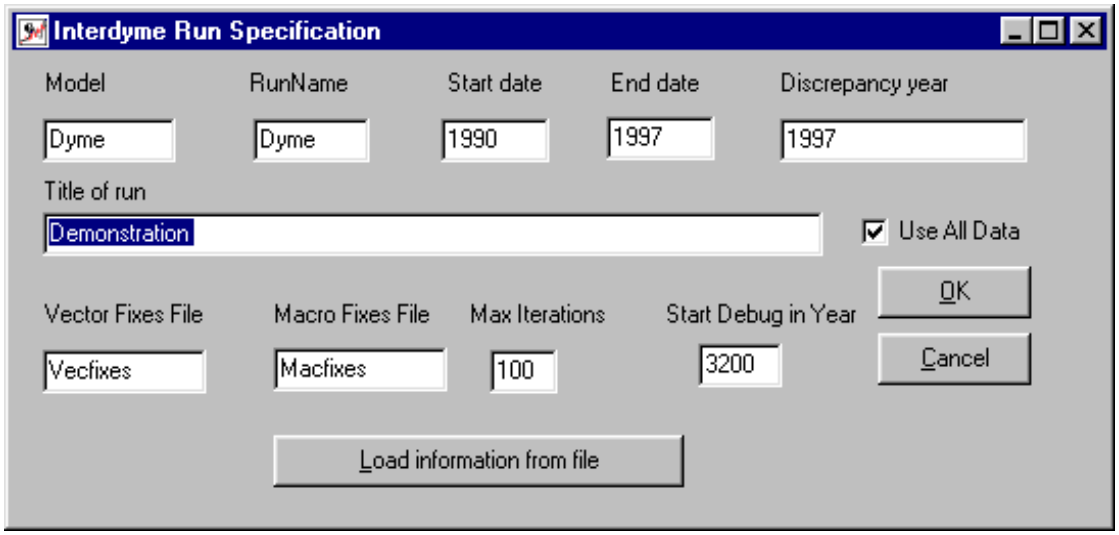
The screen shown below appears.



Preparation of the *text input file*, whose name appears in the top edit box, is described in Chapter 5 on Fixes. The *output fix index file*, whose name appears in the second box, will contain the information for connecting the fixes, which may be given in any arbitrary order in the text input file, to the appropriate vectors or matrices as the simulation program runs. It will automatically be given the extension *.fin*. Thus, in the example shown above, the name of this file would be VECFIXES.FIN. The *VAM reference file* is the file where the Fixer program will look to find information about the historical values of vector variables. *It is also the Vam file into which the values of the fixes will be stored in the vector named* fix. The *text check file* is a file into which the computed values of the fixes will be written. It is a simple text file that can be read, for example, by the G7 editor. Clicking the OK button will cause G7 to prepare the file FIXER.CFG and then execute the Fixer program. Fixer is a console program. When it finishes, close the window in which it runs by clicking the x in the upper right corner.

5. Run the model

```
Model | Run Dyme
```

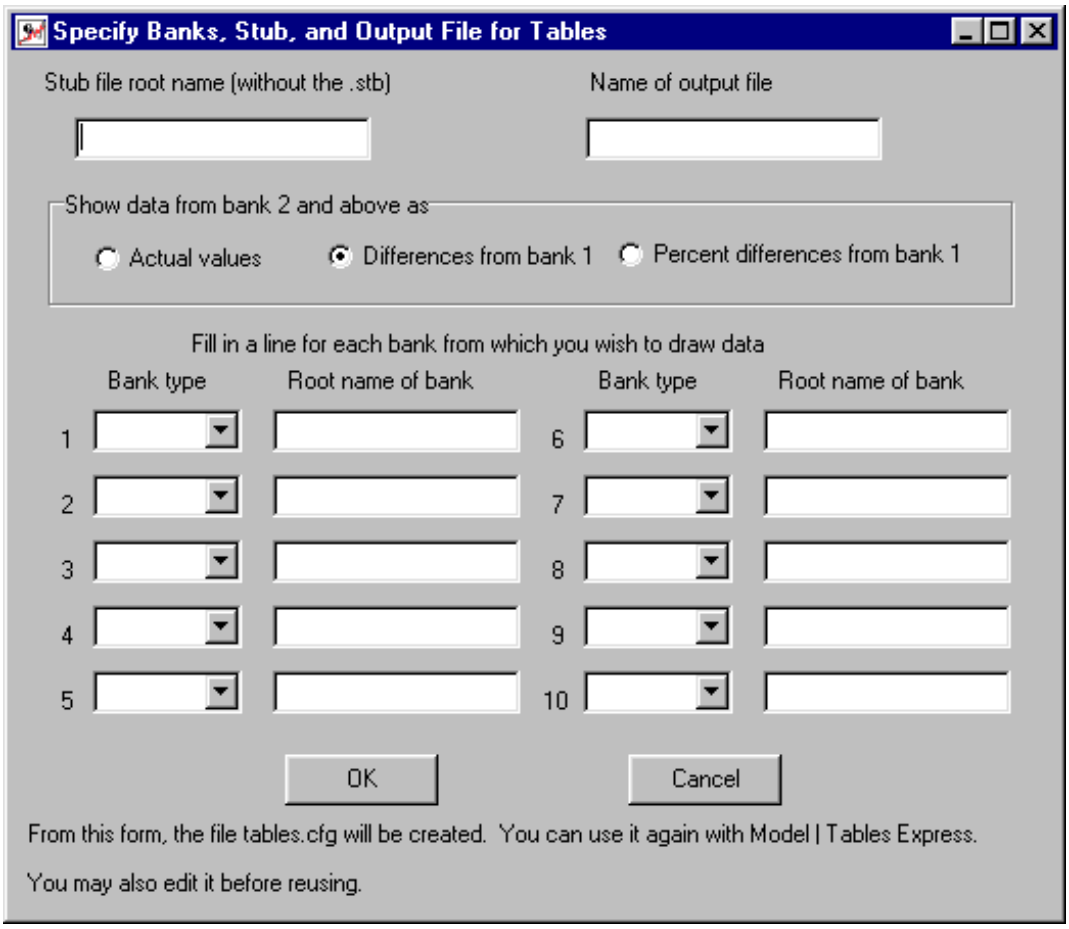


Clicking OK, will create a the file xxx.CFG where xxx is the name of the *Model* you indicate on the form. It will also copy HIST.VAM , HIST.BNK, and HIST.IND files to files with the same extension but with the root name you provide in the *RunName* box on the form.

If you return to running your model after having shut down and restarted G, you can load the information you previous had in this form by clicking the big *Load information from file* button and selecting the .CFG file which you previously created.

6. Execute the Compare program with the command

```
Model | Tables
```



The data in the form which then opens is used to make a control file for the Compare program. This form is already familiar from working with macro models. If you are running Compare again with exactly the same input control, you can use

Model | Tables Express

Once you have verified that Slimdyme is working correctly, you can create a directory for your own model, named whatever you want. To get only the necessary files into that directory, first copy the file GETDYME.BATfrom the \DYME directory to your new directory, and then type "getdyme \dyme" from that directory. You should also look into the GETDYME.BAT file to see what it is copying. Before proceeding much further, you should study the rest of this manual to learn how to build an InterDyme model.

2. WORKING WITH VAM FILES IN G7

The Vam Configuration File

Every vam file begins from a vam configuration file, often called VAM.CFG, which sets out the data content of the model in so far as the data is best thought of as vectors and matrices. This

configuration file contains some very vital information -- namely, the starting date and ending dates for all the vam file -- the range of years over which the model can be run or graphs and tables made. Then, for each vector or matrix in the model, there is a line containing

its name,
its number of rows,
its number of columns,
the number of lagged values with which a vector can be used in the model,
if a matrix, whether or not it is "packed", so that only non-zero elements are stored,
the file name of the titles for the rows of a vector or matrix,
the file name of the titles for the columns of a matrix.
a # followed by a comment usually explaining the economic nature of the variable and possibly the name of the file with historical data.

The format is free. Here is part of the VAM.CFG for the Mudan model of China:

```
#         Matrices and Vectors of the Mudan Model
#
1980 2010 # Starting and ending years
#
#Name |Number of   |Files of titles of|   Description
#     |row col lag|  rows   cols     |
#
# Matrices
am      33  33   0 sectors.ttl sectors.ttl # the input output matrix
bmcr    33  11   0 sectors.ttl hhrural.ttl # the bridge matrix for cr
bmcu    33  19   0 sectors.ttl hhurban.ttl # the bridge matrix for cu
bmv     33  40   p sectors.ttl bmv.ttl     # the bridge matrix for investment
# final demand vectors
hcr     11   1   0 hhrural.ttl             # Consumption of Rural Hh, Hh sectors
hcu     19   1   0 hhurban.ttl             # Consumption of Urban Hh, Hh sectors
...
out     33   1   3 sectors.ttl             # Output
...
#
# Vectors related to fixes
#
dump     33  1  0 sectors.ttl        # Discrepancy of Fixed Outputs
pdump    33  1  0 sectors.ttl        # Discrepancy in prices
fix     100  1  0 fix.ttl        # Fixes, to be filled in by Fixer
```

Note that the starting and ending dates do not control when a particular run of the model starts or stops, but define the range of the vam files. The model in the example cannot start earlier than 1980 or run past 2010, but it certainly does not have to run over the whole span on each simulation. In this example, the model will have a vector named "hcr" which will have 11 rows and 1 column, as indicated by the first two numbers on the line; only current year values of this vector are needed in computing the results of the model in the present year, so the number of lags used, the third number on the line, is 0. In the example above, only the “out” vector requires lagged values. Note the ‘p’ in this position of the “bmv” matrix. This ‘p’ marks a matrix that is to be "packed" so that only non-zero items are stored. (This is not presently actually the case in the Mudan model.) The next item on the line is the name of the file with the sector names to be used for the spreadsheet-like displays of the vector. Everything following the # is a comment.

Names of vectors may contain up to 16 letters or numbers and may contain the underscore mark, "_". They *must not*, however, end in a number. This restriction is necessary because it is sometimes necessary to use the sector number as a suffix to the vector name and to convert between the suffix and subscript forms of the name. For example, we have to be able to recognize that pce[23] and pce23 are the same series. If we had a vector named g2, then g2[3] would convert to g23, and g23 would convert back to g[23], which is wrong. So no numbers at the end of vector names, please!

Please note that C, unlike Fortran, is case sensitive; Q is not the same variable as q.

Creating, Assigning, Defaulting, and Closing a Vam File

To create a vam file from a vam configuration file the command in G7 is

vamcreate <vam configuration file> <vam file>

For example, to create the vam file hist.vam from the configuration file vam.cfg, the command is

vamcreate vam.cfg hist

The "vamcreate" command may be abbreviated to "vamcr," thus:

vamcr vam.cfg hist

At this point, the newly created vam file has zeroes for all of its data. The rest of this chapter deals with how to put data into it.

The first step is to assign it as a bank. The command is

vam <filename> <letter name of bank>

For example,

vam hist b

will assign hist.vam as bank b. Letters *a* through *v* may be used to designate banks. However, it is generally a good practice to leave *a* as the G bank which was initially assigned.

In order not to have to continually repeat the bank letter, most commands for working with vam files use the default vam file. It is specified by the dvam command

dvam <letter name of bank>

For example

dvam b

A vam file must already be assigned as a bank before it can be made the default. However, if several vam files are assigned, the default can be switched from one to another as often as needed.
To review, the commands

vamcr vam.cfg hist
vam hist b
dvam b

will create an all-zero vam file as specified by the vam.cfg file, assign this file as bank b, and make it the default vam file.

Assigning a vam file to G7 opens the file for writing as well as reading. The operating system insures that no two programs can ever have the same file open for writing. Hence, if after looking at model results in the file dyme.vam, we decide to run the model again, we must first close dyme.vam with the G7 *close* command. Thus the sequence might be

```
...
vam dyme c
gr c.out1
...
close  c
```

and we are then free to run the model again and write to the dyme.vam file.

Commands for Introducing Data

There are a number of ways of reading ASCII data into G. Some put individual time series into the G7 workspace bank; others put these series into individual elements of vectors or matrices in the default vam file. Another puts whole vectors into the default vam file. Finally, there are commands for introducing whole matrices into the vam file.

Individual series

There are six commands for introducing time series data from text files, so as to provide various combinations of:

data format
destination (G's regular workspace of the default vam file.)
action (initial introduction or update)

The commands are

data	vdata
update	vupdate
matdata	
matupdate	

All of the commands in the left column put data into G's workspace. The "vdata" and "vupdate" work exactly like "data" and "update" except that the series read goes into the default vam file instead. The last, vmatdata, is used to introduce a time series of a single vector or a number of vectors for a single year. Commands for introducing matrices are covered below.

Because introducing new data was not required in Part I of *The Craft of Economic Modeling*, all these commands will be explained here. The minimal abbreviation of the each command is shown under it .

The basic command for getting a time series into G's workspace bank is the *data* command.

data <series_name>

Introduces data into the workspace data bank. There are 2 forms the command can take.

Form 1:

```
data out1
   1990   34.0  56.8  44.5  55.6  45.2
   1995   39.3  41.2  43.9  47.0 ;
```

The first number on each line is the date of the first observation on that line. There may be any number of observations on the line; the format is free.. Input is terminated by a ";". The ";" may be omitted in an "add" file, except on the last line of the file.

Form 2:

```
data out1 1990
34.0  56.8  44.5  55.6  45.2
39.3  41.2  43.9  47.0 ;
```

The date of the first observation is given on the command line immediately after the series name. No dates appear on subsequent lines.

Form 1 is easiest if the data is being entered by hand. Form 2 is easier if the data is generated by a program.

Input data may contain floating point numbers in exponential form. Thus, the number 3 may be represented as 3.0, 3.0E+00, .300E+01, or 30E-01. Only E, not e, is recognized in this context. Any number of spaces between data observations is allowed.

A missing value may be represented by a single question mark ?. Therefore

```
data sales
    83.1  34.0 ? 44.5 55.6
```

indicates that the sales for 83.2 quarter is missing.

vdata

This command works just like *data* except that the series introduced goes to the default vam file. Recall that if *fd* is a vector in the vam file, fd23 will be the 23rd element of it. If *am* is a matrix, am12.14 is the element in row 12, column 14.

update <series_name>
up

Works like the data command (Form 1) but updates an existing series. The data following the command contain only the data points to be changed. The updated series is placed in the workspace only, not the assigned bank.

vupdate
vup

This works just like update except that the introduces series goes into the default vam file.

monup <series_name>
mup

This is used for updating a quarterly series with monthly data. Example:

```
mup rtb
1993.3  9.120 9.390 9.050   8.710 8.710 8.960
1994.1  8.930 9.030 9.440;
```

"rtb" is a quarterly series being updated with monthly data. 9.120 is the value of rtb in July 1993; 9.390 is the value in August 1993, etc.. Note that quarterly dates are the first numbers on each line and three monthly observations must be present for each quarter. There is presently no corresponding command to put data into the vam file.

matdat [date]

Brings data into G7 prepared by a spreadsheet program. There are two major forms in which the series can be arranged. The series can be arranged either in vertical columns with the name of the series at the top of the column, or in horizontal rows with the names appear on the same line of the matdat command. Up to 20 columns of numbers occupying up to 160 characters per line may be given. The two forms of the command are:

Form 1:

```
matdat
          gnp     c       cd      cnd     cs
1981.1    1513    950     146     359     445
1981.2    1512    949     140     361     448
1981.3    1522    956     143     361     450 ;
```

Dates appear as the first number on each line. Here 1513 is the value of "gnp" in 1981.1, 1522 is the value in 1981.2, etc.

or

```
matdat 1981.1
gnp     c       cd      cnd     cs
1513    950     146     359     445
1512    949     140     361     448
```

```
1522    956     143     361     450 ;
```

The date of the first observation appears on the command line, and no dates appear on subsequent lines. Use a semicolon to end the data. (*You must have a semi-colon on the last line of data for either form*.)

Form 2:

```
matdat  gnp out(1-4) 1981.1
1513    1512    1522
950     949     956
359     140     143
359     361     361
445     448     450;
```

Here, series names are given immediately after "matdat" with starting date given as the last argument on the "matdat" line. The first series reads from the first line after the "matdat", the second series reads from the second line, and so on. Group definitions, as explained below, are allowed here.

matup [date]

Uses the same format as the "matdat" command, but updates series already in the bank.

To use the matdat or matup commands effectively, create an "add" file containing the data or matdat statements necessary to bring in the data. For example, to transfer data from a spreadsheet where the data already exist in columns, use the following steps.

1. Print the spreadsheet data to a file. Add the necessary G7 data commands to the file just created using the G7 editor. (Alternatively, include the commands in the spreadsheet before you print the range.) If the column data do not include dates, use Form 2 of the matdat command.

2. From the G7 command box, use the "add" command to bring in the data stored in the file created by step 1.

If you wish to make a new G7 bank, say mybank, from your data, then you should precede step 1 by the *zap* command to get an empty workspace, then do steps 1 and 2, and then click File | DOS and do

```
copy ws.ind  mybank.ind
copy ws.bnk  mybank.bnk
exit
```

You can then assign mybank as, say, bank *d* by

```
bank mybank d
```

When adding long data files to G, two commands can considerably accelerate processing the data. Include these commands as part of your add file to introduce data, but only after you are sure the add file is trouble free.

addtype n
addt

Normally, G7 writes to the screen all of the input file when an "add" is underway. Turning screen writing off with this command speeds up the operation greatly. Turn screen writing back on again at the end with

addt y

wrindex n
wri

Normally G7 writes the entire index file of the workspace as each variable is added. When many series are added, this feature lengthens processing time. To defer writing the index until the entire add file is processed,"wri n" turns off writing and "wri y" turns it on again and writes the index. A "quit" with writing off will cause the index to be written before quitting, so there is no danger of losing everything by forgetting "wri y".

Entire Vectors

vmatdata

The “vmatdata” command generalizes the matdat command for putting hole vectors into the default vam file. It can be used to bring in data in a variety of formats commonly used by statistical agencies in releasing input-output tables. It can read a file which shows any one of the following:

one vector in columns for several years
one vector in rows for several years
several vectors in columns for one year
several vectors in rows for one year

The vmatdat command requires at least two lines and three if a format for reading is specified. The form of the first line is always the same, as is the form of the third line, if present. The second line has two different forms, one if several vectors are being read for one year and another if data for one vector is being read for several years. The form of the command is

vmatdat <r|c> <nv> <nyrs> <first> <last> [skip]
<year> <vec1> <vec2> ... <vecnv>
-- OR --
<vec> <year1> <year2> ... <yearnyrs>
[form]

where

r|c is 'r' or 'c' according to whether the vectors are rows or columns.
nv is the number of vectors.
nyrs is the number of years.
first is the position in the vector in the vam file of the first item in the data which follows.
last is the position in the vector in the vam file of the last item in the data which follows.
skip is the number of spaces on each line which should be skipped before beginning to read data. *If no value for skip is provided, then a form line (explained below) will be expected as the third line. If no spaces are to be skipped and no form line provided, then give skip a value of 0.*
year is the year of the vectors or the first year if nyrs > 1.
vec1 is the name of the first vector, vec2 is the name of the second vector, etc.

form If no value for skip is given, then this form line must be provided. It should have the letter r in every position which is to be read in the following table. This device allows direct reading of tables that have columns of data separated by | or other symbols or which have subtotals which are not to be read. A typical form line and a matching data line are:

```
form:                rrrrrrrrrrrrrrrrrrr         rrrrr      rrrr
data:Agriculture    | 3450  3718  249   0  | 6780|  8102 |**| 1598
```

Columns which do not have an *r* are simply totally ignored. Consequently, they cannot be used to separate data fields. Thus, reading the following

```
   rrrrrrrr       rrrr
         851       2971
```

would produce one number, 8512971, not two numbers, 851 and 2971.

The first version of the second line is used if there are several vectors for one year; the second is used when giving data for one vector in several years. In both cases, the item of which there is only one is specified first and then the items of which there are several.

Any line beginning with a # is skipped completely.
For example, the lines

```
# Example of several vectors in columns for one year:
vmatdat c 6 1 1 57 3
1986  cons  gov      inv   stk   exp   imp
1     21    0        2           -3          6           7
...      ...          ...         ...         ...         ...          ...
57    38    401      0           0           17          15
```

would read the 6 columns of data into positions 1 through 57 of the named vectors (cons, gov, ...) for the year 1986. Positions 1 - 3 of each line, which contain the sector number for easy visual reference, are skipped.

Here is another example which reads the vectors as rows, as may often be the case when reading printed tables of primary inputs.

```
# Example of several vectors in rows for one year:
vmatdat  r 6 1 1 6  16
1986 labinc propinc deprec inter profit tax
#              1           2           3           4           5           6
labor income   7303    21   368  1203  1302   820
proprietor inc 1215     9   100   107   303    92
depreciation    950     3    82   104   121    73
interest income  845    7    83    54    95    52
profits           50     3    25   217   337    38
indirect taxes   121     1    32   178   294    29
```

Note the use of the line beginning with a # to provide labels for the columns.

The second form of the second line is illustrated by the following example, which reads one vector, cons, for five years.

```
# Example of a single vector in columns for several years:
vmatdat c 1 5  1 57 2
cons 1985 1986 1987 1988 1989 1990
 1   31.8 37.2 32.5 38.7 41.2 43.1
....
57   1.2  1.7  1.8    2.0  2.1  2.2
```

Finally, here is an example which reads a single vector in columns for several years.

```
# Example of reading a single vector in columns for many years.
vmatdat r 1 28 1 10  12
demog 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999
#  Date  ncent south  west college  twoy  fs1   fs2   fs5   head1 head3
 89.000  0.243 0.345 0.208  0.220  0.469 0.243 0.323 0.106  0.274 0.350
 90.000  0.241 0.346 0.209  0.222  0.476 0.250 0.320 0.105  0.267 0.350
 91.000  0.240 0.347 0.211  0.224  0.482 0.247 0.320 0.105  0.263 0.348
 92.000  0.238 0.348 0.212  0.226  0.487 0.245 0.319 0.104  0.260 0.346
 93.000  0.237 0.349 0.213  0.227  0.493 0.244 0.319 0.102  0.257 0.345
 94.000  0.236 0.349 0.215  0.227  0.495 0.243 0.321 0.106  0.257 0.340
 95.000  0.233 0.351 0.218  0.228  0.507 0.252 0.320 0.102  0.257 0.343
 96.000  0.233 0.351 0.219  0.229  0.509 0.253 0.320 0.101  0.257 0.342
 97.000  0.232 0.352 0.219  0.231  0.511 0.254 0.321 0.099  0.256 0.342
 98.000  0.232 0.353 0.219  0.232  0.513 0.256 0.321 0.098  0.256 0.342
 99.000  0.231 0.354 0.220  0.233  0.516 0.257 0.322 0.096  0.256 0.341
```

It must be emphasized that, precisely because it can read in free format, the “vmatdat” command cannot interpret blanks as zero entries. There must be a numerical value for all cells in the tables, especially the 0's.

The flexibility of “vmatdat” often makes it possible to read in final demand columns or primary inputs as they appear in printed tables or on the diskettes provided by statistical offices.

For large models, it is convenient to be able to read "folded" vectors into the default vam file. This is the function of the fvread command. The usage is:

fvread <vector_name> <year> <skip>

The <skip> is the number of columns to skip at the beginning of each line. There must be present as many elements as are in the vector, and they must be in order.

Example:

```
fvread fd 1997 10
finaldem 1   23 36 83 92 32
finaldem 6    8 23 73 80 75
```

This example will put the vector (23, 36, 83, 92, 32, 8, 23, 73, 80, 75) int the fd vector of the default vam file for the year 1997.

Whole Matrices

There are four commands to introduce into the default vam file a matrix from an ASCII file. The first two use different input formats to put matrices into the vam file. The other two put data into packed matrix files, files from which the cells which are zero in all years have been eliminated.

The first, and most used, command is *matin*. It has the form

matin <matrix_name> <year> <firstrow> <lastrow> <firstcol> <lastcol> [skip] [form]

There should then follow a rectangular array of numbers matching the <firstrow>, <lastrow>, <firstcol>, and <lastcol> parameters of the command. As in vmatdata, the optional "skip" parameter is the number of spaces to be skipped in reading each line. The skip parameter and the form line work together exactly as described above for vmatdat: the absence of a skip indicates the presence of a form line. A ‘#’ in the first space of a line means to skip the whole line. Use of "matin" does not affect entries outside the specified area, so it can be used to update a matrix as well as to introduce it originally.

(It is also possible to introduce a matrix that is in a Windows spread sheet by a “Copy” and “Paste” operation into the window created by the *show* command described below. While this route may be a quick expedient, if the vam file is changed and the process has to be repeated, it quickly loses its appeal.)

The second command for reading matrices is solely for conversion of models previously built with the Slimforp program. If you are not converting a model from Slimforp, skip this command. Its form is

matin5 <matrix_name> <year>[<width> <decs> <IndexWidth>]

and is followed by a matrix in the "punch5" format used in Slimforp. The end of the matrix is signaled either by a ';' in the first space on a line or by the end of the file. For those not familiar with the "punch5" format, it is a fixed length format, with 5 matrix entries to a line, each entry of the form <row> <col> <number>. One possible format in Fortran would be (5X,5(2I3,F9.5)). In this case, width = 9, decs = 5, and IndexWidth = 3. Here is a sample from the beginning of an A-matrix:

```
matin5  am 1977 9 5 3
# A-matrix for 1977. 83 rows and 83 columns.
AM     1  1 0.245790  1  4 0.000220  1  8 0.000410  1  9 0.281300  1 10 0.058360
AM     1 11 0.002070  1 12 0.005680  1 13 0.000380  1 15 0.001700  1 16 0.003060
AM     1 22 0.089770  1 24 0.000070  1 48 0.001210  1 49 0.000130  1 50 0.000030
```

Packed Matrices

Finally, there are two special commands for introducing data for a "packed" matrix. Packed matrices become important in models with many sectors, in which the matrices -- especially the various bridge matrices -- often become very sparse, that is, they have relatively few non-zero elements. Interdyme has the ability to carry the matrices in a packed form in which only the non-zero elements are stored. In most places in G, the packed matrix can be used interchangeably with the "full" matrix. For example, in the "index", "lint", and "show" commands described below, either full or packed matrix names can be used without having to remember which kind of matrix is being used. For the input of data, however, there is a special format for packed matrices. Moreover, packed matrices are not included in the vam file. Only the file name of the packed matrix is in the vam file; the actual data is in a special binary file for which we use the extension .PMX. This separation of the .PMX from the vam file was done because (a) the .PMX files are fairly big and (b) they often do not change at all between various alternative runs of the model. Thus we might have five runs of the model, each with its own vam file but all with the same pmx file. The savings in disk space over having five copies of the .PMX files could be substantial.

The first command for introducing data into packed matrices assumes that the elimination of the zero elements has already been done, as is likely to be the case if the matrix comes from Inforum programs for updating a matrix. The second command reads the matrix in full rectangular form and eliminates the zeroes itself. The result is the same in either case. The first command is:

pmatin <matrix_name> <packed_matrix_filename>

where <matrix_name> is the name of a matrix in the vam.cfg file which must have the letter p in the "lags" position. The <packed_matrix_filename> is name the DOS name of the .PMX file. (The .PMX should be included in the name.) The format of the data which follows is then:

<year>
<row> <num_non-zero_elements>
<col1> <element1> <col2> <element2> ...

For example

```
pmatin bm bm.pmx
1985
1  5  # row 1 has 5 non-zero elements
1  .656  7 .352  11 .038   23 .019  27 .218
3  2  # row 3 has 2 non-zero elements
7  .098  51 .923
etc.
```

If the matrix is available for several years, the data for all years should be introduced with only one “pmatin” command, with each year's data preceded by the number of the year. It is essential that the year numbers not be abbreviated; use 1987 please, not 87. If a cell is non-zero in any year for which there is data, the packed matrix will have a place reserved for it in all years.

An alternative way of introducing data into packed matrices is the “pmatin1" command. The format is just like that of the “pmatin” command, except that the data should be arranged in rectangular rows and columns like the “matin” command. For example, if you wish to treat as a packed matrix a 3 x 3 matrix known as B in the vam file, you can introduce it with the “pmatin1" command as follows:

```
pmatin1 B B.pmx
1995
 1  0  0
 5  0  1
 6  2  0
1996
 1.5 0  0
 6   0  2
 7   3  1
```

Note that in the above example, only those cells that are zero in all years will be left out of the packed matrix. Cell (3,3) will actually be stored, since it has a non-zero value in 1996. As with the "pmatin" command, you should put the data for all years you want to read in the same file, and use only one "pmatin1" command per matrix.

Although packed matrix files often do not change between runs, they may. If, for example, we wished to study the effects of changes in the A matrix and the A matrix was packed, then we would need two different .pmx files. If the original was called AM.PMX, then to create the alternative, say, AMALT.PMX, we would go to the DOS prompt via File | DOS and do

```
copy am.pmx amalt.pmx
copy hist.vam vamalt.vam
```

Then from G’s command box do

```
vam vamalt b
dvam b
pmfile am amalt.pmx
```

This *pmfile* command will both assign AMALT.PMX to be used whenever the am matrix is referred and put AMALT.PMX into AMALT.VAM as the name of the file to be used for the packed am matrix. The general form of the "pmfile" command is:

pmfile <matrix_name> <packed_matrix_filename>

Loading the Vam File from G7 banks

It is also possible to introduce data from G7 banks. Just as the *f* command will form a variable and store it in G's workspace, the *vf* command forms a variable and puts it into the default vam file. The format is

vf <name> = <expression>

Example:
vf out1 = out1
This seemingly tautological example actually does something useful. It will look in G's workspace bank for out1 and if it fails to find it, it will look in the bank assigned as *a*. Let us say it finds out1 there. It will then copy this series over to the default vam file. The right side of the vf command can be any valid G7 expression, including the various @ functions, such as @log, @exp, @pos, @ifpos, and @csum. To specify that a variable on the right should come from the bank assigned as *b*, put b. in front of the variable name. The *vf* command works over the range of dates specified by the *fdates* command.

fdates <fdate1> [fdate2]

Specify dates which will be used as the range of action of the *f, vf, vc, index, ctrl* and other commands described below. The default values of the fdates are the beginning and ending dates of the vam file, respectively.

Commands for the Display of Data

All of the G7 commands for the display of data all work for data in any assigned vam file; it is just necessary to prefix the bank's letter designation followed by a period to the variable name. Thus

vam dyme b
ti Output, Export, and Import of Agriculture
gr b.out1 b.ex1 b.im1

will graph the first element of the out, ex, and im vectors from the vam file assigned as bank b.

For vam files, however, there is another command which shows vectors and matrices as if one were in a spreadsheet program. This is the *show* command. For vectors it format is just

show <bank letter>.<vector_name>

The values of the vector in successive years will appear as columns; dates run across the top and sector numbers down the side. The Options menu item allows the user to control the number of decimal places, fonts, and colors of the display. The display can be copied to the clipboard in the usual way for Windows programs. The contents of the clipboard can then be copied into a spreadsheet such as Lotus 1-2-3 or into a table in a word processor. On the Copy command, you get to specify how much you want copied — just the numbers or also the frame.

It is also possible to go in the other direction, from the spread sheet into the show window and thus into the vam file.

For matrices, the *show* command has a somewhat different format

show <bank letter>**.**<matrix_name> <view> <line>

The "view" argument must be one of the following:

r for row
c for column
y for year.

If view is 'r', then <line> is the number of the row to be displayed. (A row is displayed as if it were a column.) If view is 'c', then <line> is the number of the column to be displayed. If view is 'y', <then> line is the year number. Examples:

```
show am r 5
show am c 7
show am y 1997
```

Cutting and pasting works as with vectors.

load <vector_name>

When G7 is working on a vector in a vam file, it pulls the whole time series for the vector into a sort of vector workspace. Usually, this is done implicitly by simply referring to the vector or one of its elements. The *load* command enables the user to do this loading explicitly. It is used principally in connection with the *index* command described below. Example:

"load pce".

store

Store the currently loaded vector back to the vam file with the modifications that have been made. This *store* is automatic when a new *load* or implicit *load* is encountered. When a *quit* is encountered with an unstored loaded vector, the program asks whether it should store that vector. Answering "no" is a chance to escape if you know you have made a mistake and do not wish to mess up your vam file. On the other hand, giving the *store* before the *quit* is a way to avoid this question. Long .add files will, therefore, frequently end with a *store*.

Arguments and Looping in G

In working with multisectoral models, it is frequently desirable to be able to do something comparable to what subscripting does in mathematics or *for* or *do* loops do in programing. The system for doing so begins with the *add* command familiar from *The Craft* Part 1. It has a capacity not employed there to accept *arguments*. The places where the arguments are to be substituted are indicated by %1, %2, etc. For example if the file OUTEX.ADD

```
title %1 %2
gr out%1 exports%1
```

is executed with the command

```
add outex.add  2  "Animal husbandry"
```

the effect will be exactly as if we had executed

```
ti 2 Animal husbandry
gr out2 exports2
```

G7 will replace the %1, %2, ..., %9 with the arguments on the command line.

The next step is *fadd* command which combines the *add* command with a file containing the arguments. For example, we might have a SECTORS.ARG command such as

```
1   "Field crops"
2   "Animal husbandry"
3   "Agricultural services"
...
```

and so on for the rest of the sectors of the model. Then the G7 command

```
fadd  outex.add sectors.arg
```

will graph the outputs and exports for each sector of the model.

While *fadd* is the most versatile of the looping commands, it can become bothersome to have numerous .add and .arg files. If the arguments are just integers, the .arg file can be replaced by a *group* of integers. For example, if the PC.ADD file is

```
ti Percent change in output of sector %1
f pc = 100.*(out%1 - out%1[1])/out%1[1]
gr pc
```

we can do

```
add pc.add (1-80)
```

to get graphs, one at a time, for 80 industries. The specification of groups can be more complicated. (1 - 10 (3-7) 5) would be the integers 1, 2, 5, 8, 9, 10. The 1 - 10 directs the program to start with all the integers from 1 to 10; the (3 - 7) removes those from 3 to 7, and the 5 adds back in the 5. The - indicates a range; parenthesis indicates removal from the set. We could do

```
add pc.add (1- 50 (31- 38) 35 55 - 63 69)
```

For what sectors would we get graphs?

Having many separate “child” .add files can sometimes make the “parent” file unclear and clutter up a directory long after the parent has been deleted. A solution to this problem is the use of the *do* statement. Its general form is

do { <G7 commands> } <group>

For example, we could have

```
do {
    ti Percent change in output of sector %1
    f pc = 100.*(out%1 - out%1[1])/out%1[1]
    gr pc } (1 - 10 (3-7) 5 )
```

The group of integers must be specified on the same line with the closing brace.

Commands for Sweeping Modification and Projection of Data

Vector Calculations

vc <vector_name> = <expression>

The *vc*, or vector calculation, command evaluates the expression on the right and puts the results into the named vector. Only a few matrix and vector operations are implemented with *vc*. It can add or subtract any number of vectors and multiply a matrix times a vector. It cannot yet add or subtract matrices; parentheses are not yet supported. However, scalar values (either constants or G7 bank variables) can be used to premultiply or postmultiply a vector or matrix. Also, a scalar value can appear all alone on the right hand side of a *vc* equation, which indicates that the entire vector will be set equal to that scalar value. Between a matrix and a vector, an * means matrix-vector multiplication. Between two vectors, the * means element-by-element multiplication. Between a vector and a matrix, the / means multiplication by the transpose of the matrix or vector on the left. Between two vectors, the / means element-by-element division. The *vc* command is performed only over the interval defined by the current f*dates* command Example: If `am` and `cm` are matrices and `out, pdm, qcu`, and `cprices` are vectors, we could write

```
vc out = am*out + out    # matrix multiplication, vector addition
vc qcu = out*pdm         # element by element vector multiplication
vc out = qcu/pdm         # element by element vector division
vc cprices = pdm/cm      # This is actually pdm’cm.
```

The <vector_name> must be a valid vector name in the vam file, as must all names in the expression. A more extensive *vc* command is planned.

Groups of Sector Numbers

We have already seen how, with the *add* and *do* commands, it was useful to be able to specify a group of integers as arguments. In the *lint*, *index*, and *ctrl* commands which we are about to explain, this concept of a *group* of integers is carried further. For use in these commands, the groupis

specified first and then used in the commands. G7 has a dynamic group that can be specified and respecified over and over as necessary. It can also use the static groups defined by the Fixer program described below.

The dynamic group is defined by the command:

group <sector numbers>

Define the content of the dynamic group. The sector numbers are specified as in the *add* command. Example:

```
group 19-25 (20 22)
```

The name of this dynamic group is " : ". The names and content of the static groups defined in fixer are preceded by a : in the following commands.

listgroups

List the names of all the groups currently in the GROUPS.BIN file.

glist <name>

List the sectors in a group. The <name> specifies the group name. After the above *group* command, the command

```
glist :
```

would give the answer

```
19 21 23 24 25
```

If the "listgroups" command gives the answer

```
Ag Min Mfg Trans Trade Util Serv
```

then

glist Ag

might, for example, give an answer like

```
1  2  3  4
```

Linear Interpolation of Vectors and Matrices

Linear interpolation of missing values in a vector is done by the *lint* command.

lint <names>

Replaces missing values in the time series of the series by linearly interpolated values. Zeroes before the first or after the last observation are not replaced. This command applies only to series in the vam file. Groups may be used in the "names" field to refer to sectors of the currently loaded vector. Examples:

```
lint pce1
```

This example loads the pce vector and then fills in the missing values in sector 1 by linear interpolation. We could similarly fill in the values for all the sectors in the present dynamic group by

```
load pce
lint :
```

or all the values for the static group `Ag` by

```
load pce
lint :Ag
```

Entire matrices may be interpolated with one command. For example

```
lint am
```

will interpolate the entire `am` matrix. This interpolation also works for packed matrices.

The *lint* command works on the entire range of the vam file without regard to fdates.

Moving Vectors and Matrices by Indexing

A quick way to fill in values of a vector so that they all grow at the same rate is the use of the *index* command. It is used in conjunction with a guide series, a previously established single-variable time series, whose growth rates will be applied to the elements of the vector. It acts over the range of the currently specified *fdates.* The form is

index <base date> <guide> <vector name>

For example, if “years” is the name of a series of the numbers of years — 1980, 1981, etc. — then to make all elements of the vector “pce” grow by 2.0 percent per year from 1999 to 2010, the following commands can be used:

```
f  g02 = @exp(0.02*(years - 1980))
index  1999  g02  pce
```

The use of groups is allowed in the names of series to be affected. For example, to move only the sectors 1,7, and 9 of `pce,` we could -- recalling that : is the name of the dynamic group -- do

```
group 1 7 9
load pce
index  1999 g02  :
```

To move just the sectors in the static group Mfg, the command would be

```
load pce
index 1999 g02 :Mfg
```

With a specific vector+sector name like "exp2", command would be

```
index 1999 g02 exp2
```

to store the current loaded vector, load the exp vector, and move the series exp2 by the index of g02.

A vector or matrix name not followed by a sector number means to move the whole vector or matrix by the index. This device can be used to project a whole input-output matrix, say, am, to be constant, as in this example:

```
fdates 1998 2020
f one = 1
index 1998 one am
```

A matrix name followed without space by a sector number means to move that row of the matrix by the guide. Thus, "am2" means move the second row of the am matrix by the guide.

If <guide> is the name of a vector and < name> is a matrix, then the rows of the matrix are indexed by the corresponding elements of the vector. If an element of the vector is zero in the base year, the corresponding rows of the matrix are unchanged. This feature works for both standard and packed matrices. It can be used to apply across-the-row coefficient change to an input-output matrix. For example, if `movers` is a vector, we can make each row of `the am` matrix follow the index of the corresponding element of `movers` by

```
fdates 1998 2020
index 1998 movers am
```

In applying this sort of across-the-row coefficient change, one usually does *not* want it to apply to the diagonal elements, for they have little to do with the technological changes affecting other coefficients. To avoid changing them, the following two commands are convenient.

diagextract <matrix_name> <vector_name>

Extract diagonal elements from the matrix and put them into the named vector. The command works for both regular and packed matrices.

diaginsert <matrix_name> <vector_name>

Insert elements in the named vector into the diagonal of the matrix. The diagonal element will remain zero for a packed matrix which has zero coefficient in that position for all years.

This example combines these last three commands

```
fdates 1998 2020
diagextract  am  diags
index 1998 movers am
diaginsert  am diags
```

Controlling Totals of Vectors and Subvectors

In specifying scenarios, it is important to be able to control the total of the projected values of a vector or of certain elements within the vector. The *ctrl* command gives this ability.

ctrl [basedate] <guide> <sectors>

This command imposes the values of the <guide> series as a control total on the named sectors of the currently loaded vector. The control is imposed over the period specified by the current fdates. If no basedate is present -- note that it is an optional element in the command -- the absolute values of the guide series are the control total. If the basedate is present, the guide series will be scaled to the total of the series in that period before being used as a control total. In naming the sectors, the allowable names are illustrated by:

0	for all the sectors in the loaded vector.
:	for the sectors in the present dynamic group
:Mfg	for the sectors in the static Mfg
1 7 9	for the group composed of sectors 1, 7, 9

Matrix Balancing by the RAS Method, Coefficients and Flows

In work with input-output analysis, one frequently needs to balance a table to known row and column totals or to convert flows to coefficients or vice-versa. Here are the commands to do so.

ras <matrix> <row> <col> [year] <r | c>

Balance the named distribution matrix by the rAs method so that <row> = <matrix>*<col> with each column of the matrix having a column sum of 1.0. It does not actually matter whether <row> and <col> are columns or rows, though in the above equation they are to be thought of as columns. If the [year] parameter is missing, the command works over the fdates range. If the sum of the elements of <col> does not equal the sum of the elements of <row>, the vector to govern is specified by the 'r' or 'c' at the end of the command. The other vector will be scaled to have the right total. The rows will then be scaled to get the correct row totals, then the columns scaled to get the correct column totals. Every five iterations, there is a report on the row and column in which the maximum scaling occurs. When the maximum scale factor differs from 1.0 by less than .00002, convergence is declared to have been reached. The last scaling will be of the rows. If convergence does not occur after 100 iterations, the program reports the problem and continues to the next command. Once the balance is achieved, the columns are scaled to sum to 1.0. The reason for this normalizing and how to deal with it if it is not wanted is explained after the next two commands.

coef <matrix> <vec> [year]

Convert the named flow matrix to a coefficient matrix by dividing each column of the matrix by the corresponding element of the named vector. If no year is named, the command works over the fdates range.

flow <matrix> <vec> [year]

Multiply each column of the matrix by the corresponding element of the named vector. If no year is named, the command works over the fdates range. This command is the opposite of coef command.

In our experience, the matrices most frequently in need of balancing are the distribution matrices, such as those that determine how personal consumption expenditure by budget category is to be distributed to industries. For this reason, the ras command, after balancing, automatically insures that the column sums are 1.0. Of course, if the matrix is the intermediate coefficient matrix, one does not want the columns to sum to 1.0. What do we do then? Let us suppose that am is an initial coefficient matrix, q is the vector of outputs, cs is the vector of column sum controls and rs is the vector of row sum controls. Then we would do

```
flow am q
coef am cs
ras am rs cs r
flow am cs
coef am q
```

getsum <matrix> <r|c> <vector>

Get the sum of the rows or columns of a given matrix or vector, for all the years in the Vam file. The first argument must be the matrix or vector for which we want to obtain the sums; the second argument must be 'r' or 'c' (to obtain the sum by rows or by columns); the third argument is the vector where the result is to be stored.

The sum is calculated and stored for all the years, so that we cannot have mixed information in the target vector.

Commands for Writing Data to ASCII Files

It is sometimes necessary to take data from a vam file and output it to an ASCII file. For example, we may have used the Windows cut and paste commands to move data from a spreadsheet into the window of the *show* command, but we want an ASCII version of the file so we will not need to repeat this hand procedure if we rebuild the vam file from scratch. The following commands provide various ways to output matrices and vectors to files that can be easily read back by G7 or, in some cases, by other programs..

pmpunch <filename> <matname> [decs]

The non-zero elements of the matrix <matname> are written into the named file as input for the "pmatin" command. The [decs] argument gives the number of decimal places. The years written are determined by the current values of "fdates". This command is especially useful for converting a rectangular matrix in a vam file into a packed matrix. One writes it out as an ASCII file with this command and then reads it in with the "pmatin" command.

punchvec <filename> <vecname> <startyear> <endyear> [<width> <decs>]
pv

Print out a vector to a file for all sectors, for the years specified. The optional width and decs arguments again specify the field width, and number of decimal places. The output file starts with a number of comment lines, telling the name of the vector, starting and ending years, and format information. Then one comment line gives the years as column headings. Each following line of the file contains the time series for one sector, with the name of the series, followed by the time series of data.

punch5 <filename> <vecname> [<width> <decs><IndexWidth>]
p5

Prints out a matrix or packed matrix to a file in "punch5" format. The years that are written are determined by the current value of fdates. Each year is written with a header that is a "matin5" command, which tells "matin5" also what field width, decs and IndexWidth to use. (The IndexWidth parameter is the width of the row and column numbers in the file. Thus, a file created with "punch5" can be read back into Vam using the "matin5" command. This provides a convenient way to convert packed matrices into normal matrices. The "punch5" format prints 5 non-zero matrix cells to a line, with row number, column number and cell value for each cell.

vp <vector_name> [r|c] [field_width] [decimal_points]
or
vp <matrix_name><view><line> [field_width] [decimal_points]

Write the named matrix to the currently open "save" file. As you can see, this command has a format and options like those of the "show" command. Example:

```
save am.dat
vp am y 2000  9  6
save off
```

Note that the use of save files in conjunction with the "matty" or "type" commands are also useful ways of capturing vam data in ASCII form file. Furthermore, since the Compare program can work with Vam files, you can use the "\gdata" command in that program to make an neat ASCII file printout of vector or matrix time series from a vam file.

Vam File Title and Prompting Commands

commands
com

Should you -- heaven forbid -- have forgotten any of the preceding commands, the command "commands" will show you a list of them.

listvecs
lv

This command shows a list of the vectors and matrices in the model. Another way to see them is to do "ed vam.cfg". This also allows you to see the rows, columns, lags and titles files for each vector and matrix.

vtitle <title>
This command allows you to give the vam file a title. This title is displayed when using Compare.

Vam2vam -- Selective Copying From One Vam File to Another

Vam2vam is a utility program, run at the DOS prompt, to copy selected matrices and vectors from one vam file to another. It often happens that one discovers that one has left out of a vam file some essential matrix or vector. It is easy enough to modify the vam.cfg file and create a new, all-zero vam file with a place for the new matrix or vector. But how can data be copied from the old vam file to the new? Vam2vam is the answer. Or one discovers that a matrix or vector has to be enlarged. How can the data in the old vam file be copied into the new? Vam2vam is the answer. Or a vam file has been used in the preparation of data which has in it various matrices and vectors which were essential for preparing the data but which are not needed in the final model. How can we extract just the final product matrices and vectors into a new vam file for the model? Use Vam2vam.

The program is invoked by the command:

vam2vam <source> <target> <list> <startdate> <stopdate>

where
<source> is the filename, without the .VAM extension, of the source vam file;
<target> is the filename, without the .VAM extension, of the target vam file;
<list> is the name of a file with a list of the names of vectors or matrices to be copied. For example, to copy abc in the source to def in the target and xyz in the source to xyz in the target, the contents of the <list> file would be

abc def
xyz

Note that the default value of the name in the target is the name in the source, so we did not have to repeat xyz on the last line.
<startdate> is the first year whose data is to be copied
<stopdate>is the last year whose data is to be copied.

VamtoG -- Creating a G bank From Series in a Vam File

Like Vam2vam, VamtoG is a convenient way for getting data from a vam file, in this case, into a G7 "ws" type bank. The operation of this program is controlled by a configuration file, named VAMTOG.CFG. A sample of this file is shown below.

```
Root name of Vam File; hist
Root name of destination G bank; imp
Starting Date for data transfer; 1972
Ending Date for data transfer; 1994
Base Year of G databank; 1955
First period covered; 1
Maximum number of observations;60
Data requests ; im
   ex out def fpi fpe
```

In this particular example, the source vam file is HIST.VAM, the destination G bank is IMP.BNK, the data will be transferred from 1972 to 1994. The G bank will have a base year of 1955, starting period of 1, and series vectors of length 60. The vectors "im", "ex", "out", "def", "fpi" and "fpe" will be transferred. Since vam does not yet handle regressions, the Vamtog program is particularly useful for building data banks for G regression, to ensure that the data is exactly the same as what is currently in the vam file.

3. IDBUILD -- MACRO-EQUATION PROCESSOR

Idbuild is an adaptation of the Build program for aggregate model building to building Interindustry Dynamic models. Like Build, it translates the .sav files created by G7 into C++ code, builds a bank of all the scalar variables, and writes several files of C++ code for use in the simulation program. Also like Build, it requires a build.cfg file which specifies the name of the output (or workspace) bank and the initially assigned bank. Also like Build, it requires a Master file, but this file merely lists each .sav to be included in the preceded by the command "iadd". Unlike Build, the Master file should not contain identities or other code. Idbuild is invoked from G7 by the command Model | IdBuild. This command also proceeds to the compilation and linking of the model. (From DOS, IdBuild can be invoked by “idbuild master”. The output files made by Idbuild are:

HIST.BNK HIST.IND	A standard G bank containing all the series used or created by idbuild. (As in Build, variables on the right of "fex" commands are not included.)
HEART.CPP	A compilable C++ program. Each of the "iadd" files becomes a separate, callable function with a name derived from the name of the iadd file from which it was created. It also contains a function (tserin) to read in all the time series from bws and make them accessible everywhere in the forecasting program. Both past and preliminary future values are available at all times.
HEART.H	Prototypes for all the functions in the heart.cpp file. This file makes it possible to call these functions from anywhere in the forecasting program.
TSERIES.INC	A file to be included in the forecasting main module of the forecasting program declare the names of all the variables in HIST.BNK as variables which can be used in the program.
CALLALL.CPP	A program to call all of the functions in HEART.CPP. It is used at the end of each years calculations to set or update rho adjustment factors.

The Master File for Idbuild

We will illustrate the forming of the master file with the Mudan model of China. The file for this model is:

```
bank cmdm
iadd invest.sav
iadd income.sav
iadd finance.sav
iadd pseudo.sav
q
```

Except for PSEUDO.SAV, the various .SAV files contain estimated equations. For example, invest.sav begins

```
title Other State-owned units investment
f sinvest = sibac$ + sirep$
r invn$35 =   -40.346479*intercept +
       0.110044*sinvest
```

The program recognizes PSEUDO.SAV as the name of a file with a special purpose. Namely, it puts into the model's G bank those time series which are not needed in any of the code-image equations but are required elsewhere, perhaps in identities or detached coefficient equations. For Mudan, the beginning and end of the pseudo.sav file are:

```
f trsa = trsa
f rpop = rpop
f upop = upop
...
f rscale = 1.
f uscale = 1.
```

These commands put the variables named trsa, rpop (rural population), and upop (urban population) into the model's G bank. At the end, it initializes the rscale and uscale variables to 1.0 in all years. Any right-hand side legal in G7 would be legal here.

Back in the master file, the final "q" signals the end of input to Idbuild. Idbuild will produce from this command the following heart.cpp file:

```
#include <stdio.h>
#include "constant.h"
#include "matrix.h"
#include "dyme.h"
#include "groups.h"
#include "databank.h"
#include "vamfile.h"
#include "fixbank.h"
#include "dyme.ext"
#include "heart.h"
#include "tseries.ext"
FILE *fmatrix;
int i,j,k,err;
extern int t;
float depend;
/* end of standard prolog */

void investf()
{
/*  Other State-owned units investment */
 sinvest[t]= sibac_[t]+ sirep_[t];
/* invn$35 */ depend =-40.346479+0.110044* sinvest[t];
        invn_35.modify(depend);
...
    }

void incomef()
{
...
    }
```

```
void financef()
{
...
      }
void tserin()
{
sibac_.in("sibac$");
sirep_.in("sirep$");
sinvest.in("sinvest");
...
}
```

Note that all of the "iadd" files have been turned into functions whose names are formed by adding an 'f' to the end of the "iadd" file name. (Any $'s in the file name have been turned to _'s; there are none in the example.) What were "f" commands have become C statements with the exception that the '$' character in variable names has been changed to an '_'. Examples of this change are seen in both the investf() function and the tserin() function. Any "fex" commands (none in the example) have disappeared, but the variable on the left of the "fex" has been created and entered into the data bank. Regression equations appear with the regression coefficients in the code. They calculate a variable, "depend", which is passed to the routine modify(), along with the identification number of the dependent variable. The modify() routine then looks to see if there are any macro variable fixes -- add, multiply, index, growth, skip, or rho adjustment -- on that variable, and then stores the variable with the appropriate modification, if any. At the end of the HEART.CPP file is the tserin() function. It is called at the beginning of a run of the model to read into memory all the time-series variables. Finally, note that it contains all the variables which are in the PSEUDO.SAV file, even though no function was created by this file. This is the way to put into the data bank those variables, such as labfor, which appear in no code-image equation. The name "pseudo" is a keyword to the program; files by any other name create functions.

The HEART.H file, created by idbuild for this example, is:

```
void investf();
void incomef();
void financef();
void exdgf();
```

It simply provides the prototypes required by C++ in any program which uses the functions in the HEART.CPP file.

The TSERIES.INC file is

```
Tseries sibac_, sirep_, sinvest, invn_35, invn_36, d88, d90, invn_37,
rni, rpindex, rpop, rincome_, trsa, trsa_, invn_38, ulfi,
...
uscale;
```

These files are "#included" in the forecasting program to declare that sibac_, sirep_, etc. are objects of the form "Tseries". A "Tseries" is an "object" defined in the forecasting program designed to hold

a time series. One of the things that a Tseries object "knows" how to do is to load itself from the assigned G data bank. Thus, in the tserin() function in the heart.cpp program above, the command

```
ngdpc.in("ngdpc");
```

tells the ngdpc object to read in its data contents from the series called "ngdpc" in the data bank.

The final product of Idbuild is the CALLALL.CPP file, which, for our example, is simply

```
#include "heart.h"
void callall()
{
investf();
incomef();
financef();
exdgf();
}
```

As its name suggests, callall() is simply a program to call all of the functions in the heart.cpp file. Its function is in connection with rho adjustments, as will be seen in the forecasting program.

Combining Vector and Tseries Variables in a Function With Idbuild

When building interindustry macro models one usually needs to integrate the macro and the industry computations. For example, it is often necessary to form a macrovariable as a sum of components of a vector. Conversely, it may be that some sectoral variable is required on the right hand side of a macro equation. When this is the case, the Idbuild command “isvector” is particularly useful. This command indicates to Idbuild that a variable in one of the following save files is to be treated as an element of a vector, and not as a macrovariable, or Tseries variable, which is the default. For example, in the LIFT model of the U.S., the equation for railroad construction uses output of sector 59 (Railroads). Other equations use aggregates of output of many sectors. The included sections of files below show how this is handled. In G, here is part of the regression file, CONSTR.REG:

```
save constr.sav
f outman = @csum(out,9-58)/1000.
f outbus = @csum(out,64,65,72,73,77-80)/1000.
f outtrade = @csum(out,69-71)/1000.
f outmin = @csum(out,2-6)/1000
#================================
ti 15. Railroad Construction
r cst15$ = cstoth, rpoil[2], rcbr[1], out59, doutrail, doutrail[1]
gr *
```

Here is a small MASTER file, which will generate the code for this function only.

```
ba constr
isvector out,emp,pdm,cstk
iadd constr.sav
```

Here is the code for the constrf() function. Note that since the "isvector" command was in effect, Idbuild knows that "out" is a vector. Therefore, it passes "out" as one of the vectors in the argument list to constrf(), it writes out the csum function correctly as a method of type Vector, and it writes "out[59]" on the right hand side of the regression equation instead of "out59[t]".

```
void constrf(Vector& out,Vector& emp,Vector& pdm,Vector& cstk)
{
 outman[t]=out.csum("9-58")/1000.;
 outbus[t]=out.csum("64,65,72,73,77-80")/1000.;
 outtrade[t]=out.csum("69-71")/1000.;
 outmin[t]=out.csum("2-6")/1000;
        .
        .
        .
/*  15. Railroad Construction */
/* cst15_ */ depend =3699.702916+-0.405094* cstoth[t]+5.812924* rpoil[t-2]+
17.332162* rcbr[t-1]+0.020963* out[59]+0.040612* doutrail[t]+
0.117446* doutrail[t-1];
        cst15_.modify(depend);
        .
        .
        .
}
```

At the present time (Interdyme version 2.2), Idbuild doesn't know how to handle lagged values of vector variables. In this case, you can make the vector variable a macrovariable, and include code in your model to copy the macrovariable values to the vector, and vice versa. For example, another regression in the construction equations mentioned above uses construction capital stock of category 16 lagged once ("cstk16[1]"). The way to handle this is as follows. Before opening up the save file in G, first do:

```
f cstk16$ = cstk16
```

Then, include cstk16$[1] on the right hand side of the equation. In the simulation model, remember to fill up the macrovariable with the value of construction capital stock of category 16 before calling the construction function:

```
cstk16_[t] = cstk[16];
constrf(out,emp,pdm,cstk);
```

Note that in addition to the "iadd" command, IdBuild has one more command not found in standard Build. That is the "break" command. Its format is:

break <filename>

After the "break"command, subsequent C++ code will go to the named file, with the extension .CPP appended, rather than to HEART.CPP. The reason for this command is that with a large model, HEART.CPP can become too large, and may fail to compile. Also, it is somtimes more convenient to group the functions written by IdBuild into smaller logically organized files.

Macrovariables can also be used on the right hand side of equations for vector variables, called "detached-coefficient equations". These are described in the next section.

4. DETACHED-COEFFICIENT EQUATIONS

Multisectoral models often employ similar equations for similar functions in different sectors. Thus, consumption functions may be similar in different sectors but, of course, with different coefficients. However, not all functions for the various components of a single vector may be the same. For example, in the Mudan model of China there are three types of functions for consumption expenditures of rural households:

- p a linear function of income, change in income, and relative prices
- i a linear function of income and change in income
- l a double-logarithmic function in income, relative prices, and time.

The first three sectors happen to illustrate the three types. Their equations are estimated by G7 with the following commands:

```
# Estimation of Rural Household Consumption
# in constant prices, per capita
f time = @cum(time,1,0) + 69
lim 80 90
f rni$=rni/rpindex
f drni$=rni$-rni$[1]
f lrni$ = @log(rni$)
punch rconsump.eqn 11 4 1990

ti 1 Rural Consumption per capita of Grains
f relprice = rp1/rpindex
f lrelprice = @log(relprice)
f lhcr1 = @log(hcr1)
r lhcr1 = lrni$,lrelprice,time
ipch hcr 1 l
gr *

ti  2 Rural Consumption per capita of Meat
f relprice = rp2/rpindex
r hcr2 =rni$,drni$
ipch hcr 2 i
gr *

ti  3 Rural Consumption per capita of Other Food
f relprice = rp3/rpindex
r hcr3 =rni$,drni$,relprice
ipch hcr 3 p
gr *

...

punch off
```

All but two of the above commands are basic G7 commands. The new commands, introduced especially for Interdyme, are “punch” and “ipch”. The “punch” command used in this example was:

```
punch rconsump.eqn 11 4 1990
```

It creates the file RCONSUMP.EQN for writing the results of the estimation. (The word "punch" recalls the days when these results would have been literally punched into cards.) The three following numbers are written onto the first line of the file. They indicate that the coefficients should

be stored in a matrix with 11 rows and 4 columns and that 1990 was the last year of estimation of the equations and should therefore be the year in which the rho adjustment error is set for these equations.

After the first regression comes the command "ipch hcr 1 l". (That last character is the letter l.) The "ipch" is the "Interdyme format punch" command of G. The "hcr" is an identifier of the functions for "household consumption - rural"; the 1 is the sector number; and the 'l' indicates that this is the "logarithmic" type of equation. In the second equation, the type indicator is 'i' for the "income only" type; and in the third, it is 'p', indicating a linear equation with a price term. The type can be any single character.

The results of this estimation are the following lines in the RCONSUMP.EQN file.

```
11 4 1990
hcr 1 l 4
 1 2 3 4
     0.116921       5.0804     0.785301    -0.456858   -0.0614286
hcr 2 i 3
 1 2 3
    -0.155197      10.9351     0.205765   -0.0288345
hcr 3 p 4
 1 2 3 4
       0.3238     -9.82683     0.183114    -0.111612     -27.2031
```

Note that the first line in the file contains the three numbers from the "punch" command. After it, each triplet of lines represents an equation. The first line repeats the information from the "ipch" command and adds the number of regression coefficients in the equation. In the third line, the first number is the rho for the equation; the remaining numbers are the regression coefficients. The second line of each triplet has as many integers as there are regression coefficients. They are the column in the regression coefficient matrix into which the regression coefficients go. In the example, they are all consecutive, but had the third category's equation used only income and relative price (and not also change in income), then the commands in G7 would have been:

```
r hcr3 =rni$,relprice
ipch hcr 3 p 1 2 4
```

and the line of integers would have been " 1 2 4". Since the coefficient matrix is originally set to zero, this device allows one type of equation to be used for all variants of the equation that differ from the base type only by omitting one of the variables. Note that the numbers at the end of the “ipch” command are optional. If these are not specified, then this indicates that all possible variables for this equation type are being used in this equation.

The full formal description of the ipch command is as follows:

ipch [which] <label> <sector> [type] [psn1] ... [psnN]

where:

which is used only when an equation has been estimated with "stack" or "sur" so that the coefficients are in rcoef1, rcoef2, etc. "which" is the suffixed number.

label is the name of the matrix into which the parameters go.

sector is the number of the sector to which the equation applies.

type is a character (a, b, c, etc) that signals the type of form of the equation. Type should not be a numeral.

psn1 is the column number in the matrix where the first regression coefficient belongs.

...

psnN is the column number of the last regression coefficient.

The function in the forecasting program to use this equation is the following. Near the beginning of the program, there is the statement

Equation rconsump("rconsump.eqn");

It defines "rconsump" as an object of the type Equation and reads in the values of the coefficients and rho's from the file RCONSUMP.EQN. Then in the loop through the years in the model's calculations, when it is time to compute the rural household consumption functions, there is the statement

hcrfunc(hcr, rp, rconsump);

The hcrfunc() function is then the following.

```
/* hcrfunc() -- the Rural Household Consumption functions for China model */
int hcrfunc(Vector& hcr, Vector& rp, Equation& rconsump)
{
   int n, i,j,t1;
   float cons, lp,ly,time;
   char which;

   time = t - 1900;
   n = rconsump.neq; // Number of equations
   /* rni is rural income per capita, rpindex is the price index for rural
      households, so rni_ is real income per capita in rural households.*/

   // Compute variables used in several equations.
   rni_[t]=rni[t]/rpindex[t];
   ly = log(rni_[t]);
   drni_[t]=rni_[t] - rni_[t-1];

   // Loop over the equations
   for(i = 1; i <= n; i++){
      j = rconsump.sec(i); // j is the sector number of this equation.
      which = rconsump.type(i);
      if(which == 'p'){
         cons = rconsump[i][1] + rconsump[i][2]*rni_[t]
            + rconsump[i][3]*drni_[t] + rconsump[i][4]*rp[j]/rpindex[t];
         }
      else if(which == 'i'){
         cons = rconsump[i][1] + rconsump[i][2]*rni_[t]
            + rconsump[i][3]*drni_[t];
         }
      else if(which == 'l'){
         lp = log(rp[i]/rpindex[t]);
         cons = rconsump[i][1] + rconsump[i][2]*ly
```

```
            + rconsump[i][3]*lp + rconsump[i][4]*time;
        cons = exp(cons);
        }
      else{
        printf("Unknown equation type %c in hcrfunc, category %d.\n",
            which,i);
        tap(); //Pause so that error message can be read.
        continue;
        }
      // Note the use of i and j in the following statement.
       hcr[j] = rconsump.rhoadj(cons,hcr[j],i);
      }
  hcr.fix(t);
  return(n);
  }
```

Note here that rconsump, as an Equation object, has certain functions:

rconsump.neq
: gives the number of equations. It may be less than the number of elements in the vector being defined. For example, the export vector usually has as many rows as does the input-output table, but many of them are zero. There need be no equations for these elements.

rconsump.sec(i)
: gives the sector number of equation number i. The equations are stored in the order in which they are read, which may be purely random. Thus, equation 1 might be for element 17 of the vector being defined, in the case of the example, pce. In this case, rconsump.sec[1] would give the value 17. This distinction is especially important for export or import equations where there is not an equation for every input-output sector. j = imports.sec(i), however, would give the input-output sector for equation i.

rconsump[i][j]
: gives the coefficient j in equation number i.

rconsump.type(i)
: returns a character that describes equation i.

rconsump.rho(i)
: returns the rho value for equation i.

rconsump.rhoadj(cons,hcr[j],i)
: returns the rho-adjusted predicted value for equation i. Here "cons" is what the equation predicted, and "hcr" is the value that was previously in hcr[j].

rconsump.GetLoc(j)
: This function was not used in the example, but may be useful elsewhere. It returns the equation number of the equation for a given element number of the sector being defined. It is thus the inverse of the sec() function.

These functions are available in any Equation object. Note also the &'s in the declaration of the "rconsump" function. These &'s cause the Vector and Equation objects to be passed by reference. Without them, a full copy of each Vector and Equation object would be made each time the function is called. Such copying would not only be wasteful of time but would mean that the "hcr[j] = " statement at the end would put the desired value into the copy which would then be thrown away when the function returns.

Note also the call to “hcr.fix(t)” near the end of the function. This applies any fixes that have been specified to the vector hcr. The next section discusses vector, matrix and macro fixes. Note that if the fix() function for a vector is not called, the fixes will not be applied.

5. FIXES AND THE FIXER PROGRAMS

Fixes, as used here, are ways to make a model work the way we want it to, not necessarily the way that emerges from its equations. The power that fixes give over a model can certainly be, and often has been, abused. Nonetheless, they have a legitimate role. Suppose, for example, we wish to consider the impacts of some event which the equations never dreamed of, like a natural disaster or a massive overhaul of the health care system. Then a fix is the natural way to convey to the model that the equations are not to be entirely trusted.

Interdyme has three types of fixes, those for macro variables, those for vectors and matrices, and a special type for industry outputs.

Macro Variable Fixes

Macro variable fixes are fixes applied to variables of type Tseries, which are defined using the Idbuild program described above. These fixes work very like those of models built with the G-Build combination, but also have much in common with the vector fixes described in the next section. The program that handles the macro variable fixes is called MacFixer. The input to MacFixer is a file prepared by the user with a text editor. It should have the extension .mfx . Once this file has been created, the program MacFixer is run by Model | MacFixer on the G7 main menu. The results of this program are written to a "macro fix bank", which is essentially a G bank, which can be read with G. The root name (the part of the filename before the dot) of the macro fix G7 bank is passed to MacFixer through the form which the above G7 command opens. It must also be passed to the simulation program by the form that opens on the command Model | RunDyme .

MacFixer requires a configuration file, called MACFIXER.CFG. It is created by G7 from the information provided on the form opened by the Model | MacFixer command. This form requires the name of the text input file, the root name of the G bank file used for base values for the index and growth-rate fixes (this would normally be the G bank created for use with the simulation program), the name of the G bank which will contain the values of the fixes, and the name of the

output check file. This last file shows the values of each fix in each year, and serves as a check on the results in the binary file.

While it is up to the user to name files, it makes good sense to give files for the same simulation the same root name. A simulation that involves low defense expenditures, for example, could have a G bank file called LOWDEF.BNK, and a .mfx file called LOWDEF.MFX.

There are several varieties of macrofixes that may be given, and they are described in the list below:

skip

is the simplest type of fix. It simply skips the equation and uses the values in the model G7 bank. For example:

```
skip invn$35
```

would skip the equation for the macro variable invn$35, and use the value already in the model G bank.

ovr

overrides the result of the equation with the value of the time series given. Values between given years are linearly interpolated. In the example below, the macro fix program would calculate and override a fix series that starts in 1992, ends in 2000, and moves in a straight line between the two points. For example,

```
ovr uincome$
  92  154.1
  2000  182.3;
```

would override the value of the forecast of uincome$ with the values shown for the years shown. Note that year can be either 2-digits or 4-digits (they are all converted to 4-digits in the program).

mul

multiplies the equation's forecast by a factor specified by the data series on the following line. For example,

```
mul ulfi$
  1992  1.0
  1995  1.05
  2000  1.10;
```

multiplies the forecast results for the macrovariable ulfi$ by the factors shown. Values of the multiplicative fix between the years shown are linearly interpolated.

cta

does a constant term adjustment. That is, it adds or subtracts the value of the time series to the result of the equation. The time series is provided by the fix definition. For example,

```
cta nonagincome
   1992 .0001
   1995  200
   2000  180;
```

is a constant term adjustment for nonagricultural income from 1992 to 2000. Intermediate values are of course linearly interpolated.

ind, dind

is a variety of the override fix that specifies the time series as an index. There must be data in the vam file for the item being fixed up until at least the first year of the index series specified. The value for the item in that year is then moved by the index of the time series given by the fix lines. For example,

```
ind wag01
  1982  1.0  1.03  1.08  1.12  1.15
  1997  1.21  1.29  1.31  1.34;
```

will move the value of wag01 in 1982 forward by the rate of change of the series given, and will replace the calculated value of wag01 by this value when the model is run.

The "dind" version of the index fix is the "dynamic index fix". This fix can start in any year, and does not rely on historical data being present in the databank. Rather, the fix is calculated based on the value of expression during the model solution for the first year of the fix.

gro, dgro

is a type of override fix that specifies the time series by growth rates. For the growth rate fix to be legal, there must be data in the vam file up until at least the year before the first year of the growth rate fix. Missing values of the growth rates are linearly interpolated.

```
gro wag01
  1993  3.1
  2000  3.4;
```

The "dgro" version is the "dynamic growth rate fix". This fix can start in any year, and does not require data to be available in the databank for the starting year of the fix. The growth rate is always applied to the value of the variable in the previous period.

stp, dstp

is a step-growth fix. It is like "gro" except that a growth rate continues until a new one is provided. A value for the final period is necessary.

```
stp wag01
  93  4.1
  95  4.5
  2000 5.0;
```

The "dstp" version is analogous to the "dgro" fix, only the values of the fix are interpolated differently.

rho

is a rho-adjustment fix. This type of fix finds the error made by an equation in the last year for which there is data; in the next year, it multiplies this error by the given rho and adds to the value forecast by the equation; the next year it multiplies what it added in the first year by rho again and adds the result to the equation's forecast, and so on. For rho-adjustment fixes, the format is:

rho <depvar> <rho_value> <rho_set_date>

where

rho_value is the value of rho.

rho_set_date is the year in which the rho-adjustment error is to be calculated. If none is provided, it is set in the first year of the run.

for example:

```
rho invn$38 .40 1995
```

tells the model to apply a rho-adjustment to the variable invn$38 using the value .40 for rho, and starting the rho-adjustment in 1995.

A rho fix with a rho_set_date works like a "skip" in years before the rho_set_date. A variable can have a "rho" fix in conjunction with and a "cta","mul","ind" or other type fix. The rho adjustment is applied before the other fix.

eqn

is an equation fix. This type of fix lets you dynamically introduce a new equation relationship into the model at run time. The advantage of this type of fix is that users of the model who are not programmers can introduce their own assumed relationships into the model, without having to change the model program code. It is also helpful for prototyping a model, where you want to quickly try out different equation relationships to see how they work, before coding them into the model.

The equation fixes use the same expression syntax as used in the “f” command and other commands in G. The format for equation fixes for macrovariables is:

eqn <Macroname> = <expression>
<year> <value> [<value> <value> ...]
<year> <value> [<value> <value> ...]

where: <Macroname> is a legitimate name of a macrovariable, <expression> is a legitimate expression, as described below, and the <year> <value> entries are in the same format as the data for other fixes, but indicate the years for which the equation fix is to take effect. They also represent the time series for a special variable called “fixval”, which can be used within the equation expression. This “fixval” variable can be used wherever a vector or macrovariable could be used.

Just about any expression that is legal in G7 is legal for an equation fix, except that only a subset of functions are implemented. These functions are: @cum, @peak, @log, @exp, @sq, @sqrt, @pow, @fabs, @sin, @pct, @pos, @ifpos, @pct, @rand and @round.

Lagged values of any order can be used, with the constraint that they must not be before the starting year of the model G7 bank (DYME.BNK). Macrovariables are read directly from memory. Lagged values of vector variables are read from the Vam file. Therefore, you can use

a lagged value of any vector as far back as the starting date of the Vam file, and you are not limited by whether or not that vector has been declared to store lagged values in memory in VAM.CFG.

Examples:

```
# Make the T bill rate equal to the average inflation plus some percent,
#   specified in "fixval".
eqn rtb = .34*gnpinf + .33*gnpinf[1] + .33*gnpinf[2] + fixval
  1998 1.0
  2010 1.5;
```

fol

is a follow fix. The follow fix allows you to specify that a macrovariable should move like some other quantity, which may be specified as a general expression involving vector and macrovariables, just like the equation fix.

The general format for the follow fix is:

```
fol <Macroname> = <expression>
        <year> <value> [<value> <value> ...]
        <year> <value> [<value> <value> ...]
```

The variable "fixval" should not be used in the follow fix expression. Its purpose is to specify a growth rate to add to the growth of the expression.

For example, if we would like to specify that Medicaid transfer payments grow like real disposable income per capita, plus 0.1 per cent, we could write:

```
fol trhpmi = di87/pt
    1997 0.1
    2010 0.1
```

shr

is a share fix. This fixed is used to specify that the macrovariable should be a certain share of another variable or expression, with the share specified by the fix value. Actually, the "share" is just a multiplier, so it can be any number.

The general format of the share fix is:

```
shr <Macroname> = <expression>
        <year> <value> [<value> <value> ...]
```

<year> <value> [<value> <value> ...]

In the share fix, the fix value is the multiplier or share to multiply by the right hand side expression.

When the input file as described above is ready and the macfixer.cfg file calls for its use, type "macfixer" at the DOS prompt to invoke the program Macfixer. When the model is running, calls to the "modify" function will apply the fixes, using the information in the macro fix G7 bank specified in the dyme.cfg for that run. Note that to view the fixes in the macro fix databank, specify the series name as the name of the macro variable, followed by a colon (':'), followed by a one-letter code signifying the type of fix.

These codes are as follows: skip ('k'), ovr ('o'), cta ('c'), ind ('i'), gro ('g'), stp ('s'), and rho ('r'). Therefore, to view a “cta” fix on the variable invn$38, do the following command in g:

```
ty invn$38:c
```

Macro fixes provide an alternative way to supply values of exogenous variables. Exogenous variables, to review, should be put into the "hist" bank in the process of running Idbuild. If the variable appears in no .sav file for a macro equation, then it is included in the PSEUDO.SAV file. The standard way of providing the values of the exogenous variables is then through "update" or other commands in Vam. Another possibility for providing exogenous values is to have a special run of G7 with the "hist" or other bank as the workspace bank. Finally, one can provide the exogenous values as macrofixes. For example, if we want disinc to be an exogenous variable, then -- however we are going to provide the values -- we need the statement

```
f disinc = disinc
```

in the "pseudo.sav" file. To use the macrofix method of assigning values, we need in the code of the model the statements

```
depend=disinc[t];
disinc[t] = disinc.modify(depend);
```

We could then provide the values with "ovr", "ind", "gro", or "stp" commands to the macrofix program, for example, by

```
gro disinc
   1995 3.0
   2000 3.5
   2005 4.0;
```

This method has the advantage of keeping all the fixes which constitute a scenario in one place. It also allows the use of the "gro" and "stp" fixes, which may be convenient. It has the disadvantage of adding an additional series to the banks which constitute the model and an additional statement within the model.

Vector and Matrix Fixes

The vector fixes are more complicated than macro fixes because they can apply to individual elements of a vector, to the sum of a group of elements, or to the sum of all elements in the vector. However, the format of the vector fixes is very similar to that of the macro variable fixes, described above. Matrix fixes at the current version (Interdyme 2.2, Fixer 1.5) are still rather simple, one fix being applicable to only one cell of a matrix. The preparation of the vector and matrix fixes is the work of the Fixer program. (Fixer is also sometimes referred to as VecFixer.)

G, it should be noted, normally prepares vectors of exogenous variables; fixes apply to vectors of endogenous variables. However, the Fixer program can also be used to supply the values of exogenous variables as well. Also, when building a model, before all the equations are finished, Fixer can be used to project the values of right hand side variables of some of the equations.

When and how are fixes applied as a model runs? Unlike the macro fixes, which are automatically applied when a macro regression equation is calculated, vector and matrix fixes are applied where the model builder specifies. At the point where the fixes for the vector x should be applied, the model builder must put into the program the line

x.fix(t);

The input to Fixer is a file prepared by the user in a text editor. It should have the extension .vfx . Fixer also reads the definitions of static groups of sectors and writes them into the GROUPS.BIN file which can be used both by the simulation program and by G. To use the Fixer program, *it is essential that the model's VAM.CFG file should have a vector called "fix"* with enough rows to allow one for each fix. As Fixer reads the fixes from the input file, it stores the numerical values of the fixes into this "fix" vector in the vam file. It also creates a "fix index" file, which will have the extension .fin and tells the simulation what to do with each fix. Finally, it produces a binary file with the definitions of groups, called GROUPS.BIN. If G7 has already produced a GROUPS.BIN file, Fixer reads it and may add to it.

Fixer is started by the command Model | VexFixer on the G7 main menu. From the information on the form which this command creates, G7 prepares a configuration file, called FIXER.CFG, which is read by the Fixer program. This form specifies the root names of:

the text input file
the fix index file
the vam file used for base values for the index and growth rate fixes
the name of the output check file, which will show the sectors in each group and the values of each fix in each year. It is used only for manual checking of the program.

While it is up to the user to name files, it makes good sense to give files for the same simulation the same "root" name. A simulation that involves low defense expenditures, for example, could have a vam file called LOWDEF.VAM, and a .VFX file called LOWDEF.VFX.

For vectors, fixes may apply to a single element or to a group of elements. The concept of a "group", already touched upon under Vam, is central to the working of Fixer. Basically, a group is

simply a set of integers, usually representing sectors in the model. Defining groups is useful because we often want to impose a fix on a group of elements in a vector. For example, we may want to control the total exports of the chemical manufacturing sectors. We might then create a group named "chem", which would contain the sector numbers of all the sectors in question. The command for defining a group is "grp <groupname>", where the groupname can be a number or a name. The sectors defining the group are then entered on the next line. For example,

```
grp 1
   7 10 12
```

creates a group called 1 of the sectors 7, 10, and 12. The "-" sign means consecutive inclusion. Thus

```
group zwanzig
   1 - 20
```

consists of the first twenty integers. Parentheses mean exclusion. Thus

```
group duo
   :zwanzig (2 - 19)
```

makes the group "duo" consist of the integers 1 and 20.

When a group is referenced after it is defined, its name must be preceded by a colon, as shown when "zwanzig" was used in the definition of "duo" above. Names of groups are case sensitive; commands for Fixer must be lower case. Groups do not have to be kept in numerical order and can be defined anywhere in the input file before the first time you used them. If you try to redefine an existing group, the program will complain, unless the new group has less than or equal to the number of elements in the old group. References to other groups can be used in new group definitions only if the groups referenced have already been defined.

Interdyme provides a number of ways for a fix to work. In all of them, a time series is specified by the fix. The forms of the fix differ in how they obtain and in how they apply this time series. The basic format of the input file for a vector fix is:

<command> <vectorname> <GroupOrSector>

followed on the next line by the year and value of the fix. The basic format of the input file for a matrix fix is:

<command> <matrixname> <row> <col>

Definitions of the 6 legal commands and examples follow.

ovr

overrides the result of the equations with the value of the time series given. Again, intermediate values are linearly interpolated. In the example below, the fix program would calculate and override fix series that starts in 1992, ends in 2000, and moves in a straight line between the two points. For example,

```
ovr ex 10
  92  154.1
  2000  182.3;
```

would override the value of the forecast of element 10 of the "ex" vector (probably exports) with the values shown for the years shown. Note that year can be either 2-digits or 4-digits (they are all converted to 4-digits in the program). As an example of a matrix fix,

```
ovr am 1 9
       1990 .23
```

```
          1995 .26
          2000 .28;
```

would override the value of the A-matrix in the Vam file for element (1,9), from 1990 to 2000. As before, missing values are linearly interpolated.

mul

multiplies the equation forecast by a factor specified by the data series on the following line. For example,

```
mul im 44
  1992  1.0
  1995  1.05
  2000  1.10;
```

multiplies the forecast results for imports of sector 44 by the factors shown. Values of the multiplicative fix on imports between the years shown are linearly interpolated.

cta

does a constant term adjustment. That is, it adds or subtracts the value of the time series to the result of the equations. The time series is provided by the fix definition. For example,

```
cta def :Alice
   1992 .0001
   1995  200
   2000  180;
```

is a constant term adjustment for defense expenditures of all sectors in the Alice group. Intermediate values are linearly interpolated.

ind, dind

is a variety of the override fix that specifies the time series as an index. There must be data in the vam file for the item being fixed up until at least the first year of the index series specified. The value for the item in that year is then moved by the index of the time series given by the fix lines. For example,

```
ind pceio :zwanzig
  1982  1.0  1.03  1.08  1.12  1.15
  1997  1.21  1.29  1.31  1.34;
```

will calculate the sum of the elements of the pceio vector included in the group "zwanzig" in 1982, will move that sum forward by the index of the series given, and will impose that control total on the those elements when the model is run.

The “dind” version of the fix can start in any year, and indexes the series to the value of the expression in the starting year of the fix.

gro, dgro

is a type of override fix that specifies the time series by growth rates. For the growth rate fix to be legal, there must be data in the vam file up until at least the year before the first year of the growth rate fix. Missing values of the growth rates are linearly interpolated.

```
gro out 10
  1983  3.1
  2000  3.4;
```

The "dgro" version of the growth rate fix can start in any year, and always calculates the series in the present period based on the value in the previous period.

stp, dstp

is a step-growth fix. It is like "gro" except that a growth rate continues until a new one is provided. A value for the final period is necessary.

```
stp out 1
  83  4.1
  95  4.5
  2000 5.0;
```

The "dstp" version is the dynamic version, which can start in any year. It is just like "dgro", except for the method of interpolation of the fix values.

eqn

The equation fix for vectors works in the same way as the version for macrovariables, with the exception that the name of the vector must be separated from the sector number by a space. For example:

```
# Make the pce deflator for category 3 grow like the aggregate PCE deflator,
#   based on the ratio in 1997, from 1998 to 2010.
eqn cprices 3 = cprices3{1997}/apc{1997} * cprices3
   1998 1
 2010 1
# Make corporate profits in sector 1 remain a constant share of total corporate
#   profits, equal to the share in 1997:
eqn cpr 1 = cpr1{1997}/vcpr{1997} * vcpr
  1998 1
  2010 1
```

fol

The follow fix specifies that an element or group of a certain vector should follow the expression on the right, plus or minus a certain growth rate, which can be specified in the body of the fix. It is often used to make imports of a certain commodity grow like domestic demand. For example, the following follow fix makes crude petroleum imports grow like domestic demand, plus 0.2 percent per year:

```
fol im 4 = dd4
    1998 0.2
    2030 0.2
```

shr

The share fix takes the value of the body of the fix (fixval), and multiplies the right hand expression by it, before assigning the value to the left hand side variable or group. Like the follow fix, a typical use for this fix might be in controlling the relation between imports and domestic demand. The example below specifies the share of domestic demand for imports of Radio, television and video equipment:

```
shr im 42 = dd42
    1998 .9
    2000 .92
    2030 1.0
```

When the input file as described above is ready and the FIXER.CFG file calls for its use, type "fixer" at the DOS prompt to invoke the Fixer program.

Output Fixes

The output fixes allow the values of output specified in the vam file to override values computed by the input-output equations. There is then the question of what to do with the difference. Interdyme offers two possibilities: add any excess demand to imports or simply ignore the difference. The options are specified in column 18 of the SECTORS.TTL file, which is where the names of the input-output sectors are. The options for this column are:

- e use the equation
- i add the difference to imports
- d put the difference in a vam vector named "dump", where nothing is done with it, but it can be displayed.

6. THE SIMULATION PROGRAM

The simulation program must be written individually for each country model. However, the main program changes little from country to country, and a number of tools are available in the Interdyme package which make writing the program quite easy. These tools include programs for reading in all of the vectors and matrices of a given year from the vam file, for storing them back again, for handling lagged values, for performing matrix and vector algebra on the Vam vectors and matrices, for solving input-output equations, for summing vectors, and for imposing control totals on vectors. Other tools are for using the equations prepared by Idbuild and still others for using detached-coefficient equations. Perhaps an example is the best way to introduce the various possibilities. We shall look at a simple version of the Chinese model, Mudan. We shall not reproduce the whole of the program but only those lines necessary to illustrate the methods to be used.

User Written Code

All code that needs to be modified for individual country models is in the three files MODEL.CPP, CONFIG.CPP, and USER.H. MODEL.CPP contains the main function of the model, called loop(), which calls all of the vector equation (detached coefficient) functions, as well as the macrovariable functions from HEART.CPP, created by the Idbuild program. CONFIG.CPP contains

code for the user interface screen at the start of the program, that requests model-specific information from the user. USER.H contains declarations of global variables that are model-specific, as as names of input files, or other special variables that need to be shared among many machines. The idea behind the structure of the files is that these should be the only files that the model builder needs to change. As new versions of Interdyme become available, one need only get the DYMEVxxx.ZIP file from the Inforum FTP site, unzip it to a temporary directory, and run GETDYME.BAT to bring over the new source code .CPP and .H files from the Interdyme distribution. In some cases, you may want to make changes to the files FEATURES.H and DYME.CPP. FEATURES.H is a list of features of Interdyme, and each line that contains a definition such as:

```
#define INCL_MACRO
```

may be commented out if you don't need to include that feature. Comments in this file indicate what features each define enables. For example, if you are writing a model that uses no macrovariables, commenting out "INCL_MACRO" will make the model code smaller and it will take up less memory.

Interdyme System Code

Although we have distributed source code for other modules, we urge that you use it only as a sort of ultimate documentation. *Please make no changes in these other modules* except for error corrections which you have discussed with the staff at the University of Maryland. If you change these routines, you will be in deep trouble when changes, improvements, and corrections are made in the standard version, for you will not be able simply to copy the new modules into your directory and use them. You may also make your model virtually impossible to link. If you need to add routines, please put them in the MODEL.CPP or add extra modules. Be cautious also with "corrections"; many "corrections", we have found, are a result of misunderstanding and in fact introduce serious errors. Please keep us informed of suspected problems. Also, please tell us of any problems you have or nice features which you have added in your MODEL.CPP or CONFIG.CPP modules. Before working with the tutorial that follows, you may want to print out the copy of MODEL.CPP that comes with Slimdyme, and see where in that file the following sections of model code would go. Also, you may want to refer to the *Overview of Interdyme* (available upon request) which contains other samples of features of the simulation program.

A Tutorial Example

The first lines of MODEL.CPP which need to be changed for a different country are these:

```
// prototypes for China model
void loop(void);
int prtfunc(Vector& prt, Matrix& Out, Equation& producty);
int hcrfunc(Vector& hcr, Vector& rp, Equation& rconsump);
. . .
int vinfunc(Vector& vin, Equation& invetory);
```

They are prototypes of the various functions to use detached coefficients to calculate various vectors. They all have to be changed to match whatever functions the new model has. Although they appear first, changing them may be one of the last things you do in adapting the model.

In the main program, the only lines which need to be changed for different countries are these which set up the screen to be displayed while the model operates.

```
/* The following 6 statements are the only ones in main() that can be
   changed for different country models to get different colors
   and title. */
textcolor(15);  // 15 is for white letters
textbackground(4); // 4 is red, 1 is blue, 0 is black background.
clrscr();          // clear the screen
gotoxy(15,1);      // put cursor in column 15 of line 1 on the screen.
cprintf("MUDAN -- MULTISECTORAL DYNAMIC ANALYSIS FOR CHINA");
gotoxy(30,2);      // put the cursor in column 30 of line two of the screen.
cprintf("Version 2.2");  // write text to the screen.
 /* end of changeables */
```

The next program to modify is configure() in CONFIG.CPP. It displays the contents of the DYME.CFG file and allows them to be changed for the individual run. Here is the DYME.CFG file for Mudan.

```
Title of run   ;Forecasting up to year 2000
Start year     ;1987
Finish year    ;2000
Discrepency yr ;1987
Use all data?  ;yes
VecFix file    ;Vecfixes
MacroFix bank  ;Macfixes
Vam file       ;dyme
G7 bank         ;dyme
BMV Pmatrix    ;bmv.pmx
debug start yr ;32000
Max iterations ;100
```

Note that it specifically names the file containing each packed matrix. In Mudan, there is only one, BMV.PMX.

Now here are the first few statements reading the DYME.CFG file. The only command line option on the dyme program is the name of this file. At the point where we now look, whatever is on the command line has been put in the string CfgFileName; its default value, as you will see here, is DYME.CFG.

```
len = strlen(CfgFileName);
if(len == 0) strcpy(CfgFileName,"dyme.cfg");
if((fpcfg = fopen(CfgFileName,"rt+")) == 0){
   printf("Cannot open %s as the configuration file.\n",CfgFileName);
   return(ERR);
   }
err += getopt(fpcfg,RunTitle,76);
err += getopt(fpcfg,szGoDate,4);
err += getopt(fpcfg,szStopDate,4);
err += getopt(fpcfg,szDiscDate,4);
```

```
err += getopt(fpcfg,szUseall,3);
err += getopt(fpcfg,VecFixFileName,60);
err += getopt(fpcfg,MacFixFileName,60);
err += getopt(fpcfg,VamFileName,60);
err += getopt(fpcfg,GbankName,60);

//  Each packed matrix has its own file name, read here.
err += getopt(fpcfg,BmvFileName,60);

...
if(err < 0) exit(1);
```

The getopt function gets a string from the DYME.CFG file. It skips everything on each line out to the first ';' and then reads in the rest of the line into its second argument. Its last argument, an integer, is the number of characters to put into the string. The reading of the DYME.CFG file can be changed as necessary to match the model. In particular, note the reading of BmvFileName. The name of each packed matrix must be read just as this one is, so if your model has packed matrices, you must follow the example for BmvFileName and put here the necessary commands to read their names. DYME.CFG is a good place to put the reading of miscellaneous options, such as whether to run one block of the model or not, whether to write debugging output to a file, or how many model iterations to allow.

The next commands display this information and allow the user to change it. They employ the CXL windowing library. You will want to change the line

```
wtitle("[  DRC MUDAN MODEL -- RUN OPTIONS ]",TCENTER,WHITE|_BLUE);
```

which supplies a title for the display page. To read and display additional file names, follow the example of these lines:

```
wprints(10,2,YELLOW|_BLUE,"Vector Fix File");
winpdef(10,17,VecFixFileName,"******************************",0,LBLUE|_LGREY,1,NULL);
```

In the "wprints" function, the first two integers are the screen co-ordinates (column and row) of the beginning of the text on the screen. In the “winpdef” function, the two integers are again screen co-ordinates, while VecFixFileName is the a string name. The function displays the information in this string and allows the user to change it. There should be one * for each character to be displayed and read. If there are more or less *'s than characters in the string being read, it does not work correctly. Hence, the getopt() function pads the end of the strings it finds with blanks so they will display correctly. Before these strings can be used for other purposes, such as opening files, these banks must be removed. That is the function of the depad() function shown in these lines:

```
depad(MacFixFileName);
depad(VamFileName);
depad(GbankName);
depad(BmvFileName);
```

Note that if you choose not to use the CXL windowing library, you can turn off these screen features by going into the file FEATURES.H, and commenting out the line that reads:

```
#define INCL_SCREEN
```

Note particularly the last of these lines; if you use a packed matrix, you must not only read and display its name, you must also remember to depad the name.

In the MODEL.CPP module, the central function for running the model, and the one which requires the most writing, is loop(). It makes extensive use of the ability of C++ to define objects that contain more than one kind of basic data. In particular, it uses the Vector and Matrix objects, defined in the MATRIX.H file and the Equation object defined in EQUATION.H. Such a Vector is much more than just an array of floating point numbers, although it contains that array. It also includes the number of rows and number of columns in the vector (one of them must be 1), functions for reading the vector from a vam file, and functions for adding or subtracting two vectors, and functions for multiplying the vector times a scalar or a matrix. The Matrix object similarly includes more than just the array of numbers. We will quickly see these objects at work in the beginning of loop() from the Mudan model of China.

```
void loop(void)
{
   int i;
   float toler,ptoler; // tolerances for Seidel iterative solutions for outputs and
prices.
   float sum[12],sigma,initkgdp,y,rscale,uscale,emprat;  // Variables particular to
Mudan

   /* Put matrix declarations here. The format is
      Matrix <Dymename>("<vamname>",<readflag>,<writeflag>)
      Vector <Dymename>("<vamname>",<readflag>,<writeflag>)

      where <Dymename> is what the object will be called in this program.
           <vamname>  is what it is called in VAM.
           <readflag> is 'i' to read only in the initial year.
                         'n' NEVER to read.
                      'a' to read if data is available for the year.
                      'y' to read in every year.
           <writeflag> is 'n' if the item is not to be written.

      Default values of readflag = 'y' and writeflag = 'y' are supplied
      if they are omitted in the declaration.  Vectors with lagged values
      should be declared both as a Matrix and as a Vector.
   */
   Matrix A("am",'a','y'),MCU("bmcu",'a','y'),MCR("bmcr",'a','y');
   Pmatrix B("bmv",'a','y');
   Matrix Iprices("iprices",'i','n'), Out("out",'i','n'),
      Prices("prices",'i','n'), Up("up",'i','n'),Rp("rp",'i','n');

   Vector hcr("hcr",'a'), hcu("hcu",'a'), invn("invn"), invr("invr"),
      cr("cr",'n'), cu("cu",'n'), cap("cap",'n'), pub("pub"),
      vin("vin",'a'), accum("accum",'n'), export("exp",'a'),
      imp("imp",'a'), netexp("netexp",'n'), othdm("othdm"), fd("fd",'n'),
   ....

   Vector w1(11),w2(19),wagerate87(33);
```

These declarations of the Vector and Matrix objects

- grab the space necessary for storing the objects
- set up a correspondence between the objects and the vam file
- enable the C++ compiler understand references to the variables in subsequent statements.

Note that B, corresponding to “bmv” in the vam file, is declared as a Pmatrix, that is, as a packed matrix. The file that contains this packed matrix is specified in the DYME.CFG file.
Read carefully the comment in the program, especially about the use of the read and write flags. Their use can save a lot of time in execution.

Next comes the definition of three "scratch" vectors that do not appear in the vam file.

```
Vector w1(11),w2(19),wagerate87(33);
```

The following lines define detached-coefficient equation objects for consumption of rural households (rconsump), for consumption of urban households (uconsump), and so on for other sets of detached coefficient equations in the model. The argument to each of these declarations is the DOS name of the file with the results of estimating the equations in G. It is convenient but not necessary for these file names to end in .EQN.

```
Equation rconsump("rconsump.eqn");
Equation uconsump("uconsump.eqn");
Equation imports("imports.eqn");
Equation exports("exports.eqn");
.....  etc.
```

The next group of statements gives special treatment to the import equations because they will be used inside the Seidel solution of the input-output equations. There it is necessary to be able to find quickly the number of the import equation for any given input-output sector. Therefore, we make here a vector, secimp, such that secimp[i] is the number of the import equation for i-o sector i. This secimp has been declared as a global variable.

```
secimp = ivector(1,imp.rows());
for(i=1; i <= imp.rows();i++)
   secimp[i] = 0;
for(i =1;i <= imports.neq;i++)
   secimp[imports.sectors[i]] = i;
```

The next few lines just put something on the screen to watch while the model runs and to report on its progress.

```
gotoxy(30,6);cprintf("ITERATION COUNTS\r\n");
cprintf(" Year Entire Model  Output Seidel  Price Seidel  Income  OVAScale");
```

In dynamic input-output models, data on various vectors is not always available up through the same year. Interdyme allows the user the option of using actual data for a vector if it is available or of ignoring the available data. The option of ignoring the available data is important for testing the model by a historical simulation or for performing counter-historical simulations. For forecasting, however, one usually wants to use all available data. Consequently, there must be some way to tell the program how far actual data goes for each vector which has an 'a' read flag. This information is provided in a file named LASTDATA; for Mudan, this file begins

```
hcr 1990
hcu 1990
vin 1990
exp 1991
imp 1991
```

```
...
```

This file is read by the next statement:

```
// Set date of last available data.
lastdata();
```

The lastdata() function will check that a date has been declared for any Vector or Matrix declared with a read flag of 'a'.

The function CheckDeclar checks to see that all vectors in the vam file have been declared and that all vectors that have been declared in vam.cfg to have lags have also been declared as Matrices. It also sets up the correspondences between these vectors and the corresponding matrices. If no reporting on the screen is desired, the value of its argument, called "silence" should be "TRUE". Normally, "silence" should be FALSE when initially running the program. Mudan, however, has some vectors used only in Vam; they will produce warning messages if silence = FALSE, so in the final model it is set to TRUE. We show it here, however, as it should be in the early stages of model development:

```
int silence=FALSE;  // call CheckDeclar to report trouble
CheckDeclar(silence);
```

The next line sets the value of the tolerance in the Seidel iterative solution method. toler is for the solution for outputs, while ptoler is for prices. Note that prices are equal to 1 in the base year while outputs are large numbers, so the two tolerances must be quite different.

```
toler = 1.; ptoler = .0001;
```

Calculating the base year error term for subsequent rho adjustment should be done only after the last and best forecast of the year's values has been made. By putting "setrho" equal to 'n', that is, to "no", we tell all of the equations that they should not set the value of the error term for rho adjustments.

```
setrho = 'n';
```

At last we are ready to calculate. The following **for** loop moves the model through the years of the simulation. We here eliminate a great deal of the actual Mudan model to show just the essence of this loop. It begins:

```
for (t = godate; t<= stopdate; t++) {
   cprintf("\r\n%5d",t);  //Show the year on the screen.
   load(t);  // Load this year's values of all variables.
```

This simple load function is one of the most powerful in Interdyme. It brings all the Vectors and Matrices their values for this year from the vam file. In Mudan, a preliminary program, Across, has produced, for 1980 - 1990, balanced versions of the A matrix and the three bridge matrices, MCR for rural consumption, MCU for urban, and B for capital investment. The row scaling factors, the ami Vector, is exogenously projected for the years beyond 1990. Thus, in running the model, for 1990 and prior years, these matrices should be read from the vam file. After 1990, the 1987 base year matrices are read (the A Matrix, for example, by A.load(1987)) and subjected to row scaling by the

ami Vector. The distribution matrices are then normalized so that their columns sum to 1. Thus, the 1987 matrices become, as it were, parameters in a more complicated method of producing coefficients. Also, Tseries macro variables whose values are "missing" (really, equal to -.000001) need to be started at the previous year's value. This "moving up" of the time series is done by the call to "upets". The code for all of this work is:

```
if(t > 1990){
/* If this year's value of a time series is "missing", set it
   equal to last year's value. */
upets();
// Read 1987 values of matrices
A.load(1987); B.load(1987);MCR.load(1987); MCU.load(1987);
/* Scale the rows of A by the ami vector; the 'y' means skip
   the diagonal element. */
rowscale(ami,A,'y');
rowscale(ami,MCR,'n');
rowscale(ami,MCU,'n');
rowscale(ami,B,'n');
/* Set all values of the bmcucum vectors equal to 1. */
bmcucum.set(1.);
bmcrcum.set(1.);
bmvcum.set(1.);
/* Scale the columns of MCR to equal 1. Record scaling factors
   in bmcrcum by multiplying them by what was already there. */
colnorm(MCR,bmcrcum);
colnorm(MCU,bmcucum);
colnorm(B,bmvcum);
arith("after colnorm B");  // Check for arithmetic errors
}
```

At various points in the code, there are calls to arith() with a message about where in the code the call is. This arith() routine checks to see if there has been a arithmetic exception -- divide by 0 or the like. If none has been found, it continues; but if one has been found it writes "Arithmetic exception" and your message. In the above example, it would have written "Arithmetic exception after colnorm B". The program then pauses and gives you the chance to exit by tapping Escape or to clear the error and continue by tapping any other key. The arith() call can have a second argument, an integer, which will be displayed after the message.

Mudan has both a product and an income side. It begins with the product side, them computes the price and income side, and then returns to recompute the product side. This iterative process is repeated until convergence is reached or a maximum number of iterations is reached. Here we start the iterative loop.

```
iter = 0; // Start an iteration count
top:   // top of loop for solving for kgdp
iter++;
gotoxy(12,wherey()); // move cursor to column 12 of the current line.
cprintf(" %d",iter); // write the iteration count
 initkgdp = kgdp[t];  // record the initial value of constant price GDP

/* Compute percapita, constant-price consumption of rural households
   -- the hcr vector. */
 hcrfunc(hcr, rp, rconsump);
```

```
/* Scale rural consumption so that consump + saving = income.  Note the use of the
   "dot" or "inner product" function.  rp is the vector of rural prices; rni is
   rural income, and rsavrat is the rural saving rate.*/

rscale = (rni[t]*(1. - 0.01*rsavrat[t]))/dot(hcr,rp);
```

The ‘useall’ flag is set on the opening screen. It is 'y' to use all available data. This option would be used for forecasting. When used to do historical or counter-historical simulations for testing or policy analysis in the past, useall is set to 'n'.

```
if((useall == 'n' && t > godate) ||
   (useall == 'y' && t > hcr.LastData())) hcr = rscale*hcr;

// Compute percapita consumption of urban households -- the hcu vector.
hcufunc(hcu, up, uconsump);

// Scale urban consumption so that consump + saving = income
uscale = (ulfi[t]*(1. - 0.01*usavrat[t]))/dot(hcu,up);
if((useall == 'n' && t>godate) ||
   (useall == 'y' && t > hcu.LastData()))
   hcu = uscale*hcu;

// Multiply percapita by population to get total consumption
w1 = (rpop[t]/10000.)*hcr;//w1 is vector of total rural hh consump
w2 = (upop[t]/10000.)*hcu;//w2 is vector of total urban hh consump

// Convert from household categories to IO sectors
cr = MCR*w1;
cu = MCU*w2;
...
```

Several lines are omitted here to calculate the pub, accum, export, and othdm vectors of final demand. They involve nothing new. Then comes the vector addition to obtain total final demand. It is possible to write the addition so simply because addition has been defined for vector objects in the matrix.cpp module of code. This "overloading" of operators -- in this case, the + operator -- is a C++ trick not available in C.

```
fd = cr + cu + pub + accum +  export + othdm;
```

No programs are right the first time. Finding bugs is often a matter of printing out vectors. The following lines are a built-in debugging device. If the year is greater than "debug", then printing occurs. The "debug" value is set in the DYME.CFG file and on the opening screen. If no debugging is desired, it is set to a number beyond the last year of the simulation. Display() is a function of a Vector object. The arguments are the title of the vector as displayed on the screen, the field width of each item displayed, and the number of decimal places to show. Default values for the last two are 7 and 1. Of course, if you do extensive work with Interdyme programming, it would also pay to become familiar with the Borland Turbo Debugger.

```
if(t >= debug){
   printf("\r\nrscale = %6.3f  uscale = %6.3f\r\n",rscale,uscale);
   fd.Display("fd before Seidel");
   hcr.Display("hcr before Seidel");
   hcu.Display("hcu before Seidel");
   ....
   }

/* Compute the outputs, out, given the A matrix, the final demands, fd,
   the import Equations, imp (the import vector), the triangulation order, triang,
```

```
   and the error tolerance, toler.
   */

Seidel(A,out,imp,imports,fd,triang,toler);

// Seidel did not put the computed imports into the imp vector. Do so.
impfunc(imp, out, imports);

// Subtract imports from the previously computed fd vector.
fd = fd - imp;

/* Compute GDP in constant prices as the sum of the elements of fd.
   sum() is a function which every Vector has.
   */
 kgdp[t] = fd.sum();

// Put the current values of "out" into OutLag with 0 lags.
update(out,OutLag);
// Compute labor productivity
prtfunc(prt, Out, producty);

// Compute employment, emp. ebediv is "Element-by-element division".
emp =100.*ebediv(out,prt);

....
/* A number of lines are omitted here leading up to the definition of unit value
   added, unitva, which then goes into the PSeidel function to obtain prices.
   */
 PSeidel(A,prices,unitva,triang,ptoler);     /*  Get Prices  */

/* With the prices known, we can compute DGP in current prices with the dot-
   product function.      This function returns a float, which is what we need.
   If we wrote ~prices*fd, we would get a 1-by-1 Matrix, and the compiler will
   tell us that it cannot convert a Matrix to a      float.
   */
gdp[t]  = dot(prices,fd);
deflator[t] = gdp[t]/kgdp[t];

/* A/x where A is a Matrix and x is a vector means what is usually written as A'x.
iprices =B/prices;  // Investment prices
rp = MCR/prices;    //Rural prices
up = MCU/prices;    //Urban prices
rpindex[t] = dot(hcr,rp)/hcr.sum(); // Rural price index
upindex[t] = dot(hcu,up)/hcu.sum(); // Urban price index
...
/* Test whether implied constant price GDP is close enough to its
   assumed values.  If not, go back to the top of the loop.
   */
if(fabs((initkgdp - kgdp[t]) > 10. || (emprat > .005 && t > godate))
   && iter < maxit) goto top;
```

At this point, either a solution has been reached, or the maximum number of iterations has been met. In either case, it is now time to go on to the next year's calculations. Before doing so, however, we make a final call to all functions with a rho adjustment to set or update the error term. To signal to these functions that they should now do this update, we turn the ‘setrho’ flag to 'y'.

```
setrho = 'y';
hcrfunc(hcr, rp, rconsump);
hcufunc(hcu, up, uconsump);
impfunc(imp, out, imports);
....
```

Next, we call all of the macro equations for the sole purpose of setting or updating the error term of the rho adjustments.

```
callall();
setrho = 'n';
```

Finally, we store to the vam file this period's values of all the Vectors and Matrices which have 'y' as a write flag. In the process, the store() function will also put this period's values into all matrices that carry lagged values of vectors and shift the vectors in these matrices all back a year.

```
store(t);
}
```

That } marks the end of the loop on t which moves the model through the years. At the end of the run, all macro variables (Tseries variables) must be written to the model G bank by the storets() command.

```
storets();

printf("\nThis run has been finished . Please see the results!\n");
}
```

And that is the end of the simulation program. To compile and link it, you can just type "dm" (for Dyme Make) at the DOS prompt. This batch file simply gives the command:

```
make -fdyme
```

Note that there is a make file available for building large models with the Borland Power Pack DOS extender. This make file is named DYME.MX4. To use it, just change the DM.BAT file to read:

```
make -fdyme.mx4
```

The Use of Multiple Vam Files and G Banks in an Interdyme Model

This is an advanced topic. Note that it is possible to build a model which uses more than one Vam file for reading and writing. In the Slimdyme version of Interydme which is distributed, there are commented-out examples of how this may be done. The use of multiple vam files is especially useful in cases where you want to link various models, such as in a bilateral trade model, which uses data from many different country models.

Look in the DYME.CPP file for the lines:

```
// vf[0] is the default simulation destination file.  Other VamFiles
//   may also be opened.
vf[0].OpenVam(VamFileName); // vf declared globally.
// We seem to be having problem with GCC inline:
check=vf[0].IsErr();
check=vf[0].IsOpen();
if(!vf[0].IsOpen()) {
printf("\nCannot open %s for update.",VamFileName);
goto cleanup;
}
// This is an example of how to open a second Vam file:
/*
vf[1].OpenVam("test");
```

```
if(!vf[1].IsOpen()) {
    printf("\nCannot open %s for update.","test.vam");
    goto cleanup;
    }
*/
vf[0].Setrunname(RunTitle);     // Put runtitle in Vam File.
  // (VamFile destructor automatically writes it out)
InitMat(vf[0]); // Read information from VamFile.
```

Note that the vf[] array is an array of vam files. By default, only the first one (position 0) is opened, and this is the file which the load() and store() functions will access. The OpenVam function can be used to open vam files on vf[1], vf[2], etc., up to the number MAXVAMFILES defined in DYMESYS.H. The vf[] array itself is declared in the file DYME.INC. If you look at this file, you will see that vf is an array of type VamFile, which is another class used in Interdyme, to work with vam files.

In MODEL.CPP, you may specify that a given vector is to be read or written from a given vam file by giving the vam file as the first argument to the Matrix or Vector constructor. For example:

```
Vector lout(vf[1],"out"), pout(vf[2],"out"), q(vf[3],"out");
```

This code specifies that lout will contain the vector from the 2nd vam file with name "out", pout will contain the vector of the same name from the 3rd vam file, and q will contain yet another vector of the same name from the 4th vam file. Note, however, that if you try to put a fix on the vector "out", that this will apply only to the vector in the first declaration encountered in MODEL.CPP.

Summary of Vector, Matrix and Tseries Functions and Operators

Now that the tutorial example has shown how the Vector, Matrix, Equation, and Tseries objects may be used in practice, it is perhaps useful to give a complete description of their properties, not all of which were illustrated.

If A, B, and C are each a Matrix, and x, y, and z, each a Vector and f a float, then the following uses of operators are defined provided the dimensions of the matrices and vectors are appropriate:

A + B	matrix sum
A - B	matrix difference
A*B	matrix product
f*A	scalar matrix product (note that you cannot do A*f)
x + y	vector sum
x - y	vector difference
f*x	vector product. The result is a Matrix.
A*x	matrix-vector product
x*A	vector-matrix product
~A	the transpose of A
~x	the transpose of x
x/y	the transpose of x times y (x'y)
A/x	the transpose of A times x (A'x)
x/A	the transpose of x times A
A/B	the transpose of A times B
x/f	vector divided by a scalar
!A	the inverse of A
x[i]	element i of Vector x. This may be used on either side of an = sign. The subscript i is checked for falling in the allowable range.
A[i][j]	element i,j of Matrix A. The row subscript is checked, but not the column. The expression may be used on either side of an = sign.
A(i,j)	element i,j of Matrix A. Both row and column subscripts are range-checked. The expression may be used on either side of an =.

Any combination of the operators is also possible. Thus an expression like

```
x = !(I - A)*(c + i + g)
```

works fine.

Any vector or matrix, A, has these functions

A.rows()	gives the number of rows.
A.columns()	gives the number of columns.
A.firstrow()	gives the first row (default = 1).
A.lastrow()	gives the last row.

A.firstcolumn() gives the first column (default = 1).
A.lastcolumn() gives the last column.
A.load(year) loads into the matrix or vector the value for the specified year (regardless of what year's values are in other matrices or vectors.)
A.set(value) Sets all cells of A equal to the given value. (The value should be a float.)

Any vector, x, has the following special functions.

int x.First()
returns the number of the first row of x if x is a column or the first column of x if x is a column.

int x.numelm()
returns the number of elements in x.

float x.sum()
the sum of the elements of x

float x.csum(char* Group)
forms the sum of the elements of x in the group definition specified. For example, you could write: x[t] = out.csum("1-3,20-41");

void x.scale(float total)
causes the vector x to be scaled by right-direction scaling to reach the specified total. In right-direction scaling, if both positive and negative elements move in the same direction. Either both get larger algebraically or both get smaller.

void x.SetLastData(int date)
records that the last year for which data is available for Vector x is the given date.

int x.LastData()
returns the last year for which data is available for vector x.

void x.Display("message",[width],[decimal])
displays the values of the x vector. The optional parameters are the field width and the number of decimal places. Default values are 7 and 1. If the decimal places are to be specified, the field width must also be given.

void x.set(<float>)
sets all elements of x equal to the given floating point number.

x.fix(t)
applies to the current contents of the x vector the fixes specified for it by the Fixer program, for the year t

x.fix(t,i)
applies fixes only to sector i in the vector for year t

x.fix(t,-1)
applies only the group fixes to the vector for year t.

There are also three special functions for working with two vectors, x and y, which have the same number of elements.

Vector ebemul(x,y)
gives the Vector obtained by element-by-element multiplication of x and y.

Vector ebediv(x,y)
gives the Vector obtained by element-by-element division of x by y.

float dot(x,y)
gives the inner or dot product of the vectors as a float. It differs from x*y or x/y (which is ~x*y) in two respects. First, dot returns a float while x*y returns a matrix. (Generally it is the float that is needed in a model.) Second, in dot(x,y), x and y may be both columns or both rows or one of each. In x*y, one must be a row and the other a column. dot(x,y) just gives the sum of the element-by-element products.

Vector ebemul(A,i,y)
gives the Vector created by an element-by-element multiplication of row i of the matrix A by the vector y. In practice, A is usually a matrix of lagged values of a vector.

Several functions involve matrices, here denoted A or B.

void A.Display("message",[width],[decimal])
displays the A matrix row by row. For large matrices, the rows will be "folded" so that it takes several lines on the screen to show each row. The row number is shown at the beginning of each row, and the column number of the first column on a line precedes the data on the line. The optional parameters are the field width and the number of decimal places. Default values are 7 and 1. If the decimal places are to be specified, the field width must also be given.

rowscale(b,A,flag)
scales the each row of the Matrix A by the corresponding element of the Vector b. The flag answers the question Skip the diagonal elements? It is y for yes and n for no.

colnorm(A,x)
scales the columns of A to equal 1. The scaling factors are multiplied by the previous contents of the x vector and the product is put back into the x vector.

pulloutcol(v, A, k)
pulls column k of A into v.

putincol(v, A, k)
puts v into column k of A.

pulloutrow(v, A, k)
pulls row k of A into v.

putinrow(v, A, k)
puts v in row k of A.

v = colsum(A)
puts the column sums of A into the vector v.

v = rowsum(A)
puts the row sums of A into the vector v.

A = ebemul(B,C)
multiplies each element of B by the corresponding element of C. Thus, A(i,j) = B(i,j)*C(i,j).

A = ebediv(B,C)
devides each element of B by the corresponding element of C. Thus, A(i,j) = B(i,j)/C(i,j). If C(i,j) = 0, A(i,j) = B(i,j).

Lagged Values

If a vector, say q, is declared in the VAM.CFG file to have lagged values used in the model, then both a Matrix and a Vector should be declared for q, something like this:

```
Matrix Qlag("q",'n','n');
Vector q("q",'i');
```

The current year's value of q will be in row 0 of Qlag, last year's in row 1, and so on. When the "store" command at the end of the year is done, the Qlag matrix will automatically be shifted back and the values just computed put in both rows 0 and row 1. If it is necessary to update row 0 of Qlag with current values of q before the end of the year's computations, the update command can be used. Its form is just

```
update(q,Qlag).
```

Seidel and PSeidel

Two functions are available for the Seidel solution of equations, Seidel and PSeidel. The Seidel is used for solving the equations for outputs while PSeidel is used for solving the equations for prices. Seidel makes special provision for solving for imports simultaneously with outputs. The prototypes for the calls for the two equations are:

```
Seidel(Matrix& A, Vector& out, Vector& imp, Equation& imports, Vector& fd,
    int *triang, float qtoler);

PSeidel(Matrix& A,Vector& price, Vector& va, int *ptriang, float ptoler);
```

Note that both Seidel and Pseidel have int* arguments triang and ptriang. These are the lists that define the triangulation ordering. Upon initializing, the Dyme program looks for two files TRIANG.DAT, and PTRIANG.DAT, from which to read these vectors. The vectors serve two purposes. The first is to implement a reordering of the Seidel solution, to make it as recursive, and therefore as fast, as possible. Inforum has developed a program called TRIANG.CPP which you can use to calculate this triangulation ordering for your own input-output matrix, final demands and value added. The second purpose of the triangulation file is to leave certain sectors out of the Seidel solution, by simply leaving them out of the triangulation list. It is not necessary to calculate these vectors to run the model, but more efficient. Since Interdyme version 2.2, if you choose to skip the creation of these files, the Interdyme model will create a vector for you, equal to the sector list.

There are several functions for working with the "macro" or Tseries variables. If ts is a Tseries, then

ts.in()	reads in from the G bank
ts.out()	writes to the G bank
ts[t]	represents the value of the series in year t on either side of an equation.

Two functions work on all Tseries variables:

upets()	replaces missing values in the current period by the previous year's values in all Tseries variables.
storets()	stores into the model G bank the values in all Tseries variables.

7. COMPARE -- TABULATING AND COMPARING RESULTS

Vam can be run directly after a run of the model to display the results on the screen or to make graphs. But to make printed tables, Compare is the tool.

The Compare program (version 6.49) is a general table making program that can be used to make tables from various source data bank formats. As the name suggests, it is particularly adapted to comparing results of several alternative runs of a model, but it can also list the contents of a single data bank or the results of a single run of a model. When being used to show a base case and several alternatives, it can show the alternatives as actual values, or as deviations from the base, or as percentage deviations from the base. In addition to making tables of macroeconomic and vector variables, Compare also has the ability to make a "matrix listing" of all the cells in any input-output table. The results are written to a file which can then be viewed with an editor or printed. Compare is part of the G-Build software, but is also part of the Interdyme distribution.

The Compare Stub File

To use Compare, one first prepares a "stub"[1] file. One for the Mudan model begins like this:

```
\ date 1987 1990 1995 2000  87-90 90-91 91-95 95-100
;                                    OUTPUT
;
&
out1   ; 1  Agriculture
out2   ; 2  Coal
out3   ; 3  Crude Oil & Natural Gas
...
out33  ; 33 Public Administration
*
;                                    EXPORTS
;
&
exp1   ; 1  Agriculture
exp2   ; 2  Coal
exp3   ; 3  Crude Oil & Natural Gas
...
```

The first line of the .stb file should give the dates of the periods which are to be printed. It must begin with "\date" as shown in the example. The example will show on each line of the table the

[1] Experience shows that non-native speakers of English can never imagine why these files should be called "stub" files. A "stub" is something that remains after part has been cut or broken off. It can be used of limbs (human or tree), or of teeth; but it is also the part of a table down the left side with the names of the various series from which the values, to the right, have been cut off. Hence our usage of the word.

actual values of the series being listed on that line for 1987, 1990, 1995 and 2000. It will then show the annual growth rates between 1987 and 1990, between 1990 and 1991, between 1991 and 1995, and between 1995 and 2000. A ';' at the beginning of a line causes the line to be printed as it stands. A '&' cause the dates to be printed on the line. A '*' causes a new page to be started. A line like

```
out1   ; 1  Agriculture
```

causes the series "out1" to be listed on a line of the table with the title

1 Agriculture

on the left. The "..." in the example are not commands to Compare; they show where lines have been omitted in the example. The table produced by these lines looks something like this.

```
             Page   1

                           Base          VAM          file,          before          simulations

                                  OUTPUT

                                   1987   1990   1995   2000   87-90  90-91  91-95  95-00
 1  Agriculture                  4675.7 5380.9 7322.9 9521.2   4.68   6.72   6.02   5.25
 2  Coal                          273.0  348.5  464.5  587.0   8.15   7.72   5.25   4.68
 3  Crude Oil & Natural Gas       265.2  300.3  331.4  353.2   4.14   2.42   1.87   1.27
...
 33 Public Administration         378.8  480.1  616.5  791.6   7.90   5.00   5.00   5.00

                    Page   2

                           Base          VAM          file,          before          simulations

                                  EXPORTS

                                   1987   1990   1995   2000   87-90  90-91  91-95  95-00
 1  Agriculture                   193.7  238.1  321.1  404.1   6.88   6.74   5.79   4.60
 2  Coal                           14.4   18.1   25.1   32.0   7.61   7.36   6.25   4.88
 3  Crude Oil & Natural Gas        43.4   48.7   52.3   55.8   3.82   1.44   1.39   1.31
```

More formally, the commands that can be in the "stub" file are:

*** [m] [n]** If there is not room on the page for m more items plus n lines, go to the top of a new page and print the titles of the alternatives runs and the dates across the top. Examples:

* this is an unconditional new page.

* 3 If there are two alternatives and there are less than eight lines remaining on the page, start a new page. The eight lines is derived from the six lines required to list three items with two alternatives each, plus two lines for the spaces between them.

* 3 4 If there are two alternatives and there are less than twelve lines remaining on the page, start a new page. The twelve is derived from the eight in the preceding

example plus the 4 indicated on the line. These additional four lines would be text provided by the stub file itself.

; Print the line just as it stands.

& Print the dates across the page above the appropriate columns. Note that this has changed from the '@' used previously! This allows '@' to be used for function names.

\ date Set the dates to be listed and the periods over which growth rates are to be shown. The \ date command can be used to change the dates in the course of the listing.

\ fw dp pl tm bm tw
where fw dp pl tm bm and tw are all numbers. This is an optional line to set the field width of each printed number to fw, the number of decimal places to be printed to dp, the page length to pl lines, the top margin tm lines, the bottom margin to bm lines, and the width of the titles to tw. The last three may be omitted. The default values are 7 1 60 3 9 32.

\ar use if monthly or quarterly data is given at annual rates (the default). (ar = annual rates)

\pr use if monthly or quarterly data is given at monthly or quarterly rates (pr = period rates).

\ti <title>
supply a title which will be listed at the top of each page.

\matlist <sectors>
perform a matrix listing of the indicated sectors. This option will be explained further below. It has two related commands:

\row
\column Specify whether the matrix listing shows rows or columns, and

\cutoff <fraction>
Specify the fraction of the row or column total which a matrix cell must account for to be listed.

A line beginning in any other way will be presumed to begin with a variable name, such as out1, or an expression. For the expressions, all the functions available in G7 are also available in Compare. In addition, Compare has a new function, called @csum(), which is used to sum up a specified group of industries for a given data concept. For example, suppose you wanted a certain line in your table to contain the sum of exports ("exp") for sectors 1 to 10. Then the formula to use in the name section of the stub file would be:

```
@csum(exp,1-10)
```

The expression:

```
@csum(exp,1-12 (4-7) 15 18)
```

would include exports of sectors 1 to 12 inclusive, except for sectors 4 to 7 inclusive, and then sectors 15 and 18 in addition.

After the name or expression comes a ';', and after the ";" come up to thirty-two characters which will be printed at the left side of the line. Leading blanks -- blanks between the ";" and the first visible letter on the line -- will be printed and can be used to indent the titles. Actually, the number of characters after the ';' which will be printed is controlled by the "tw" parameter of the "\ fw do ok tm bm tw" command described above. The number of characters actually present on the line of the stub file may be more or less than the number to be printed. If less, the title will be padded with blanks on the right; if more, it will be truncated to the number to be printed.

We like to use the extension .STB in the names of such stub files, but the practice is not necessary.

Once the stub file is ready, we can run Compare by typing "compare" at the DOS prompt. As soon as Compare starts, it asks you how many alternatives you want to see in the table. (See the sample session in the example below). Respond with '1' if you are just making a table from just one bank. Next, for each alternative, you are asked to specify what type of bank this alternative should be read from, and then the root name of the data bank file. Answer this first question with one character, and then hit [Enter]. Type 'w' if this is a normal or workspace type G bank (.BNK), 'c' if this is a compressed G bank (.CBK), 'h' if this is a hashed G bank (.HBK), 'd' if this is a dirfor file (.DFR), and 'v' if this is a vam file (.VAM).

Note that if you want to use a Dirfor file made with the former Slimforp software, you must have the DIRFOR.DAT file corresponding to that file in the current directory in which you are running Compare. You must also make sure that the MACRONAM.BIN specified in that file points to a valid location. (See the LIFT manual, in the Display chapter for more details on DIRFOR.DAT.)

If you are assigning a vam file, note that Compare automatically assumes that each vam file has a "sister" G bank, with the same root name in the same directory. It will try to open this file to find series such as macrovariables which may not be in the vam file.

When asked for databank file names, give only the root name of the file without the three-letter extension -- .VAM, .BNK, .CBK, and so on.

Next, you are asked whether you would like to see the alternatives in actual values ('a'), as differences ('d') from the base run, as percentage differences ('p') from the base. Type the indicated letter and press [Enter] to show your choice. (If you are listing only 1 data bank or a single run of a model, it does not matter how you answer this question.)

You will then be asked for the name of the stub file and you reply with the name of the stub file you have prepared. (Use the full name, including the.STB.) Finally it asks for the name of the output file. We advise using a.OUT extension for these files. When Compare has finished and given the DOS prompt again, you can use the DOS Print command to obtain a printed copy.

Here is a sample session with Compare. You begin with the DOS command "compare". After the initial sign-on message, a dialog went like this. The user's responses are in italics.

```
How many alternatives?  2
Alternative 1:
Bank type (w=workspace,c=compressed,h=hashed bank,d=dirfor,v=vam):v
Root name of bank: dyme
Alternative 2:
Bank type (w=workspace,c=compressed,h=hashed bank,d=dirfor,v=vam):w
Root name of bank: cmdm
Show alternatives as (a)ctual,(d)eviations, or (p)ercentages? a
With what stub? test.stb
Name of output file: test.out
Now making the table as: test.out
```

As an alternative to answering the individual questions asked by Compare, you can put the answers into a file such as COMPARE.IN and start the program with

```
compare compare.in
```

Compare assumes that a filename given on the command line is a file containing input responses. The example below shows the response file COMPARE.IN that would correspond to the previous example.

```
2
v
dyme
w
cmdm
a
test.stb
test.out
```

Matrix Listing

A "matrix listing" displays all of the cells in a row or column of an input-output table. The command in the stub file is simply

\matlist <sectors>

For example, it could be

\matlist 1 5 7 15-30 (21 23-25)

where we have used the now-familiar system of indicating a list of sectors.

The "\matlist" command should be preceded by two related commands, "\row" or "\column" and "\cutoff". Here is a complete example of a stub file for a matrix listing:

```
\date 1987 1988 1989 1991 1993 1995 2000 91-95 90-100 95-2000
\ti MUDAN MATRIX LISTING
\9 1 66 1 3 33
\row
\cutoff 0.02
\matlist 18-25
```

This will produce a row listing; a cell of a matrix must account for at least .02 of the total (two percent of the row total) in order to be listed. Rows 18 - 25 will be listed.

The \ matlist command causes Compare to look for a file which at present must be called "matlist.cfg". Here is an example from Mudan.

```
Matrix listing identity; out = am*out+cr+cu+bmv*capital+vin+exp-imp+othdm
# Title file name for the rows of out, the lefthand side vector
out; "sectors.ttl"
# Title file names for matrix columns
am; "sectors.ttl"
bmv; "bmv.ttl"
# headers for each term
header for out;            "Output"
header for am*out;   "Intermediate"
header for cr;             "Rural Consumption"
header for cu;             "Urban Consumption"
hdr for bmv*capital; "Investment"
header for vin;            "Inventory"
header for exp;            "Exports"
header for imp;            "Imports"
header for othdm;          "Other demand"
```

In this file, all lines beginning with a # are comments, and anything before a ';' on a line is also a comment. The first line that does not begin with a # must be the matrix listing identity. It specifies, in terms of the particular model, an input-output identity. Usually this identity says, in effect, total output = intermediate demand plus final demand. But other identities are possible. The identity must have a single vector on the left and then on the right an expression that is the sum of a number of terms. Each term may be either a single vector, like cr and cu in the example, or may be a matrix*vector product, such as am*out and bmv*capital. The terms should be joined by + or - signs. The matrix*vector terms result in a display of all flows obtained by multiplying each column of the matrix by the corresponding element of the vector.

Following the identity, the next non-comment line must name the title file name for the vector on the left hand side of the identity. The file name must be enclosed in quotation marks (""). This title files can be the same one used with Vam, but Compare skips the 10-letter abbreviation at the beginning

of each line which is used by Vam in the show command. Rather, Compare looks for a string within " marks, and uses that. There can be other material in front of or after the quoted string. Here are the first few lines of the sectors.ttl file used in the above example.

```
Agricul     ;1   e "Agriculture"
Coal        ;2   e "Coal"
CrudeOil    ;3   e "Crude Oil & Natural Gas "
```

There must also be file names for the titles of the columns of every matrix in a row listing or the row of every matrix in a column listing. In the example they are sectors.ttl and bmv.ttl. These files should be of the form as just explained with the titles between quotes.

Finally, for each term in the identity, MATLIST.CFG must show the name to be listed with the term. These term names are also show as strings within quotes. For a vector term, this name will be listed on the left. For a matrix*vector term, it will be centered above the display of the cells of the matrix.

The last page of this manual shows the first page of the matrix listing produced by our example.

Seller: 18 Non-electrical Machinery

	1987	1990	1995	2000	90-95	95-00
			Sales to Intermediate			
3 Crude Oil & Natural Gas	20.2	37.1	42.0	45.8	2.49	1.76
14 Chemicals	24.8	60.9	93.0	138.4	8.46	7.96
15 Building Materials	24.7	52.3	77.8	108.8	7.95	6.71
16 Metallurgy	46.0	91.7	124.0	161.6	6.03	5.31
18 Non-electrical Machinery	283.6	376.0	538.6	750.3	7.19	6.63
19 Transportation Equipment	43.4	102.7	138.3	203.8	5.95	7.76
20 Electrical Machinery	31.5	74.7	112.7	177.1	8.21	9.05
25 Construction	109.9	164.3	228.8	290.3	6.63	4.76
31 Education, Health, and Scienc	19.2	37.7	56.6	89.0	8.14	9.07
SUM: Intermediate	757.4	1314.5	1856.4	2576.7	6.90	6.56
			Sales to Other Final Demand			
Rural Consumption	80.9	83.3	139.4	205.3	10.30	7.75
Urban Consumption	36.5	43.0	84.5	158.2	13.49	12.56
			Sales to Investment			
1 Agriculture etc.	12.3	12.3	20.5	30.8	10.14	8.16
11 Electricity etc.	39.3	39.7	56.5	86.0	7.07	8.42
14 Chemical Industry	30.7	25.3	41.1	62.6	9.66	8.42
16 Metallurgical Indust	32.3	21.3	36.2	54.7	10.61	8.24
25 Trans.& Communication	42.6	31.0	60.8	92.6	13.48	8.43
27 Public utility and service	22.4	12.8	22.8	34.7	11.59	8.42
35 Other state-owned units	33.1	17.2	34.1	56.5	13.66	10.10
36 Urban collective-owned units	30.4	14.1	31.1	53.3	15.91	10.77
37 Rural collective-owned units	61.3	31.5	57.3	92.3	11.97	9.54
38 Rural individuals	117.6	76.1	102.3	126.9	5.92	4.31
40 Balancing item	27.1	17.4	22.2	33.8	4.82	8.42
SUM: Investment	660.9	458.7	726.4	1094.5	9.19	8.20
Inventory	70.7	51.4	72.8	93.8	6.94	5.07
Exports	155.8	357.1	502.8	648.6	6.84	5.09
Imports	385.9	500.9	771.7	1124.1	8.64	7.52
Other demand	19.3	42.2	42.2	42.2	0.00	0.00
Output	1398.6	1854.0	2656.1	3700.2	7.19	6.63

Seller: 19 Transportation Equipment

	1987	1990	1995	2000	90-95	95-00
			Sales to Intermediate			
1 Agriculture	5.6	12.6	15.8	21.7	4.52	6.33
18 Non-electrical Machinery	13.0	33.8	44.6	65.6	5.55	7.71
19 Transportation Equipment	90.1	131.7	172.8	248.5	5.44	7.26
23 Equipment Repairing	17.5	39.0	51.7	75.0	5.61	7.44
25 Construction	18.0	32.5	40.7	53.1	4.47	5.35
26 Freight Transportation	29.9	68.6	85.1	120.5	4.32	6.96
27 Commerce	6.9	12.9	15.6	21.1	3.89	5.99
29 Passenger Transportation	13.0	26.3	35.3	58.3	5.87	10.03
31 Education, Health, and Scienc	5.5	13.1	17.7	28.7	5.98	9.66
33 Public Administration	5.4	13.4	15.8	21.5	3.35	6.08
SUM: Intermediate	235.7	461.1	590.4	849.1	4.94	7.27
			Sales to Other Final Demand			
Rural Consumption	7.8	9.7	14.6	22.2	8.15	8.34
Urban Consumption	4.8	6.9	12.1	23.4	11.34	13.15
			Sales to Investment			
1 Agriculture etc.	4.5	5.5	8.2	12.7	7.98	8.75
11 Electricity etc.	14.5	17.7	22.6	35.4	4.92	9.01
14 Chemical Industry	11.3	11.3	16.4	25.8	7.50	9.01
16 Metallurgical Indust	11.9	9.5	14.5	22.5	8.46	8.83
25 Trans.& Communication	15.6	13.7	24.1	37.8	11.32	9.02
27 Public utility and service	8.3	5.7	9.2	14.4	9.44	9.01

Index

arith() 65
ASCII files
  reading from 13
  writing to 32
Callall.cpp 36, 39
Compare 7, 76
  & 77
  @csum() 78
  \cutoff 78
  \date 76
  commands 77
CXL windowing library 60
Data
  to introduce 13
depad() 61
Detached-coefficient equations 42
Dyme 6
Dyme functions
  callall() 67
  CheckDeclar 63
  depad() 61
  getopt 60
  lastdata() 63
  store(t) 67
  storets() 67
Equation file 42
Equation function 44
Equation object 44, 62
  functions 45
Exogenous variables
  how to specify 52
Files
  .fin (fix index) 53
  .mfx 47
  .stb 76
  .vfx 53
  callall.cpp 39
  config.cpp 58
  dyme.cpp 58
  Dyme.inc 68
  Dymesys.h 68
  load 24
  Features.h 58, 61
  heart.cpp 38
  heart.h 38
  macfixer.cfg 47
  matlist.cfg 81
  pseudo.sav 37
  tseries.inc 38
  vamtog.cfg 34
fix vector in vam.cfg 53
Fixer 5, 53
  gro 55
  group 54
  ind 55
  mul 55
  ovr 54
  stp 56
Fixes 47
Fixes, vector
  types 54
G 4
G7 commands 12, 13
  add 25
  addtype 17
  arguments in 25
  coef 31
  ctrl 30
  data 14
  diagextract 30
  diaginsert 30
  do 26
  fadd 25
  fdates 23
  flow 31
  fvread 20
  getsum 31
  glist 27
  group 27
  index 28
  ipch 42
  lint 28
  listvecs 33
  matdat 15

matin 20
matin5 21
matup 16
monup 15
pmatin 22
pmatin1 22
pmfile 23
pmpunch 32
punch 42
punch5 32
punchvec 32
ras 30
show 24
store 25
update 15
vam 12
vamcreate 12
vdata 15
vf 23
vmatdata 17
vp 33
vupdate 15
wrindex 17
zap 17
Groups 3, 5, 27
dynamic group 27
in "lint" command 28
in Fixer 54
Vector csum() function 71
Heart.cpp 36
Heart.h 36
Hist.bnk 36
Idbuild 5, 36
iadd command 38
Installing the software 8
interpolation 28
Lagged values 4, 11, 12, 40, 67, 72, 73
Lastdata
file 63
function 63
function of Vector 71
load function
in simulation model 64
load() 64
Macfixer 47
MacroFixer 5
Macrofixes
and exogenous variables 52
cta 48
gro 49
ind 48
mul 48
ovr 48
rho 49
skip 48
stp 49
to display 52
Make file 68
Matrix balancing 30
Matrix fixes 53, 54
Matrix functions 70
Matrix listing
how to make 80
missing values 28
modify() 38
Mudan 7
Multiple vam file 68
OpenVam() 68
Output fixes 57
Packed matrices 21
in vam.cfg 12
Pseudo.sav 37, 38
Rho adjustment 3, 6, 36, 38, 39, 43, 45, 49, 64, 67
Seidel solution 62, 74
setrho 64, 67
Simulation program 58
Slimdyme 8, 59
store() 67
storets() 74
Stub file 76
Triangulation ordering 74
Tseries 39
Tseries.inc 36
Tserin() 39
upets() 64, 74
useall flag 65
Vam Configuration File 11

vam file 4
Vam.cfg 4, 11
Vam2vam 34
VamFile object 68
VamtoG 34
Vector fixes 53
Vector functions 70