An Overview of Interdyme

Douglas S. Meade
January 18, 2016

Origins

Since the mid 1960s, INFORUM has been developing and refining macroeconomic interindustry models. These models include various vintages of models of the U.S. and over a dozen other countries. Parallel to this model development has been the creation of software and systems of programs for building models. In particular, Inforum has developed two major software applications that work together. The first is the *G7* regression software, which can also be used to develop databanks. The other is *Interdyme*, which is used to build interindustry macro models.

G7 was first developed for building quarterly and annual macro models. A sister program called *Build* takes output files written by *G7* and writes C++ code that can be compiled into a running model. *Interdyme* endeavors to apply the lessons learned with *G7* for building macro models to the process of building macroeconomic interindustry models. It borrows much of the capabilities of *G7/Build* for the macro elements of the model, and introduces the power of C++ object-oriented programming to the handling of matrices, vectors, equations and simulation output files. *G7* can be used to develop a collection of vectors and matrices needed to build a model. Equations for both vector and macroeconomic variables can be estimated in *G7*, and then incorporated into an *Interdyme* model in a standardized framework[1].

At this time, Interdyme models have been constructed for the U.S. (*LIFT*, *Iliad* and *STEMS*) and many partner countries. System development is ongoing at INFORUM/USA, and updated software can be easily distributed to partner groups. In turn, we incorporate into *Interdyme* ideas and suggestions from our partners, leading to benefits that can be shared among the entire group of partners.

What Is Interdyme?

Interdyme is a system of programs for building ***Inter***industry ***Dy***namic ***M***acro***e***conomic models. Interdyme concentrates on data development aspects of such models and leaves the model builder with more time to devote to the formulation of model structure, equation estimation and simulations. Unlike other model-building software such as GAMS, Matlab or Eviews, *G7* and *Interdyme* work together to produce compilable C++ code, which results in models that run much faster, and can be debugged and tested with many C++ compilers. There is virtually no limit to the size of an Interdyme model. There are very few programs required in the entire *Interdyme* framework. These are listed below:

- *G7* – is used to create a single file, called a Vam (vector and matrix) file, which contains time series of vectors and matrices of the model, whether they be assumed or calculated in the model. *G7* puts into this file historical values of these variables and exogenous projections of some of them. (Filling in future values of the endogenous vectors and matrices is the

[1] The subjects treated in this overview are covered in more detail in *The Craft of Economic Modeling, Part 3*. Relevant sections are referred to in this text.

business of the *Dyme* model program, described below.) *G7* allows for graphical or spreadsheet display of data, as well as the capabilities to read or write data to and from ASCII files. *G7* can also be used to project matrices forward using across-the-row indexes, it can scale a matrix to controls using the *rAs* technique, it can control groups of elements of a vector to a pre-defined control total, and it can interpolate missing data. *G7* is used for organizing and creating vector and matrix data, as well as for viewing results of simulations. *G7* is also used to estimate behavioral equations, as well as to manage databanks of data necessary to create macrovariables, vectors and matrices. (Macrovariables are any variables that are not vectors or matrices.)

- *IdBuild* - This program is used to write out functions that implement macro variable equations, much as they are written with the *Build* program for aggregate models.
- *Fixer* - The program implements the input of "fixes" for vectors, in a format much like the "data" statement in G. A "fix" is either an exogenous projection or a modification of a behavioral equation in the model.
- *Macfixer* - This program implements fixes for the macrovariables in the model.
- *Dyme* - This is the simulation program, which must be written by the model builder. *G7* and *IdBuild* prepare some of the components that comprise the simulation model, and the fixes read by *Fixer* and *Macfixer* are applied as the model runs. The principal output of *Dyme* is a *Vam* file with calculated values of all the endogenous vectors and matrices, and a G bank with the macro variables.
- *Compare* - This program can be used to make attractive tables of either historical or forecast data, as well as to show some matrix input-output relationships.

Here are some other general features of the *Interdyme* system:

- Interdyme models are capable of either historical simulations or forecasts. There is no restrictive "starting year" which must be constrained by the availability of historical data, equation estimation interval, etc.
- High level C++ classes for model building are available, including Matrix, Vector, Equation, Tseries (macrovariables), GBank and VamFile. These classes encapsulate the behavior of commonly used components in a well-defined way.
- A natural syntax for matrices and vectors can be used in the model code. For example, if b and c are vectors, and A is a matrix, one can write
  ```
  b = A*c;
  ```
  as a single line of code in an Interdyme model.
- In general, model code is concise and understandable. Macro variable functions written by *Idbuild* parallel the contents of the *G7* .sav files used to create them, and all macrovariable names and equation parameter values are shown transparently. Equations for vector elements, (such as employment by industry) must be written by the model builder, but follow a common template, or pattern.
- A simple model called "Tiny" is distributed with the Interdyme model source code. This is the code needed to build a very simple model in Interdyme, and is based on a 8-sector I-O table.

Planning an Interdyme Model

Perhaps the most important step in successfully building an Interdyme model is careful planning. What matrices and vectors will be required in the model? Over what horizon should the model forecast, and how far back should historical data be maintained? What will be the sectoring scheme of the model, and what will be the base year of the constant price data? What macro variables will be required? What are the important exogenous variables, and which variables should be endogenous?

While in the planning stage, it is helpful to develop some preliminary files for the model. These files will be needed later, and they help one to think out the model structure. These files are the vam.cfg file and the titles files (extension .ttl).

The central program of Interdyme for data development is *G7*. The vector and matrix data file this program operates on is called a Vam file, and the structure of the file is defined by the file vam.cfg (cfg is for "configuration"). The vam.cfg file is a list of all the matrices and vectors which will be included in the Vam file. All of these matrices and vectors will be available for use in the forecasting model. The vam.cfg file for the Tiny model is shown in Figure 1.

Figure 1. Vam.cfg File for the Tiny Model

```
# vam.cfg for the tiny model, a very simple model based on an aggregated
#   8-sector IO framework
# Years of the vam file.
1995 2020
# Name  |Number of  |Files of titles of|   Description
#       |row col lag|  rows   cols     |
FM         8   8  0  sectors.ttl sectors.ttl #Input-output flow matrix
AM         8   8  0  sectors.ttl sectors.ttl #Input-output coefficient matrix
LINV       8   8  0  sectors.ttl sectors.ttl # Leontief inverse
out        8   1  3  sectors.ttl # Output
pce        8   1  0  sectors.ttl # Personal consumption expenditure
gov        8   1  0  sectors.ttl # Government spending
inv        8   1  0  sectors.ttl # Investment
ex         8   1  0  sectors.ttl # Exports
im         8   1  0  sectors.ttl # Imports
fd         8   1  0  sectors.ttl # Total final demand
# Value added
dep        8   1  0  sectors.ttl # Depreciation
lab        8   1  0  sectors.ttl # Labor income
cap        8   1  0  sectors.ttl # Capital income
ind        8   1  0  sectors.ttl # Indirect taxes
```

The dates at the top of the vam.cfg file should be 4-digit dates, and should span the interval of the historical data and the allowable simulation period for the model. The rest of the vam.cfg file consists of a list of matrices and vectors that will be contained in the Vam file, and available to be used in the forecasting model. For each matrix or vector, the name, number of rows, number of columns, number of lags, and titles files are shown. Any text following a '#' character on a line is treated as a comment. The name of the matrix or vector is the name as it will be known in *G7* and in the model. Row and column sizes are self-explanatory, except that vectors are generally column vectors (i.e., with one column). The titles files are used by *G7* to display short sector titles when viewing the data in spreadsheet format, with the *G7* "show" command.

The standard titles file, which shows the sectoring of the input-output table, is named sectors.ttl. A part of the file for the Slimdyme version of the Tiny model is shown in Figure 2.

The names at the left are abbreviations for the sectors, which will appear when headings are needed in the spreadsheet display of vectors and matrices in *Vam*. This file is in fixed format, with 12 columns required before the semicolon. The letter 'e' must be in column 18. The sector number, in columns 14 to 17 is optional, but recommended. The sector title follows, enclosed in quotes. In this file, one can also include documentation to the right of the sector title, if so desired. This could include classification definitions, comments about the sector, or other information.

Figure 2. Sectors.ttl File for the Tiny Model

```
Agricul     ;1   e   "Agriculture"
Mining      ;2   e   "Mining and quarrying"
Elect       ;3   e   "Electricity and gas"
Mfg         ;4   e   "Manufacturing"
Commerce    ;5   e   "Commerce"
Transport   ;6   e   "Transportation"
Services    ;7   e   "Services"
Government  ;8   e   "Government"
```

The G7 Program

Once a vam.cfg file and the necessary titles files have been created, you can begin to build a Vam file of matrices and vectors. A Vam file is not the same as a *G* bank, but holds only matrices and vectors listed in the vam.cfg file. Whenever you create a new vam.cfg file, or make alterations to an existing vam.cfg file, you should recreate your .vam file in *G7*. A common name for the starting vam file that holds historical data is hist.vam. The *G7* command to create this file from vam.cfg is “vamcreate”, abbreviated “vamcr”. The following sequence of commands creates the vam file, assigns it to position a, and sets it as the “default vam file”.

```
vamcr vam.cfg hist
vam hist a
dv a
```

G7 writes out the initial empty hist.vam file for the list of matrices and vectors over the time interval specified in vam.cfg. You can use the "listvecs" (“lv”) command in *G7* to see the current list of matrices and vectors. If you type "show <vecname>", where <vecname> is one of the available vectors, you will see a spreadsheet with sectors down the row, and years across the column headings. Until you supply data to the vector, this should be all zeroes.

Some of *G7’s* most important capabilities are listed below. More information can be found in the Inforum Help Documentation.

1. *G7* uses a simple interactive format, with short, easy-to-remember commands. Files of commands can be executed in batch mode using the "add" command.
2. Data for either vectors or matrices can be read in from ASCII text files, in a variety of formats, including formats that could easily be exported from Lotus or Excel. In addition, data can be read in directly from Excel
3. G banks of all types can be assigned, so that vector or matrix data that has already been organized in *G7* can be easily transferred to a Vam file.
4. Data can be "massaged" in a number of ways. Vectors or matrices can be indexed, or moved forward by the growth of another vector or series. The *rAs* technique can be used to balance a matrix to known row and column controls. Whole vectors, or selected groups of sectors can be forced to sum to a control total. A powerful vector calculate ("vc" command) feature allows for matrix and vector calculations using a natural syntax.
5. Data can be viewed or edited in a convenient spreadsheet format, using the "show" command.
6. Graphs can be made of vector or matrix elements, and either printed or saved as a .wmf file for inclusion in a document.

7. Once data have been organized into a Vam file, attractive tables can be made using the *Compare* program. For those already familiar with *Compare*, the stub files for Vam file data are identical to those for printing the same data from G banks. *Compare* can also make matrix listings using matrices and vectors from a Vam file.
8. Vector and matrix data can be saved to ASCII text files, for use in other programs.
9. G banks can be created, using the “workspace" feature.

Macrovariables and Idbuild[2]

A macrovariable in Interdyme is any variable that should logically be treated as a single time-series. For example, a variable such as an interest rate that will be predicted with a behavioral equation is a macrovariable. Its equation will be estimated in G7, and incorporated into the model as described below. Likewise, exogenous variables such as population, the money supply, and tax rates are macrovariables. Projections of these variables can be read into a G databank which is used with the simulation program, or have fixes applied to them by using the *Macfixer* program. Aggregates of vector variables, such as total consumption, total investment, etc., are also conveniently stored in macrovariables. The many relationships inherent in the numerous identities in the national accounts can be implemented using macrovariables.

Macro variable equations are estimated in *G7*, and results are saved to .sav files, which capture output from *G7* relating to transformations of variables, and regression results. For example, the following simple add file for *G7*, DISINC.REG, estimates a regression for total disposable income based on total wages:

DISINC.REG

```
bank tiny                  # assign a G data bank
lim 1982 2010              # set limits of the regression
save disinc.sav            # start saving equation results
r disinc = totwag          # do the regression
save off                   # close the save file of equation results
gr *                       # view a graph of the regression fit
```

This creates the following .SAV file:

DISINC.SAV

```
r disinc = 20191.218824*intercept +
          1.144324*totwag
```

The program *Idbuild* is run with a master add file, usually called MASTER, which includes many "iadd" (Interdyme "add) commands to bring the various .SAV files into the model. Most of the .SAV files contain behavioral equations. For each equation, C++ code is generated to create a callable function that will calculate the variable according to the results in the .SAV file. The name of the function will be the root name of the .SAV file with an 'f' appended. For example, the above equation is written out (in a file called HEART.CPP) as:

```
void disincf()
```

[2] This topic is discussed in section 14.7 of *The Craft of Economic Modeling, Part 3*

```
{
/* disinc */ depend =20191.218824+1.144324* totwag[t];
        disinc.modify(depend);
    }
```

The function disincf()can be called from anywhere in the forecasting program, when the variable disinc needs to be calculated. Note for now that the variable totwag is referred to with the index t. This variable t is always the value of the current forecasting year in Interdyme, and indexing a macro variable with that variable retrieves the value for the current forecasting year. Likewise, totwag[t-1] refers to the lagged value. The macro variable calculated in the above function is called disinc. After the calculated value from the equation is stored in depend, the modify() function is called, with depend as its argument. This function applies fixes to macro variables, and rho-adjustments.

In addition to the .sav files for behavioral equations, *Idbuild* also recognizes a special file called pseudo.sav. This file does not contain equations, but introduces data for macro variables that are exogenous or will be determined in the model by identity.

Even with a fairly large macro component of a model, *Idbuild* runs fairly quickly. When it finishes, it has created the files listed in Figure 3. Whenever any new macro variable equations are estimated, or new variables are added to PSEUDO.SAV, *Idbuild* should be run again, to re-generate the files in Figure 3. Then the model should be recompiled, as described in the section below on the forecasting program.

Figure 3. Files Created by *Idbuild*

tseries.inc	A C++ include file that declares all the macrovariables in the model as type Tseries. This file will be included as part of the simulation program.
heart.cpp	A C++ program file that contains the functions for all macrovariable behavioral equations and identities introduced as .sav files in *IdBuild.* This program module becomes part of the simulation program.
callall.cpp	A C++ program file that contains the function callall(), which calls all of the macrovariable equations. This program model also becomes part of the simulation program
hist.bnk hist.ind	A G databank, containing all the macrovariables needed for behavioral equations, as well as all macrovariables in pseudo.sav.

Estimating a Sectoral Equation in Interdyme[3]

Since an Interdyme model contains equations for employment, investment and many other variables at a detailed industry or category level, the estimation of sets of similar equations is important. For example, it is convenient to estimate employment for all industries of a model in the same *G7* add file. (Frequently, only a small number of different types of equations are used, even though there are equations for many sectors.) In this case, it becomes advantageous to save the equation parameters for all sectors to a file, called an equation file. These files generally have the extension .eqn.

[3] This topic is covered in section 14.10 in *The Craft, Part 3*.

In the forecasting model, these equation parameters are read in at run time, so that they do not need to be a compiled part of the model. In this way, one can re-estimate and test equations without having to recompile the model.

In *G*, the creation of the equation file is accomplished with the "punch" and "ipch" commands. A typical add file to estimate the a set of equations for employment might look like the following:

```
ba employ
add mkvar.add  # calculate some variables needed in employment equations
punch employ.eqn 8 7 2010
add empa.reg 1 "Agriculture"         # type a equation
add empb.reg 2 "Mining and quarrying"            # type b
add empc.reg 3 "Electricity and gas"             # type c
add empb.reg 4 "Manufacturing"       # type b
.
.
punch off
```

Note the "punch" command in the above example, which opens the equation file employ.eqn for writing, with up to 8 equations, with up to 7 estimated parameters each, with estimation period ending in 2010. There are 3 different types of equation used for employment, called types 'a', 'b', and 'c'. Each type uses a different *G* add file to estimate a regression. The file empa.reg is:

```
# EMPA.REG -- Version "a" of the employment regressions.
ti %2
r emp%1 = out%1, out%1[1], out%1[2]
gr *
ipch emp %1 a
```

The %1 and %2 in the above file are add file command line arguments, which in this case are the sector number and the title. After the regression has been calculated, the "ipch" command writes the equation parameter information out to the equation file. Figure 4 below shows a sample equation file. The first equation indicates that this is a type 'a' equation for sector 1, and there are 4 estimated parameters. The first number shown on each line is the estimated value for rho.

Figure 4. A Few Lines of an Equation File

```
8 7 2010
emp 1 a 4
 1 2 3 4
     0.319757     -1.65843   -0.0458105  -0.00602098   -0.0147023
emp 2 b 4
 1 2 3 4
     0.750327      2.48859    0.0411307    -0.108004   -0.0694493
```

The equation file is then read into the model as an Equation object, described in the section on Interdyme classes. The Equation object contains the equation parameters, the number of equations, the values for rho for each sector, and other information.

Writing the Simulation Program: DYME.EXE[4]

"Building" is a word often applied to the process of developing models. Perhaps this is because building models is much like building a house. You must first draw plans and make measurements. The general infrastructure, such as plumbing, electricity, network and phone, must be supplied. After surveying, you lay the foundation, and then build up from there. You put in the expensive wood paneling after the windows and roof are in place.

A set of program files that implements a simple Interdyme model, called Tiny, provides a general plan for the program structure of the model. The model builder works on and elaborates the program file model.cpp. The other files contain the "plumbing and electricity" for your model. You shouldn't need to change them unless you have ambitions to be a plumber or electrician.

The Tiny model includes the variables output, final demand, imports, total value added and prices, using 8-sector input-output accounts. It performs a Gauss-Seidel output and price solution, calculating output and imports from final demand, and prices from value added and output. It is liberally peppered with comments indicating where the various sections of a typical Inforum model would go.

If you plan to build a interindustry macroconomic model with Interdyme, perhaps the best place to start with Tiny is to get your data for the above variables into the Vam file, and try to get the tiny model to work with your data. You may have to edit the section which declares the matrices and vectors of the model. Note that these are of type Matrix and Vector, which are types of objects in Interdyme.

In Tiny, this section consists of the following lines in MODEL.CPP.

```
// Matrix and Vector declarations should go here:
// Vectors

out.r("out");pce.r("pce");gov.r("gov");
inv.r("inv");ex.r("ex");im.r("im");fd.r("fd");dep.r("dep");
depc.r("depc",'y','n'),lab.r("lab");labc.r("labc");cap.r("cap");
capc.r("capc"),ind.r("ind");indc.r("indc");pcec.r("pcec");
invc.r("invc"),govc.r("govc");exc.r("exc");imc.r("imc");
x.r("x");
y.r("y");

// Matrices
AM.r("AM");
```

The name in quotes in the above declarations is the name of the matrix or vector in the Vam file. The characters after the name, if they are present (see depc) are the read/write flags. Using these flags, the model builder has complete control over whether a matrix or vector will be read from or written to the Vam file during each year of the simulation. Depreciation coefficients ("depc") are read but not written, since they are not changed by the model.

On your first attempt with Tiny, you should also estimate a set of import equations for the model. The form of this equation should be a linear regression of imports on domestic demand, defined as output plus imports, or intermediate plus final demand before subtracting imports. Then, assuming

[4] The simulation program, DYME.EXE, is introduced in pages 46-53 in *The Craft, Part 3*.

your import equation file is called IMPORTS.EQN, the following line of code in MODEL.CPP initializes the `Equation object:`

```
Equation ImportsEq("imports.eqn");
```

This import equation object is then passed to the Seidel function, which calculates output and imports in the Gauss-Seidel algorithm:

Seidel(A,out,im,sdc,**ImportsEq**,fd,triang,toler); // get gross domestic output

The other arguments passed to Seidel() are the A-matrix (A), output (out), imports (im), the discrepancy (sdc), final demand (fd), a triangularization vector (triang), and an optional tolerance for convergence (toler).

As mentioned above, as a model builder, you will do most of your programming in the file model.cpp. Specifically, most of your work is within a function called loop(). In addition to this, you must usually write functions to implement the equation calculation of each set of vector equations, such as consumption, investment, exports or employment.

Figure 5. Overview of Interdyme loop() function

```
void loop() {
    // Declare Matrix and Vector objects.
    // Declare Equation objects.
    for(t=godate;t<=stopdate;t++) {
        // Load data (and fixes) for this year.
        while( not converged) {
            // Calculate final demands.
            // Call Seidel() to calculate output.
            // Calculate productivity and employment.
            // Calculate wage rate and total wages.
            // Calculate other value added categories.
            // Call Pseidel() to calculate prices.
            // Perform aggregation, identities and macrovariable equations.
            }
        // Call all equations again, with rho-adjustment error calculation.
        // Store data.
        }
    } // end of model.
```

Figure 5 summarizes the structure of the loop() function, which is important for understanding the structure of an Interdyme program from a bird's eye view. The main part of this function is a loop over the years of the simulation, which run from godate to stopdate. Before this loop some data must be declared, which will be used throughout the forecasting model. This includes matrices and vectors which will be read from and written to the Vam file, as well as any Equation objects used in the model. At the beginning of each year, the data from the Vam file is automatically read in by the load() function, which also loads any fixes are present (these are in a special Vector, called "fix").

Final demand categories, such as consumption, equipment and structures investment and exports are calculated first, using vector equations. These are summed to yield total final demand, before imports have been subtracted. This vector is then passed to the Seidel() function. The Seidel() function calculates outputs and imports, and if necessary, calculates or applies a discrepancy during the Seidel calculation. Next employment by industry is calculated, using the outputs calculated in Seidel(). A wage rate function generally is the first function calculated on the price/income side. From wage rates and employment, we can calculate total wages. Other equations come next, for categories of value added such as profits, depreciation, rent and indirect taxes. Pseidel() is the function called to perform the Seidel calculation of prices from unit value added. Finally, aggregate prices, other aggregates, and some macrovariables are calculated.

Within each year of the model, the model usually needs to iterate before reaching convergence, if it is not strictly recursive. For example, in a typical model, consumption is based on income and prices, which are not calculated until the income/price side of the model is finished. On the first iteration, guesses must be made for these right hand side variables. They can be started at the previous year values, or they can be trended forward from two previous years. Consumption is calculated with these first guesses of these variables first. After the entire model is finished for that iteration, it returns to re-calculate consumption, using the just-calculated values of income and prices. It repeats this sequence until converging on some appropriate variable or set of variables.

The Classes of Interdyme

You have already been introduced to a few of the classes of Interdyme, such as Matrix, Vector and Equation. These are the classes you will work with most directly. Another class, called Tseries, is used in the implementation with macrovariables. Pmatrix, which means "packed matrix", is another matrix class, which is convenient for working with large, sparse matrices. Finally, deep in the "plumbing" of Interdyme are classes called GBank, VamFile, FixBank and MacFixBank that handle the infrastructure of working with G databanks, Vam files, vector fixes and macro fixes.

What exactly is a "class"? A class in C++ defines a new type of data. You can think of the predefined data types in C++, such as integers (int), real numbers (float), and characters (char) to be classes, in a sense. They are simple, scalar objects, but already the compiler must know that a ***** sign between two floats means to do something different with the binary representation than a ***** sign between two integers. In a similar way, with a class Matrix and Vector, we can define a new "overloaded" version of the * operator that means to do one operation when both operands are of type Matrix, and a different operation with Matrix and Vector.

A C++ object consists of a set of data, along with functions and operators defined to operate on that data. For example, the Vector class contains a function called fix(), which is called to impose fixes on the Vector for the current year. If the vector were emp, and the current year t, applying the fixes is done with the following line of code:

```
   emp.fix(t);  // Apply fixes to employment.
```

To obtain the sum of all elements of a Vector, a sum() function is defined. If totemp were a macrovariable (class Tseries), then we could store the sum of employment vector as follows:

```
   totemp[t] = emp.sum();
```

In fact, the subscript operator [] on a macrovariable is an overloaded operator. Usually the year t is a four-digit year, like 2015. You can be assured that totemp does not have 2,015 elements contained in it! Rather, the Tseries object type knows that if 1990 is the base year of the model, then the year 2015 is in position 25 of the data component of totemp (which starts at 0 position). In this way, we can assign use the current or lagged values of a macrovariable in a natural, easily understandable format. The Tseries object also has a function called modify(), which applies fixes and rho-adjustments. This is called by every macrovariable equation function written by *Idbuild* in heart.cpp, but can also be inserted by the model variable for any macrovariable, anywhere in the model code.

The Equation object also has a subscript operator defined for it, so that it can be addressed just like a two-dimensional array in C++. If you have declared an Equation object called imports, then imports[i][j] is the j'th parameter for the i'th equation. The following section, which shows a typical equation function, shows some of the other functions of Equation.

While the many classes in Interdyme may seem daunting at first, each of them has been devised to make the writing of models easy and natural.

A Typical Equation Function in an Interdyme Model[5]

Most of the time you spend building a model will probably be devoted to estimating and coding the various behavioral equations for the vectors in the model, such as components of final demand, value added, employment and wage rates. These equations generally follow a common pattern, or template. Once you have struggled through writing the first equation function, the others are much easier.

This section will present the code for the simple employment by industry equation presented earlier. This is not the simplest function that could have been shown, but it is short, and provides a good example of how to handle lagged vectors in Interdyme.

A lagged vector in Interdyme is handled in the following way. First, both a Vector *and* a Matrix are declared with the same tag name "out". The two variables must be given two distinct C++ variable names, say out and Out. As we shall see below, the Matrix version of output can be used to access lagged values. The first subscript of this matrix is [0] for the current year value, [1] for the previous year, and [2] for the year before that. In other words, out[k] will be the same as Out[0][k], for all industries k. Out[1][k] represents output for industry k, lagged once.

The vector "emp" must also be declared, and an employment Equation object must be initialized. This is done with the following lines of code, near the top of the loop() function.

```
Vector emp("emp"), out("out");
Matrix Out("out");
Equation empeqn("employ.eqn");
```

Further down in the loop function, the Seidel() function is called, to calculate output and imports. After Seidel is finished, the employment equation function empfunc(), can be called.

```
Seidel(A,out,im,sdc,imports,fd,triang,toler);  //  get gross output
empfunc(emp,Out,empeqn);
```

The employment function itself is included lower in the MODEL.CPP file, and is simply the following:

[5] This topic is covered on pages 63-66 of *The Craft, Part 3*.

```
/* empfunc() -- simple employment function */
short empfunc(Vector& emp, Matrix& Out, Equation& empeqn) {
   short n, i, k;
   float empcalc, empact, emp_rho;
   char  which;
   n = empeqn.neq; // number of equations

   for(i = 1; i <= n; i++){
        k = empeqn.sec(i);   // sector number k for each equation i
        if(k==0)  // no equation
          continue;
        which = empeqn.type(i);  // equation type
        empact = emp[k];    // save actual value for rho-adjustment

        if (which == 'a'){  // this is the type 'a' equation
           empcalc= empeqn[i][1] + empeqn[i][2]*Out[0][k]
              + empeqn[i][3]*Out[1][k] + empeqn[i][4]*Out[2][k];
           }
        else    {
           printf("Unknown equation type %c in empeqn, category %d.\n",
                  which, i);
           tap(); // Pause so that error message can be read.
           continue;
           }
        emp_rho = empeqn.rhoadj(empcalc,empact,i); // rho-adjustment
        emp[k] = emp_rho;  // save rho-adjusted value
        }
   emp.fix(t);  // apply fixes
   return(n);
   }
```

The equation function is passed three arguments, employment (emp), the Matrix of current and lagged outputs (Out), and the employment Equation object. The body of the function illustrates a typical pattern for calculating a vector equation. Here is an explanation of the logic:

1. Find the number of equations with `n = empeqn.neq.`
2. Loop through all equations, for each equation, determine the sector number, and the type of equation. This is done with the `Equation class functions type() and sec().`
3. Save the actual value for doing the rho-adjustment.
4. Calculate the equation with an expression parallel to its estimated equation in *G*. Note again the syntax for referring to lagged values of output.
5. *Rho-adjust*[6] the calculated value, using the Equation rhoadj() function, the calculated and actual values, and the equation number. Save the result in the vector element.
6. Apply the fixes, if any.

These steps are self-explanatory, except for the fixes and the rho-adjustment. They are described in the next two sections.

[6] See the section below on rho-adjustments for more details.

Fixes and How to Use Them[7]

Fixes serve the following purposes:

- Supplying values for variables which must be exogenous.
- Overriding the calculation of single behavioral equations.
- Modifying the calculation of single behavioral equations, either by addition or multiplication.
- Controlling a set or group of sectors in a vector to sum to, or move like, a certain control total.

For vector variables, fixes are applied using a program called *Fixer*. Fixer reads its input from a file, usually named VECTORS.VFX. The .VFX file can also contain group definitions, which can be used in *Vam* or in the specification of fixes. The following file may be used to apply fixes to our hypothetical 8-sector model.

Figure 6. Sample of Vector Fixes File

```
group Primary
   1-2
ovr emp 6
   1993 3.1 3.2 3.3
   2000 4.0;
ind emp :Primary
   2010 1.0 1.05 1.12 1.15 1.17 1.19
   2016 1.22 1.25 1.28;
```

The group definition establishes a named group, which can be accessed in most syntaxes by prefixing its name with a colon (':') character. For example, the group called "Primary" is defined to consist of sectors 1 and 2. The second fix in the file is an index (ind) fix on the total of primary sector employment. It should move like the index shown. When the *Fixer* program is run on this file, it reads actual data for 2010 from the Vam file, calculates the sum of manufacturing employment for this year, and then writes out a projection of manufacturing employment that moves like the specified index.

Where are the vector fixes written? To the Vam file itself, in a vector called "fix". In fact, if you want to use vector fixes, you *must* declare a vector "fix" in the VAM.CFG file, and it also must be declared at the top of the `loop() function:`

```
Vector fix("fix");
```

When the model is running, and control is in the employment equation function, the line of code

```
emp.fix(t);
```

applies any single-sector or group fixes to the elements of the emp vector. For example, the employment in the two manufacturing sectors will be scaled to control to the group fix on primary employment shown in Figure 6.

[7] The topic of fixes is covered in section 14.14 in *The Craft, Part 3*.

Macrovariables are fixed in a way similar to vector variables. The fixer program is called *Macfixer*, and the ASCII fix file is usually macrofix.mfx. This file usually contains projections of exogenous macrovariables, or modifications and overrides to macro equations. For example, the following lines of the fix file show how to project population (pop) growth at 1.8% per year, and to specify a projection for m2.

```
# population - grow at 1.8%
gro pop
2010 1.8
2020 1.8;
# money supply
ovr m2
2011    3617.800   3788.100   3907.300   4087.100   4283.700
2016    4498.900   4729.500   4972.000   5226.900;
```

The available types of fixes for both vector and macro variables are ovr (<u>ov</u>e<u>r</u>ride, or "actual"), ind (<u>ind</u>ex by link point), gro (exponential <u>gro</u>wth rate), stp (growth rate in <u>st</u>e<u>p</u>s), cta (add-factor or <u>c</u>onstant <u>t</u>erm <u>a</u>djustment), and mul (<u>mul</u>tiplicative factor). In addition, the macrovariables use rho (<u>rho</u>-adjustment fix) and skip (<u>skip</u> behavioral equation, use data in the G bank).

Rho-Adjustments on Vector and Macro Variables

Rho-adjustments are often desirable because of the well-known "jumping on" or "jumping off" problem. This occurs when the fit of an equation is not very close in the last year of estimation, and therefore not likely to be close in the first year of forecast either. In this case, viewing historical data beside forecast, we observe a distinct jump (or even a leap!) in the first year of forecast.

The rho-adjustment procedure works as follows:

1. In the starting year of the rho-adjustment (called the rhostart year), an error for the equation is calculated and stored. The error is added back to the calculated value, so that the forecast value for this year is really the actual value.
2. In forecast years, the equation is adjusted by

$$x_t = \hat{x}_t + \rho^n \varepsilon$$

where:

x_t is the final rho-adjusted value

$\hat{x}_t$ is the calculated equation value

ρ is the estimated value of rho, the autocorrelation coefficient of the residuals.

n is the difference in years between the forecast year and the rhostart year.

ε is the error calculated in the rhostart year.

With vector fixes, the value for rho is stored in the equation file. With macro fixes, it must be supplied as a rho fix.

In the simulation model, vector rho-adjustments are performed with the rhoadj() function of the Equation object, which stores the values for rho and the error for each equation in its data area.

Macrovariable rho-adjustments are performed with the Tseries modify() function, which is automatically called in all functions written by *Idbuild*. If you supply an exogenous variable in the fixes file, you must supply the call to modify in your program code, or the exogenous fix will not be applied. For example, the following code fixes the exogenous variables pop and m2:

```
float depend;
pop.modify(depend);
m2.modify(depend);
```

Making a Simulation with Interdyme

After changing program code in MODEL.CPP, or after making changes to the macrovariables by running *Idbuild*, the model must be recompiled and re-linked, with the C++ compiler. If you are using Borland C++, there is a "make" file already supplied with Tiny, that you can use with little or no modification, called DYME.MAK. To compile and link the model, type:

```
make -fdyme
```

from the MSDOS Command prompt. If you have errors, they will probably be in model.cpp, which you have been editing. Check for missing semicolons (';') at the end of statements, misspelled variables, quotes or comment characters that have no match, or mismatched braces. Another common error is either to forget to declare a variable, or to declare a variable as one type, such as a Vector, and then try to use it as a variable of another type, such as a Tseries or an Equation.

A file called DYME.CFG is used to specify various parameters and filenames to the model simulation, such as the model title, the interval of the simulation, the discrepancy year, and the names of the G bank for macrovariables, the fixes files, and the Vam file that will be used to do the simulation. Figure 7 shows a sample DYME.CFG file.

Figure 7. Sample DYME.CFG File

```
Title of run   ;Demonstration of the Tiny Model
Start year     ;2010
Finish year    ;2020
Start MacEq yr ;2010
Discrepancy yr ;2010
Use all data?  ;yes
VecFix file    ;Vecfixes
MacroFix file  ;Macfixes
Vam file       ;base
G bank         ;base
Debug start yr ;3200
Max iterations ;100
Optimization specification file; none
Number of random draws; 0
Additive random errors; no
Random coefficients; no
```

In this file, text before the semicolon is comment, to indicate the purpose of each configuration item. The first line shows the run title, which will be saved to the Vam file that holds the simulation. The simulation will start in 2010, and run to 2020, and the output discrepancy will be calculated in 2010. "Use all data?" is answered "yes" for normal forecasts, and "no" for historical simulations. The vector fixes and macro fixes file names are standard. The model will forecast macrovariables in the G bank dyme.bnk, and the Vam file will be dyme.vam. The rest of the parameters control model debugging and number of iterations for the Seidel function and for the whole model. All of these parameters can be accessed from within the model.cpp code. For example, the starting and ending date are godate and stopdate.

A file called run.bat is often used for running the model, as it enforces the proper sequence of operations. These are:

1. Copy the macro bank hist.bnk/hist.ind created by *Idbuild* to dyme.bnk/dyme.ind.
2. Copy the Vam file with historical data hist.vam to dyme.vam.
3. Run the *Fixer* and *Macfixer* programs
4. Run the model, dyme.exe
5. Copy dyme.vam, dyme.bnk and dyme.ind to a new file, to save the simulation.

Figure 8 shows a typical run.bat file. The simulation results will be in the Vam file/G bank combination newsim.

Figure 8. Typical RUN.BAT

```
copy hist.bnk dyme.bnk
copy hist.ind dyme.ind
copy hist.vam dyme.vam
fixer
macfixer
dyme
copy dyme.vam newsim.vam
copy dyme.bnk newsim.bnk
copy dyme.ind newsim.ind
```

Viewing Historical Data, or Results of a Simulation

Once a simulation has finished and the results are stored in a file, they can be analyzed with the Inforum tools *G7* and *Compare*. *G7* can be used to assign the simulation macrovariable bank (dyme.bnk) and type or graph the forecasts of macrovariables. For example, to type the forecast of total gdp, you could do:

```
ty gdp 2010 2020
```

To make a simple graph of GDP, which includes historical and forecast data, you could type:

```
ti Gross Domestic Product
gr gdp 2000 2010 2020
```

In *G7*, you can type or graph single elements of any vector or matrix in the model. For example, to graph the history and forecast of output for sector 1:

```
ti Agriculture
subti Output in Constant Dollars
gr out1 2000 2010 2020
```

To graph the diagonal coefficient in the A-matrix for Agriculture:

```
ti Coefficient for 1 Agriculture
subti Sales to 1 Agriculture (diagonal)
```

```
gr am1.1 2000 2010 2020
```

In *G7* the "show" command is also a convenient way of quickly viewing forecasts and data on the screen. If you wanted to view the output vector, from the *G7* prompt type:

```
show out
```

If you wanted to view the A-matrix for the year 2035, you could type:

```
show am y 2020    # 'y' is to view by year
```

If you wanted to view the A-matrix over time for row 3:

```
show am r 3
```

Compare is also extremely useful for generating printed tables from an Interdyme historical dataset or forecast. When running *Compare*, choose databank type 'v' for Vam file. *Compare* will automatically search for a G bank, compressed bank, or hashed bank of the same name as the Vam file to assign for searching for macrovariables. Therefore, *Compare* treats the Vam file/G bank combination as a logical unit.

A simple macro stub file for our hypothetical model is shown in Figure 9.

Figure 9. Macro Stub File for Simple Interdyme Model

```
\announce Macro Summary
\ti Macro Summary
\gd 3
*
;
\decs 1
&
gdp     ; Gross Domestic Product
@csum(pce,1-8);  Personal Consumption Expenditures
@csum(inv,1-8);  Investment
@csum(pde,1-33)  ;  Equipment Investment
@csum(ex,1-8)   ;  Exports
@csum(im,1-8)   ;  Imports
@csum(gov,1-8)  ;  Government
```

The title of this table will be "Macro Summary". The table will contain GDP and its components. Note that these are calculated within *Compare* through the use of the @csum() function, which sums up the elements of a vector over a specified range.

Where Do We Go From Here?

Interdyme is a set of tools for building hybrid macroeconomic interindustry models. These models generally possess a number of features, such as the ability to handle matrices and vectors in Vam files, use macro variables, incorporate vector and macro fixes, calculate "detached coefficient" equations for vectors. In Interdyme 2.0, changes were made to the system to make these features more modular. Using a file called features.h, features can be turned on or off. For example, you can now build an Interdyme model that uses matrices and vectors, but no macrovariables, equations or fixes. Examples of such models include an Occupational forecasting model, some components of the Defense modeling system, as well as some programs written to work with the linked trade data. On the other hand, one can build a pure macro model such as the AMI model explained in *The Craft of Economic Modelingm Part 1*. Such a model uses no vectors, matrices or detached coefficient equations, but does use macrovariables and macro fixes. In many ways, the Interdyme environment is superior to *G*/*Build* for building macro models, since the code is easier to read, and the macro fixing capabilities are more sophisticated. However, Interdyme currently can only handle annual data. An improvement would be the addition of quarterly and monthly frequencies to the system.

Organizing Interdyme as a system with features that can be turned on or off encourages the independent development of new features. For example, a ReSector class was recently developed that handles aggregation of vectors and matrices. This class eases the aggregation of occupational categories from 628 to some more reasonable number, or enables the quick aggregation of LIFT data to something like 15 industries.