The Craft of Economic Modeling,

Part III. Multisectoral Models

Contents

Chapter 14. Input-Output in the Ideal Case

1. Input-Output Flow Tables

Multisectoral models begin from an accounting of the flows of goods and services among various industries of the economy. Table 1 shows a simple interindustry accounting, or input-output flow table, for an imaginary but not unrealistic eight-sector economy which we will call TINY, a reference to the simplicity of its national accounts, a simplicity designed, of course, to make it easy for us to concentrate on essential concepts without being overwhelmed by big tables of data. In Table I, the selling industries are listed down the left side of the table. The last, abbreviated to "GovInd," is "Government Industry", a fictitious industry which in this table simply supplies the government with the services of its own employees. Below these come the classes of factor payments, here Depreciation, Labor compensation, Capital income (such as interest, profits, rents, or proprietor income), and Indirect taxes (such as property taxes, sales taxes, and excise taxes as on alcohol, tobacco, and gasoline). Note the similarity of these categories of factor payments to the categories of national income. Their sum is the row Value added. Across the top of the table the same eight industries are listed as buyers of products. Here they are followed by columns corresponding to the principal divisions of the "product side" of the national accounts, namely

Con Personal consumption expenditure
Gov Government purchases of goods and services
Inv Investment
Exp Exports
Imp Imports (as negative numbers)

In input-output terms, these are the final demand columns. The next-to- last column, labeled FD for "Final Demand," shows their sum. It is shaded to emphasized that it is derived by summing other columns. The next last column, also shaded, is the sum of all the (non-shaded) elements row.

Across each row of the table are shown the sales of that industry to each of the industries and final demand columns. Thus, the 100 in the Agriculture row and Manufacturing (Mfg) column means that Agriculture sold 100 billion dollars (bd) of products to Manufacturing in the year covered by this table. Typical sales here are grains to milling, live animals to meat packing, or fruits and vegetables to plants which can or freeze them. The 15 in the Personal consumption (Con) column of the same row means that Agriculture sold 15 bd of products directly to households during the year. These sales are primarily fresh fruits and vegetables and eggs. In the table shown here, which is said to be in producer prices, they are recorded at the price the farmer received for them. These products are not necessarily bought at the farm gate, however, for going through wholesale and retail trade channels does not change the industry of origin of a product; going through a manufacturing process does. Thus, an orange sold as an orange to she who eats it appears as a sale from Agriculture to Personal consumption, despite the fact that it went through a store. Another orange that was turned into frozen orange juice appears first as a sale from Agriculture to Manufacturing at the price received by the farmer. It then reappears as a sale from Manufacturing to Personal consumption at the manufacturer's price. But the price paid by the ultimate consumer is neither the price received by farmer in the first case nor by the manufacturer in the second. Where is the difference, the commercial margin? In this table, it is in the sales of Commerce to Personal consumption expenditure. Transportation margins are handled similarly. Tables made with this pricing convention are said to be "in producer prices". We shall look at other ways of handling the problem of margins in Chapter 2.

Table 1. An Input-Output Flow Table

Buyer / Seller	Agri-cult.	Mining	Gas Elec	Mfg	Com-merce	Trans-port	Ser-vices	Gov Ind	Con	Gov	Inv	Exp	Imp	FD	Row Sum
Agriculture	20	1	0	100	5	0	2	0	15	1	0	40	-20	36	164
Mining	4	3	20	15	2	1	2	0	2	1	0	10	-10	3	50
Gas&Electric	6	4	10	40	20	10	25	0	80	10	0	0	0	90	205
Mfg	20	10	4	60	25	18	20	0	400	80	200	120	-170	630	787
Commerce	2	1	1	10	2	3	6	0	350	10	6	10	0	376	401
Transport	2	1	5	17	3	2	5	0	130	20	8	5	0	163	198
Services	6	3	8	45	20	5	20	0	500	40	10	30	-20	560	667
GovInd	0	0	0	0	0	0	0	0	0	150	0	0	0	150	150
Intermediate	60	23	48	287	77	39	80	0							614
Deprec.	8	4	40	40	25	30	20	0							167
Labor	68	21	31	350	150	107	490	150							1367
Capital	20	2	56	60	40	12	59	0							259
Indirect tax	8	0	20	50	109	10	18	0							215
Value added	104	27	147	500	324	159	587	150							2008
ColSum	164	50	205	787	401	198	667	150	1477	312	224	215	-220	2008	

As we look down the column for an industry, we see all the products which it needs for making its own. In the Agriculture column, we see first of all 20 bd from Agriculture itself. These are sales primarily of feed grains to animal husbandry, but include also sales of seed, hay, manure, and other products. These sales within the industry are common and are referred to in input-output jargon as "diagonals" because they appear on the main diagonal of the table. Further down the Agriculture column we see 4 bd for Mining, primarily crushed limestone, but also some coal. The 20 bd spent on Manufacturing bought gasoline, fertilizers, and pesticides. The 2 bd spent on Commerce were trade margins on these manufactured products. The 2 bd spent on Transport included transportation margins on the products of the other industries as well as costs incurred by the farmer in getting products to market. The purchases from Services includes the services of veterinarians, lawyers, and accountants. All the purchases of the industries from each other are called "intermediate" purchases because they do not go directly to the final user but are "mediated" by other industries. The sum of the intermediate purchases by each industry are in the row labeled "Intermediate" and shaded, as before, to show that it is derived by adding other entries in the table. Many tables also have a total intermediate column; our table omits it for the simple reason that it would not fit on the page.

Below the "Intermediate row" are the value-added rows. We find that Depreciation of equipment came to 8 bd. Labor received 68 bd. (In our imaginary economy, we imagine that proprietor income has been divided between labor and capital income. In most actual tables, it will be shown separately or classified as capital income.) The 20 bd of capital income includes interest payments, corporate profits, and capital's portion of proprietor income. The 8 bd of Indirect taxes is mostly property taxes.

Now precisely because the Capital income row of value added -- which includes both corporate profits and proprietor income -- is the total of sales minus the total of expenses, the column sum for each industry is equal to its row sum. For example, the row sum of Agriculture is 164 and the column sum (of the unshaded entries) is 164, and so on for all eight industries. This fact has a remarkable consequence which is the cornerstone of national accounting, namely that the sum of all the value-added entries is equal to the sum of all the final demand entries. In our table, each of these groups of entries is surrounded by a double line and each adds to 2008. Why is the total the same? Since the sum of each of the eight industry rows, say R, is equal to the sum of the corresponding column, the sum of all eight rows, 2622, is equal to the sum of all eight columns, say C, which is also 2622. Thus we have with R = C. But the total of the final demands, D, is R minus the total of the intermediate flows, say X, or D = R - X. Likewise, the total value added, V, is C, the sum of all the industry columns, less the sum of that part of them which is intermediate, or V = C - X. But R = C implies that R - X = C - X or D = V. Naturally, this D or V has a name, and that name is Gross Domestic Product. We have thus proved the fundamental identity of national accounting: Gross Domestic Product (GDP) is the same whether measured by the products that go to final demand or by the income which goes to factors. In our table, this identity appears in the fact that the sum of the FD column, 2008, is the sum of the Value added row, also 2008, which is the GDP of this economy. Arrayed in format of national accounts, our economy would appear as in Table 2.

Table 2. The Income and Product Account

Gross domestic product	2008	Gross domestic product	2008
Personal Consumption	1477	- Depreciation	167
Investment	224	= Net domestic product	1841
Exports	215	- Indirect taxes	215
Imports	-220	= National income	1626
Government purchases	312	Labor income	1367
		Capital income	259

Before leaving Table 1, we must make a fundamental point about it. With one small exception, the table makes sense in physical units. We can measure the output of Agriculture in bushels, that of Mining in tons, that of Gas and Electricity in BTU's, Transport in ton-miles, Labor in worker hours, Capital income in ounces of gold, and so on. Detailed tables in physical terms have in fact been made for China. Wassily Leontief, maker of the first input-output table, used to often insist in seminars that any calculations had to make sense in physical terms.

The small exception, however, is important: the column sums of a table in physical terms are utterly meaningless since all the elements are in different units. Naturally, the row totals -- which are meaningful -- do not equal the meaningless totals of the corresponding columns. This point would seem so obvious as to be not worth making were it not for the fact that it is often forgotten, precisely by the makers of input-output tables. For if a table is made in the prices of some year other than the year to which it refers, it is essentially in physical units. Thus, we can make a table for 2000 in 1980 prices, where the physical measure in each row is "one 1980 dollar's worth" of the product. In other words, the physical unit for each product is how much of it one dollar would buy in 1980. For any product for which a price index can be made, 2000- dollar amounts can be converted into 1980-dollar physical units by the price index. For value added, since there is no very natural unit, one can simply deflate all of the value-added cells by the GDP deflator. The total real value added will then be the same as total real final demand. One can have in this way a perfectly sensible, meaningful table. *But its column sums are meaningless and certainly do not equal the corresponding row sums.*

Unfortunately, some table makers have disregarded this fact and have simply forced the value added in each industry of such a table to equal the difference between the row sum of the industry and the sum of the intermediate inputs into it. The results make as much sense as saying that five squirrels minus three elephants equals two lions. The arithmetic is right but the units are crazy.

This practice is called "double deflation" because first the outputs are deflated and then the purchased inputs deflated and subtracted from the deflated output to obtain a mongrel, mixed-up-units number, possibly positive but also possibly negative, mistakenly alleged to be a measure of "constant-price value added". It is, in fact, what would have been left over for paying primary factors, had producers, contrary to economic theory, gone right on producing with the previous period's inputs after prices have changed. That is certainly no measure of "real value added," for it is not, in all probability, what producers did. The error would perhaps be easier to see if labor input, for which we have some measures of cost, were considered as an intermediate input and indirect taxes were simply subtracted in current prices from output. The double-deflation procedure should then give a measure of "real capital income." In such a table, the deflators for capital income would be different in different industries. The residuals might well be negative, especially if there were a few years between the two periods. Trying to deflate the difference between two numbers that are very close together by deflating each of the two numbers by different deflators and then taking the difference between the two deflated items is simply asking for trouble.

The nonsense involved in double deflation is often masked by the taking the time periods of the tables close together and "chaining" the index, so that negative values are unlikely. But nonsense in small increments is still nonsense. Unfortunately, this nonsense is compounded by the fact that these procedures are sanctioned by international statistical standards, and many statistical offices engage in them. Economists have made matters worse by taking these mixed-units numbers as measures of "real" product in studies of productivity.

As far as I am aware, there is no satisfactory way of measuring real productivity at the individual industry level, precisely because industries cooperate with one another in production, and how they do so changes. In one year, for example, the “television set industry” is a collection of plants that make the cabinets, the tubes and the electronics, and assemble the sets. In a later year, the industry has become assembly plants that buy cabinets, tubes, and electronics and assemble them. Clearly, changes in sales (even in constant prices) divided by labor input in worker hours in this one industry is not an appropriate measure of productivity increase. Rather, changes in “productivity” in this case is meaningful only as applied to how much labor and capital is required *by the whole economy* to produce a television set. We shall see how it can be meaningfully calculated. The meaningful, correct calculation has nothing whatever to do with double deflation. But the quest to allocate the changes in whole-economy productivity for particular products to individual industries is a search for a nonexistent – and superfluous – El Dorado.

2. Input-Output Equations. The Fundamental Theorem.

An input-flow table describes an economy in a particular year. Its greatest value, however, lies in the ability it gives us to answer the question What would the outputs, value added, and intermediate flows have been had the final demands been different? To answer that question in the simplest possible way, we must assume that the ratio of each input into an industry to that industry's output remains constant when the final demands are changed. These ratios are known as the "input-output coefficients," and may be defined by

$$a_{ij} = x_{ij} / q_j$$

where x_{ij} is the flow from industry i to industry j in Table 1.1 and q_j is the output of industry j, that is, it is the sum of row j or column j in the same table. For example,

$$a_{1,4} = 100/787 = 0.12706$$

Table 3 shows the complete matrix of these input-output coefficients corresponding to Table 1.

If we are willing to suppose that these coefficients remain constant as the final demand vector changes, then for any vector of final demands, f, we can calculate the vector of industry outputs, q, from the equation

$$q = Aq + f \qquad (14.2.1)$$

where A is the matrix of input-output coefficients in Table 3. If we happen to choose as f the column vector of final demands in Table 1, (the first eight elements of the FD column: (36,3,90, ..., 150)'), then q should be the column vector of industry outputs of Table 1 (the vector of row sums of the eight industry rows: (164,50,205,...,150)'). For other values of f, of course, we will find other values of q.

Table 3. Input-Output Coefficients

	Agric	Mining	Gas&Elec	Mfg	Com	Trans	Serv	GovInd
Agriculture	0.12195	0.02000	0.00000	0.12706	0.01247	0.00000	0.00300	0.00000
Mining	0.02439	0.06000	0.09756	0.01906	0.00499	0.00505	0.00300	0.00000
Electricity	0.03659	0.08000	0.04878	0.05083	0.04988	0.05051	0.03748	0.00000
Manufacturing	0.12195	0.20000	0.01951	0.07624	0.06234	0.09091	0.02999	0.00000
Commerce	0.01220	0.02000	0.00488	0.01271	0.00499	0.01515	0.00900	0.00000
Transportation	0.01220	0.02000	0.02439	0.02160	0.00748	0.01010	0.00750	0.00000
Services	0.03659	0.06000	0.03902	0.05718	0.04988	0.02525	0.02999	0.00000
GovInd	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

One way of solving (1.2.1) is to rewrite it as

$$(I - A)q = f$$

or

$$q = (I - A)^{-1}f.$$

The matrix of $(I - A)^{-1}$ on the right of this equation is known as the Leontief inverse of the A matrix. For our example, it is shown in Table 4. Its elements have a simple meaning. Element (i,j) shows how much of product i must be produced in order to produce one unit of final demand for product j. This interpretation is readily justified by taking f to be a vector of zeroes except for a 1 in row i. Then q will be the ith column of $(I - A)^{-1}$, and its jth element will show exactly how much of product j will have to be produced in order to supply exactly one unit of i to final demand. In our example, in order to supply one unit of Agricultural product to final demand, 0.1691 units of Manufacturing must be produced. Note that, in the example, all elements of the Leontief inverse are non-negative. In view of the economic interpretation, that result is hardly surprising. Later in this chapter, we will show mathematically that the Leontief inverse from an observed *A* matrix is always non-negative.

Table 4. The Leontief Inverse $(I - A)^{-1}$

1.1647	0.0620	0.0107	0.1634	0.0263	0.0165	0.0096	0.0000
0.0405	1.0830	0.1126	0.0352	0.0144	0.0150	0.0092	0.0000
0.0617	0.1137	1.0683	0.0748	0.0623	0.0641	0.0452	0.0000
0.1691	0.2530	0.0538	1.1201	0.0791	0.1091	0.0396	0.0000
0.0184	0.0276	0.0093	0.0185	1.0077	0.0180	0.0106	0.0000
0.0210	0.0319	0.0304	0.0297	0.0120	1.0151	0.0102	0.0000
0.0604	0.0911	0.0548	0.0791	0.0612	0.0379	1.0368	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000

We may also ask how much of a primary resource, such as Labor or Capital, would be needed for the production of a given final demand. We may define the resource coefficients similarly to the input-output coefficients by

$$r_{ij} = y_{ij}/q_j$$

where y_{ij} is the payment to factor i by industry j. For example, from Table 1, $y_{2,4}$, the payment to resource 2, Labor, by industry 4, Manufacturing, is 360. If we denote by R the matrix of the r_{ij}, then the vector of total payments to each resource for an output vector *q* is *Rq*, and for a final demand vector, f, it is $R(I - A)^{-1}f$.

If we now think of each row of this matrix as a row vector and sum these vectors -- a process which makes sense if all the rows are measured in monetary values in the prices of the year of the table -- we get a row vector, *v*, of value-added per unit of output. Just as previously we asked how outputs, *q*, would change if *f* changed while *A* remains constant, we can now ask how prices, *p*, would change if *v* changed while *A* remains constant. The row vector *p* must satisfy the equations

$$p = pA + v. \tag{14.2.2}$$

These equations state simply that the price of a unit of each product is equal to the cost of all products used in producing that unit (the first term on the right) plus value-added per unit produced. Just as the equations (14.2.1) provide the fundamental connection in multisectoral models between final demands and outputs, so these equations provide the fundamental connection between unit value added and prices. If we want to know how specific changes in productivity or in wages in one or several industries will affect prices in all industries, these equations are the key. If we calculate the prices for *v* vector given in the table, we should find that all prices are equal to 1.

There is, furthermore, a relation of fundamental importance between the solutions of the two sets of equations. Namely, given any A, f, and v, the q and p which satisfy $q = Aq + f$ and $p = pA + v$ also satisfy

$$vq = pf. \tag{14.2.3}$$

This equation says that the value of the final demands evaluated at the prices implied by equations (14.2.2) are equal to the payments to the resources necessary to produce those final demands by (1.2.1). Thus, if our outputs and prices satisfy the required equations, we can be certain that GDP measured by the final demands in current prices will be equal to the GDP measured by the payments to resources (or factors) in current prices. If we build these equations into our models, we can be certain that the models will satisfy the basic accounting identity in current prices. This relation may well be called the fundamental theorem of input-output analysis. Fortunately, it is as easy to prove as it is important, and you shouldproduce your own proof. If you need help desperately, look in the footnote. [1]

3. Combining Input-Output and Institutional Accounts

The national accounts which we have presented so far in connection with the input-output table lack some of the concepts which we found very useful in macroeconomic modeling, concepts like Personal income, Personal disposable income, Personal saving, Personal income taxes, and Government transfers to persons. The basic “institutions” in national accounts are (1) Persons, (2) Businesses, (3) Governments, and (4) Rest of World. Sometimes businesses are divided between financial and non-financial businesses, but we will not make that distinction in TINY. “Persons” includes non-profit corporations such as private universities. The Rest of the World, abbreviated as RoW, shows only transactions of “institutions” of other countries with the “institutions” of the country concerned.

The institutional accounts begin with the allocation of components of value added from the input-output accounts to the institutions which receive them. Labor income is allocated to Persons; Depreciation and Capital income is allocate to Business; Indirect taxes are allocated to Governments. Government transfers, such as social insurance and welfare payments, are then moved from Governments to Persons, to give Personal income. Then taxes are moved from Persons and Business to Governments, with Disposable income as the balance.

There are several ways to present these accounts. The simplest is similar to that used in the USA NIPA and familiar from Part 1 of this book.

[1.] Multiply (14.2.1) on the left by p to get
(A) $pq = = pAq + pf$
Multiply (14.2.2) on the right by q to get
(B) $pq = pAq + pf$
Subtract (B) from (A) to get
(C) $0 = pf - vq$ or $pf = vq$.

Institutional Accounts for TINY: NIPA-Style Presentation

Year 2000

	Persons	
+	Labor income	1367
+	Interest and dividends received	220
+	Government transfers	150
=	Personal Income	1737
-	Personal taxes	226
=	Disposable income	1511
-	Personal consumption expenditure	1477
=	Personal saving	34
	Business	
+	Depreciation	167
+	Capital income	259
-	Interest and dividends paid	220
-	Investment	224
=	Business saving	-18
	Governments	
+	Indirect taxes	215
+	Personal taxes	226
-	Gov't purchases of goods and services	-312
-	Gov't transfers to persons	-150
=	Gov't saving	-21
	Rest of World	
+	Imports	220
-	Exports	215
=	RoW saving	5

A consequence of the fundamental identity of the total value added and the total final demand in the input-output table is that the total saving is identically zero. You can exercise your mental arithmetic to quickly verify this identity for TINY. The NIPA-style account is clear, easy to read, and easy to convert into a program for calculation. Furthermore, data for several years can be conveniently shown in parallel columns that make comparison easy. Its disadvantage is that its form does not make evident why total saving is zero or what are matching entries. For example, the form of the accounts does not show that Personal taxes paid by Persons is the same as Personal taxes received by Governments.

That shortcoming is overcome in a second way of presenting the institutional accounts, a way I will call the Balances presentation. This presentation also makes clear why total saving is zero. It is shown in the table below.

Institutional Accounts for TINY: Balances Presentation

Transaction	Persons	Business	Gov	RoW		PCE	Gov	Inv	NetExp
Primary distribution	1367	426	215	0	=	1477	312	224	-5
Interest and dividends	220	-220			=				
Gov’t transfers	150		-150		=				
Balance: Inst. Income	1737	206	65		=	1477	312	224	-5
Direct taxes	-226		226		=				0
Balance: Disposable income	1511	206	291		=	1477	312	224	-5
Personal consumption	-1477				=	-1477			
Govenment purchases			-312		=		-312		
Business investment		-224			=			-224	
Net imports				5	=				5
Balance: Saving	34	-18	-21	5	=	0	0	0	0

In the first line, the “Primary distribution” of Value added, labor income is given to Persons; Depreciation and Capital income, to Business; and Indirect taxes, to Governments. To the right of the = sign are the components of Final demand. The sum of the items to the left of the = sign is, of course, equal to the sum of those on the right.

Next follow two transfer lines that (1) move Interest and dividends from the Business column to the Persons column, and (2) move Government transfers to persons from the Government column to the Persons column. The next line, labeled “Balance: Institutional Income,” is a balance line, the sum of the preceeding lines. In the Persons column, it gives Personal income. Below it, the Direct taxes transfer line moves personal income taxes from Persons to Government and could also move corporate profit taxes from Business to Governments. (For TINY, however, we have assumed that these corporate taxes are zero.) The next balance line, the sum of the previous balance line with the intervening transfer line, gives Disposable income by institution. Then follow the lines which subtract the final demand expenditures from the institutions which make them. The final balance line then gives the savings of each institution on the left of the = sign and zeroes on the right. Of course, the sum of the items on the left of this last line equals the sum of the items on the right, namely, zero. Thus, this presentation makes it clear why total saving, including that of the Rest of the World in our country, is always zero. The major disadvantage of this layout is that it cannot show data for several years in close proximity so as to make comparison easy.

The international System of National Accounts (SNA) used by most countries other than the USA, uses a presentation based on the Balances Presentation, but somewhat more complicated and much less clear. Here it is for TINY.

Institutional Accounts for TINY: SNA-Style Presentation

Institution	Persons		Business		Governments		Rest of World	
Transaction	Sources	Uses	Sources	Uses	Sources	Uses	Sources	Uses
Primary distribution	1367		426		215		220	215
Interest and dividends	220			220				
Government transfers	150					150		
Personal tax		226			226			
Totals	1737	226	426	220	441	150	220	215
Balance:Disposable income	1511		206		291		5	
Personal consumption expenditures		1477						
Government expenditures						312		
Business investment				224				
Totals	1511	1477	206	224	291	312	5	
Balance: Saving	34		-18		-21		5	

Under each institution are two columns, one for sources of funds for the institutions and one for uses of of funds. Instead of a single line for each of the balances, two lines are necessary, one to take the totals and one to show (in the Sources column) the result of subtracting total uses from total sources. I have not shown a balance line of Institutional of income (of which Personal income is a highly useful instance) because this concept plays no role in the SNA, which thus fail to give a concept useful as a base for calculating personal income taxes. The SNA presentation does not make clear why total saving is zero and requires two lines for each balance instead of one, though I have seen a number of presentations in which the total lines ares omitted, thus making it very hard for the reader to figure out what is going on. About the only virtue of the SNA system is that it largely avoids negative numbers.

Yet a fourth presentation combines the input-output table with the institutional accounts in what is called a Social Accounting Matrix or SAM. The SAM for TINY is shown in the box below. In an input-output table, the row sums equal the corresponding column sums for the industries. The SAM generalizes that idea so that all accounting identities are expressed by requiring the sum of each row to equal the sum of the corresponding column in a square matrix. In the SAM for TINY, the first rows are those of the input-output table, both the products and the value-added. Below these rows, we add a row for each institution and then one for each final demand column and finally a row for saving. Between the columns for industries and the final demand columns we slip columns with the same names as the value-added rows, and then a column for each institution. After the final demand columns, we append one corresponding to the Savings row. The “Primary distribution” line of the SNA-Style accounts is then represented by the total of each type of value added into the cell at the intersection of row for the institution receiving the income and the column of the type of income. At this point, the row totals equal the column totals for the industries and for value-added components. The transfers among institutions are then shown by entering the amount in the row of the receiver and the column of the payer. The totals of each final demand column are entered into the corresponding row in the column of the institution purchasing that final demand. All row totals now equal corresponding column totals except for the four institutions. Their row totals are their receipts while their column totals are their expenditures. They differ by the amount of saving by each institution. So if we now enter these savings in the Saving row at the bottom of the table, the row totals equal the column totals also for the institutions. The row sum of the Saving row is, as has been said repeatedly, zero, so to match the Saving row, we just need an all-zero Saving column.

Social Accounting Matrix for TINY

	1 Ag	2 Min	3 G&E	4 Mfg	5 Com	6 Trans	7 Serv	8 Gov Ind	9 Dep	10 Labor	11 Capital	12 Ind Tax	13 Per-sons	14 Bus	15 Gov't	16 RoW	17 PCE	18 Gov	19 Invest	20 Exp	21 Imp	22 Sav	23 Tot
1 Ag	20	1	0	100	5	0	2	0									15	1	0	40	-20		164
2 Mining	4	3	20	15	2	1	2	0									2	1	0	10	-10		50
3 G&E	6	4	10	40	20	10	25	0									80	10	0	0	0		205
4 Mfg	20	10	4	60	25	18	20	0									400	80	200	120	-170		787
5 Commerce	2	1	1	10	2	3	6	0									350	10	6	10	0		401
6 Transport	2	1	5	17	3	2	5	0									130	20	8	5	0		198
7 Services	6	3	8	45	20	5	20	0									500	40	10	30	-20		667
8 GovInd	0	0	0	0	0	0	0	0									0	150	0	0	0		150
9 Deprec.	8	4	40	40	25	30	20	0															167
10 Labor	68	21	31	350	150	107	490	150															1367
11 Capital	20	2	66	60	40	12	59	0															259
12 IndTax	8	0	20	50	109	10	18	0															215
13 Persons										1367				220	150								1737
14 Firms									167		259												426
15 Gov't												215	226										441
16 RoW																							0
17 PCE													1477										1477
18 Gov Purch															312								312
19 Invest														224									224
20 Export																215							215
21 Import																-220							-220
22 Saving													34	-18	-21	5							0
23 Col Sum	164	50	205	787	401	198	667	150	167	1367	259	215	1737	426	441	0	1477	312	224	215	-220	0	

Social Accounting Matrices have proven quite popular with economists. I find the term used to mean simply national accounts with a consistent input-output table and institutional accounts. In fact, a SAM is just one way of presenting such a system. Its only advantage, as far as I can see, is that the form makes evident the consistency. Otherwise, it is a perfectly terrible way to present data, as you can readily see by comparing the ease of reading any of the other ways of presenting the TINY accounts with the SAM. And as the input-output table increases in detail, the SAM becomes worse and worse as a way of actually viewing data. Consequently, we shall make no further use of SAM's and will generally use the NIPA-like presentation because of the important advantage that data for several years can be shown in parallel columns.

To illustrate the use of integrated national accounts in combination with interindustry tables, we need historical series for at least the national accounts aggregates. I have made up such a data bank for TINY with the values shown above for the year 2000 and with values for other years from 1978 to 2003 made up by assuming a movement similar to that of the corresponding entry in the USA NIPA. These "historical" series are in the TINY data bank.

4. Introduction to Input-Output Computing with just G

In this section, we will see how to turn the TINY input-output table and data bank into a simple input-output model using only commands available in G. In this model, we will move each final demand column forward and backward over the period 1995 - 2003 by the index of the corresponding GDP component in the TINY data bank. Then we move all the final demand vectors except investment up by 3.0 percent per year from 2003 to 2010. Investment is moved forward by a wavy series composed of a base series growing at 3.0 percent per year plus a sinusoidal function. Input-output coefficients and the composition of the five final demand components are kept constant. Outputs by each industrial sector are then calculated for every year 1995 to 2010. With the additional assumption that the shares of each type of income in value added by each industry remain constant, we calculate income of each type in each industry. Piecewise linear trends in the input-output coefficients, value-added coefficients, and composition of the final demand vectors could easily be introduced, but that has been left as an exercise. This model is incomplete and somewhat inconsistent with itself for many reasons, including th following: (a) it does not assure consistency of Personal consumption expenditure with the Personal income it implies (b) it does not relate the imports of a product to the domestic use of the product, and (c) investment is not detailed by industry and related to the growth of the industry as found by the model. Introducing such features to exploit the full potential of input-out modeling will require the Interdyme software described in following sections. Despite these limitations, such simple models as the one described here, though with greater industry detail and more finely divided final demands, have been widely used by groups which have a macroeconomic model and want the industry outputs consistent with the its final demand forecasts.

Working with input-output in G requires the use of a new sort of data bank known as a VAM (Vectors And Matrices) file. As the name suggests, this type of data bank holds time series of vectors and matrices. G has commands which can add, subtract, multiply, and invert matrices and add and subtract vectors and multiply them by matrices. Thus, the operations discussed so far, and several others, can easily be performed in G. A VAM file differs in two important respects from the G data banks we have worked with so far:

(1) In the standard G bank, all elements are the same size, namely a time series of a single variable beginning at the beginning of the data bank and extending over the number of

observations in the bank, as specified by the G.cfg file. In VAM files, elements are time series of vectors or matrices of various dimensions. As in the standard G bank, all time series are the same length.

(2) In standard G banks, we can create new series as we work, for example, with *f, fex,* or *data* commands. In VAM files, we buy the flexibility of having elements of various sizes by specifying at the outset the contents of the file, that is, the names and dimensions of each vector or matrix in the bank along with the names of the files giving the titles of the row or columns of the vector or matrix. One might suppose that it is a bit of nuisance to have to specify this structure of the VAM file at the outset. In practice, however, this need to prespecify structure proves a useful discipline in building complex models. If, as a model evolves, it becomes necessary to revise the specification of the VAM file, it is easy to copy the contents of the old file into the new, enlarged file. This specification is accomplished by a file usually named VAM.CFG.

We can illustrate the use of the VAM file and some new G commands for making some simple calculations with the input-output table presented in section 1 of this chapter, which we will assume is for the year 2000. The box below shows the VAM.CFG file for this model, which we will call TINYI. It and all the files used in this chapter are in the file TINY.ZIP. I suggest that you make a directory (also called a folder), copy TINY.ZIP into it, and unzip it.

The first line in VAM.CFG gives the beginning and ending years for the VAM file. The next line, the one beginning with a #, is a comment to clarify the structure of the file. Comments beginning with a # can be placed anywhere in the file. Then come free-form lines giving

1. The name of the element
2. Its number of rows

VAM.CFG File for the TINY Model

```
1995 2010
# Vam file for Simplest Model
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
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
depc       8   1  0  sectors.ttl # Depreciation coefficients
labc       8   1  0  sectors.ttl # Labor income coefficients
capc       8   1  0  sectors.ttl # Capital income coefficients
indc       8   1  0  sectors.ttl # Indirect taxes coefficients
pcec       8   1  0  sectors.ttl # Personal consumption shares
invc       8   1  0  sectors.ttl # Investment shares
govc       8   1  0  sectors.ttl # Gov shares
exc        8   1  0  sectors.ttl # Export shares
imc        8   1  0  sectors.ttl # Import shares
x          8   1  0  sectors.ttl # Working space
y          8   1  0  sectors.ttl # Working space
```

3. Its number of columns
4. The maximum number of lags with which a vector occurs in the model or a *p* if the matrix is a “packed matrix” – a device useful in large-scale models.
5. The name of a file containing the names of the rows of a vector or matrix
6. The name of a file containing the names of the columns of a matrix
7. A # followed by a brief description of the element.

As far as the computer is concerned, these lines are free format; all that is needed is one or more spaces between each item on a line. But this is a file also read by humans, so putting in spaces to make the items line up in neat columns is also a good idea. The accompanying box shows the vam.cfg file for the TINY model based on example of section 1 of this chapter.

To create a vam file from a vam configuration file the command in G 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:

The FD.dat File for Introducing the Final Demands into the VAM File

```
vmatdata c 5 1 1 8 15
2000            pce  gov   inv     ex      im
#            PersCon   Gov Invest Exports Imports
Agriculture      15     1     0    40     -20
Mining            2     1     0    10     -10
Electricity      80    10     0     0       0
Manufacturing   400    80   200   120    -170
Commerce        350    10     6    10       0
Transportation  130    20     8     5       0
Services        500    40    10    30     -20
Government        0   150     0     0       0
```

```
vamcr vam.cfg hist
```

At this point, the newly created vam file has zeroes for all its data. We will now see how to put data into it and work with that data. 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.

The usual ways to introduce data into a VAM file are with the *matin* command for matrices and the *vmatdat* command for vectors. We can illustrate them with the data for TINY from section 1.

The Flows.dat File for Introducing the Input-Output Flow Matrix into the VAM File

```
matin FM 2000 1 8 1 8  15
#             Agricul Mining Elect  Mfg Commerce Transp Services Govt
Agriculture       20      1      0  100        5      0        2    0
Mining             4      3     20   15        2      1        2    0
Electricity        6      4     10   40       20     10       25    0
Manufacturing     20     10      4   60       25     18       20    0
Commerce           2      1      1   10        2      3        6    0
Transportation     2      1      5   17        3      2        5    0
Services           6      3      8   45       20      5       20    0
Government         0      0      0    0        0      0        0    0
```

The *matin* command on the first line is followed by the matrix name in VAM.CFG file, then by the year to which the matrix belongs, then the number of the first row and last row in the following rectangle of data, then the number first column and last column in the rectangle. (In the present case, the rectangle is the whole table; but this ability to read in a table rectangle-by-rectangle is quite useful for reading tables scanned from printed pages.) The last number on the *matin* line is the skip count, the number of characters to be skipped at the beginning of each line. These characters usually give sector names or numbers. The # in the first position marks the second line as a comment. Then come the data; each line is in free format after the initial skip. (Do not use tabs in characters which are to be skipped; the tab character will be counted as just one character.)

The FD.dat file shown below illustrates the introduction of vectors, in this case, the final demands. The *vmatdat* command is rather flexible; it can introduce a number of vectors for one year or one vector for a number of years. The vectors can be the rows or the columns in the following rectangle of data. Because of this flexibility, we have to tell the command how to interpret the rectangle of data. The command must therefore by followed by a *c* or an *r* to indicate whether the vectors appear as columns or rows in the following rectangle of data. Here, the vectors are clearly columns. The next number is the number of vectors in the rectangle; here 5. Next is the number of years represented in the rectangle. Here it is 1, for the columns are different vectors for the same year. (Either the number of vectors or the number of years must be 1.) The next two numbers are the first and last element numbers of the data in the rectangle, and the last is the skip count as before. Since this command is introducing several vectors for *one* year, that year is specified at the beginning of the next line, and the names of the vectors follow it. (If we were introducing data for one vector for several years, the vector name would be in the first position on this line, followed by the year numbers.)

The value-added rows are introduced by the *vmatdat* command and data shown in the box below.

The VA.DAT File for Introducing the Value-added Vectors

```
vmatdata r 4 1  1 8 15
2000  dep lab cap ind
#               1      2      3      4      5      6      7      8
Depreciation    9      4     40     40     25     30     20      0
Labor          68     21     31    350    150    107    490    150
Capital        20      2     56     60     40     12     59      0
Indirect tax    8      0     20     50    109     10     18      0
```

Here, finally, are the G commands to create the VAM file and load the data into it:

```
# Create and load the VAM file for TINY
vamcreate vam.cfg hist
vam hist b
dvam b
# Bring in the intermediate flow matrix
add flows.dat
# Bring in the final demand vectors
add fd.dat
# Bring in the value added vectors
add va.dat
```

These and the following commands to G for making the calculations described in this section are in the file Gmodel, shown in an accompanying box. To fit this large file on a single page, some commands have been doubled up on a single line but seprated by a semicolon – a trick which works in G.

Now let us look at some of the data we have introduced by displaying them in a grid on the screen. The command

```
show FM   y   2000
```

will show in a spreadsheet-like grid the FM matrix, the flow matrix for the year 2000. To adjust the default column width and the number of decimal places in the display, click the Options menu item. Not only does this display look like a spreadsheet display, it also works like one in that you can copy and paste between data from one to the other.

To look at a row, say row 2, of the FM matrix for all years of the VAM file, the command is

```
show FM r 2
```

while to show column 5for all years, the command is

```
show FM c 5
```

Thus, in showing a matrix, we have to choose among showing the whole matrix for one year and showing one row or column for all years. The choice is indicated by the letter – a *y*, *r* or *c* – following the matrix name.

 # Gmodel.pre –File to Build a TINY model using only G, no Interdyme

```
zap ;clear
bank tiny
vamcreate vam.cfg hist
vam hist b
dvam b
# Bring in the intermediate flow matrix
add flows.dat
show b.FM y 2000
# Bring in the final demand vectors
add fd.dat
# Bring in the value added vectors
add va.dat
fdates 2000 2000
# Add up the intermediate rows
getsum FM r out
# Add on the final demand vectors to get total output
vc out = out+pce+gov+inv+ex+im
show b.out
# Copy intermediate flows to AM and convert to coefficients
mcopy b.AM b.FM
coef AM out
vc depc = dep/out ; vc labc = lab/out; vc capc = cap/out; vc indc = ind/out
# Copy the 2000 coefficient matrices to all the other years
fdates 1995 2010
# Copy the 2000 AM matrix into 1995 - 2010
dfreq 1
f one = 1.
index 2000 one AM
# Demonstrate that AM has been copied by showing its first column.
show b.AM c 1
index 2000 one depc ; index 2000 one labc ;index 2000 one capc ;index 2000 one indc
# Move the four final demand columns by their totals
# in the historical years, 1995 - 2003
fdates 1995 2003
index 2000 pcetot pce;index 2000 invtot inv; index 2000 govtot gov;index 2000 extot  ex
index 2000 imtot  im
# Extend the final demands from 2003 to 2010 using a 3 percent growth rate for all
# but inv and a wavy pattern for it.
fdates 1995 2010
# Create a time trend
f time = @cum(time,one,0)
f g03 = @exp(.03*(time-9))
f waves = g03 + .3*@sin(time-9)
fdates 2003 2010
index 2003 g03   pce ; index 2003 waves inv ; index 2003 g03   gov ;index 2003 g03   ex
index 2003 g03   im
# Take the Leontief inverse of the A matrix
fdates 1995 2010
mcopy b.LINV b.AM
linv LINV
# Add up the final demands
vc fd = pce+gov+inv+ex+im
# Compute total outputs
vc out = LINV*fd
# Compute Value added
# The following are element-by-element multiplication
vc dep = depc*out ; vc lab = labc*out; vc cap = capc*out; vc ind = indc*out
gdates 1995 2003 2010
fadd graphs.fad sectors.ttl
```

Showing vectors is simpler because we do not have to make this choice; we just name the vector and get all values for all years. Here are two examples

```
show ind     # Display the indirect tax vector
show b.pce   # Display the personal consumption expenditure vector
```

The second of these examples shows that the *show* command allows us to specify by the bank letter followed by a dot the bank from which the item is to be shown.

Now that we have read in the data and displayed it to check that it was accurately read, we can begin to compute. To calculate the input-output coefficient matrix, we need *out*, thevector of outputs by industry. It was not read in, but it can be computed by summing the rows of the FM matrix and then adding to this row sum the final demand columns. Here are the two commands and the *show* command to see the result:

```
# Add up the intermediate rows
getsum FM r out
# Add on the final demand vectors to get total output
vc out = out+pce+gov+inv+ex+im
show b.out
```

We are now ready to copy the flow matrix, stored in FM, to AM and then convert it to input-output coefficients by dividing each element of each column by the corresponding element of the *out* vector. We do the copy with the with the *mcopy* command, for "matrix copy." The general form of the mcopy command to copy matrix or vector A from bank x to element B in bank y is

mcopy y.B [=] x.A

The = sign is optional but is useful reminder of which way the copy is going. The y. is optional if y is the default VAM file, and the same is true for the x.. Since this copy and these calculations need be done only for one year, the first, 2000, we first set the fdates so that the *mcopy* and *coef* commands work only on the years from 2000 to 2000 (which is to say, only for 2000). Here are the commands

```
# Copy intermediate flows to AM and convert to coefficients
fdates 2000 2000
mcopy b.AM = b.FM
coef AM out
show AM y 2000
# Create value-added coefficient vectors.
vc depc = dep/out
vc labc = lab/out
vc capc = cap/out
vc indc = ind/out
# Set fdates back to the entire range of the VAM file.
fdates 1995 2010
```

With the input-output coefficients calculated, we can now go on to illustrate finding the Leontief inverse, calculating outputs from exogenous forecasts of final demands, calculating value-added components, and displaying, graphing, and making tables of the results. We will first copy the input-output coefficient matrix and the value-added coefficient vectors from 1995 to the other years out to 2010. We can conveniently do this with G's *index* command. This command is used to move all elements of a vector or matrix in the default VAM file forward or backward in proportion to a guide series. Its general form is:

index <base year> <guide series> <matrix or vector>

It operates over the range specified by the current value of the fdates. Since we just want to copy the coefficients to all the years, our guide series will be simply a series of 1's, which we shall call *one*. Here are the commands

```
# Copy the 2000 AM matrix into 1995 - 2010
dfreq 1
f one = 1.
index 2000 one AM
index 2000 one depc
index 2000 one labc
index 2000 one capc
index 2000 one indc
show AM c 1
```

The last command displays in a grid the first column of the AM matrix for all the years; all columns of this display should, of course, be identical. For purposes of our illustration, we will let *AM* remain constant in all years.

The final demands, however, we will move in a slightly more interesting way. Between 1995 and 2003, each the elements of each final demand column will follow the index of the total of that column as given in the national accounts. Here are the G commands to make that happen.

```
# Move the four final demand columns by their totals
# in the historical years, 1995 - 2003
fdates 1995 2003
index 2000 pcetot pce
index 2000 invtot inv
index 2000 govtot gov
index 2000 extot  ex
index 2000 imtot  im
```

From the base of 2003, we will have all of them except investment grow at a steady 3 percent per year to 2010. Investment will also have one component growing at this same rate but added to it – to make the results more interesting to view – will be a sine curve with a period of 2π years. Here are the commands for this operation.

```
fdates 1995 2010
# Create a time trend
f time = @cum(time,one,0)
f g03 = @exp(.03*(time-9))
f waves = g03 + .3*@sin(time-9)
fdates 2003 2010
index 2003 g03   pce
index 2003 waves inv
index 2003 g03   gov
index 2003 g03   ex
index 2003 g03   im
```

To add up the components of final demand to the total, we use the *vc* (for vector calculation) command. It can add up any number of vectors to get a total. Here are the commands.

```
# Add up the final demands
vc fd = pce+gov+inv+ex+im
show fd
```

We are now going to ignore the fact that the *AM* matrix is the same in all years – we could have changed it had we wanted to – and take its Leontief inverse in all years in the fdates range. The command

linv <square matrix> [year]

converts the square matrix into its Leontief inverse. For example,

linv A

converts $\boldsymbol{A}$ into $(I - A)^{-1}$. We then multiply this inverse by the final demand vector to compute the output vector. The *linv* command works over the fdate range unless the optional *year* argument is present.

```
# Take the Leontief inverse of the A matrix
mcopy LINV = AM
linv LINV
show LINV y 2000

# Compute total outputs
vc out = LINV*fd
show b.out
```

With the outputs known, we can compute the implied value-added of each type by each industry with the following commands. In them, the *vc* command will recognize that the dimensions of the vectors on the right are such that element-by-element multiplication makes sense and perform it.

```
# Compute Value added
# The following are element-by-element multiplication
vc dep = depc*out
vc lab = labc*out
vc cap = capc*out
vc ind = indc*out
show lab
```

As we went along, we showed results in spreadsheet-like grids to check that our answers were generally reasonable. Now we need to graph the results. In doing so, we use the fact that elements of vectors in a VAM file can be referred to in G simply by the name of the vector followed by a numeral. We can graph the second element of the *out* and *pce* vectors from the VAM file assigned as bank with the graph command like this:

```
gr  b.out2   b.pce2
```

If the VAM file is the default VAM file, we can omit the bank letter and period. Thus, in the instance just given, we could do just

```
gr out2  pce2
```

This way of working with a time series of elements of a vector works also for *type* and *r* commands and for the right-hand side of *f* or *fex* commands. Similarly, we can refer to an element of a matrix in a *type, graph,* or regression command or the right side of an *f* command an element of the matrix name followed by the row number, followed by a dot, followed by the column number. For example,

```
type AM3.5
```

will print to the screen the values of the element in the third row and fifth column of the *AM* matrix.

We can get a lot more graphs very quickly by use of G's *fadd* command. The name *fadd* is a contraction of "file-directed add command." It works with text substitution in a way that is very convenient in working with multisectoral models. The general form is

fadd <command file> <argument file>

In our case, the "command file" will be the following file, named GRAPHS.FAD:

```
vr 0
ti %3 %5
subti Output and Final demand
gname out%3
gr b.out%3 b.fd%3
subti Depreciation,Labor income, Capital income, Indirect taxes
gname va%3
gr b.dep%3 b.lab%3 b.cap%3 b.ind%3
ti
subti
```

and the argument file will be the same SECTORS.TTL file which we used for supplying row and column names for the matrices and vectors in the VAM file, namely:

```
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"
```

Note that some of the lines in the command file – for example, the second – have a % followed by a number. These numbers refer to "arguments" from the "argument" file. For example, on the first line of the argument file, argument 1 is *Agricul*, argument 2 is *;*, argument 3 is *1*, argument 4 is *e* , and argument 5 is *Agriculture* . Normally an argument is ended by a space or punctuation. Enclose arguments which contain spaces – such as the names of some sectors – in quotation marks. When the second line of the command file,

```
ti  %3  %5
```

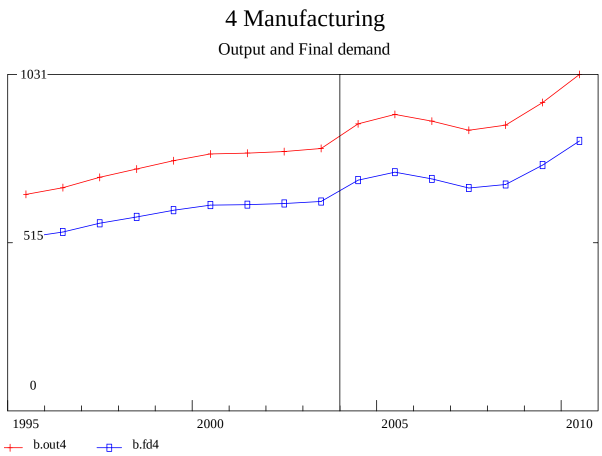
is executed with the arguments 3 and 5 from the first line of the argument file replacing the %3 and %5, the effect is that G executes the command

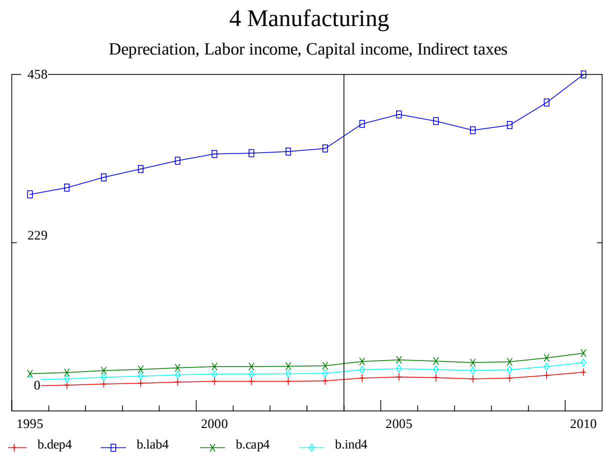
```
ti 1 Agriculture
```

The effect of the *fadd* command is that the entire command file is executed first with arguments from the first line of the argument file, then with the arguments from the second line of the argument file, and so on. Thus, with the single command

```
fadd  graphs.fad  sectors.ttl
```

G will draw for all sectors graphs like the two shown below for Manufacturing.





We have used some but not all of the G commands for matrix arithmetic in a VAM file. For reference, here are some others.

minv A	converts A into its inverse
madd A = B + C	adds B and C and stores in A
madd A = B - C	subtracts C from B and stores result in A
mmul A = B*C	multiply B and C and store result in A
mmul A = B’C	multiplies B transpose by C and stores result in A
mmul A = B&C	does element-by-element multiplication of B and C and stores in A
mmul A = B/C	element-by-element division of B by C stored in A
mtrans A B	the transpose of B is stored in A

In all of them, the command may be followed by an optional year in which to do the calculation; absent the year, the calculation is done for all years in the fdates range.

For tabulating the contents of a VAM file, we use exactly the same program, Compare, as we have used for macro models. It has, however, some capacities we have not used previously but now need. First of all, when we click Model | Tables on the G main menu, we need to choose “vam” as the type of the first bank, then give “hist” as its name; in the “Stub file” control, fill in “tiny”, and in the “Output file name” box type “tiny.out”.

The TINY.STB File

```
\dates 1995 2000 2005 2010 1995-2000 2000-2005 2005-2010
\pages off
\noformat
\title TINY G-ONLY MODEL, ILLUSTRATIVE FORECAST

; out  Output of Industries
&
out1 ;1 Agriculture
out2 ;2 Mining and quarrying
out3 ;3 Electricity and gas
out4 ;4 Manufacturing
out5 ;5 Commerce
out6 ;6 Transportation
out7 ;7 Services
out8 ;8 Government
;
\add tiny.tab pce "Personal Consumption Expenditure"
;
\add tiny.tab gov "Government Expenditures"
;
\add tiny.tab inv "Investment by Supplying Industry"
;

# The next line forces a new page
*
\matcfg Matlist.cfg
\center Matrix Listing
\row
\cutoff .001
\matlist 1-8
```

The first lines with the "\dates" command is familiar from macro models. Since I want to bring the results into a word processor for printing, I have turned off the page numbering and all commands to the printer in the next two lines. The "\title" command gives a title to be printed across the top of each page of output. As with macro stub files, a line beginning with a ";" just puts the rest of the line in the output file, and a "&" command puts a line of dates across the page. The next eight lines then show the output and its growth rates for the eight industries of the Tiny model for the dates specified.

We have not previously used Compare's *\add* command, which works just like G's *add* command, including a feature of the *add* command which we have not much used, namely, that it accepts arguments. The TINY.TAB file is shown in the box below. Instead of the lines in TINY.STB for printing the output of industries, we could have used the single line

```
\add  tiny.tab  out  "Output of Industries"
```

The effect would have been exactly the same.

The TINY.TAB File

```
; %1 %2
&
%11 ;1 Agriculture
%12 ;2 Mining and quarrying
%13 ;3 Electricity and gas
%14 ;4 Manufacturing
%15 ;5 Commerce
%16 ;6 Transportation
%17 ;7 Services
%18 ;8 Government
```

The TINY.TAB is a bit confusing to the eye because of the strings "%11" , "%12", and similar strings below them. To the eye, this may look like a reference to argument 11 or argument 12. But the computer knows that there can be only nine arguments and thus the third character in these strings is not part of the argument specification. It will read these as "argument 1 followed by the character 1" or "argument 1 followed by the character 2."

The results the tabulations described thus far are shown in the first box below.

The last five lines of TINY.STB are concerned with making a *matrix listing* from the VAM file. What is meant is best explained by looking at the results, which are shown for the first three industries in the second box below. For each row of the input-output table, the matrix listing shows each element of the identity:

output = intermediate demand + final demand.

Indeed, each element is shown in each year specified by the *\dates* command and growth rates of the element are shown for the periods specified by the same command. This matrix listing technique is important not only for the information it displays but also the consistency of the forecasts which it emphasizes.

TINY G-ONLY MODEL, ILLUSTRATIVE FORECAST

out Output of Industries

	1995	2000	2005	2010	95-00	00-05	05-10
1 Agriculture	140.7	164.0	189.6	216.7	3.1	2.9	2.7
2 Mining and quarrying	43.5	50.0	57.5	66.1	2.8	2.8	2.8
3 Electricity and gas	171.1	205.0	228.6	263.8	3.6	2.2	2.9
4 Manufacturing	663.0	787.0	908.3	1030.9	3.4	2.9	2.5
5 Commerce	331.0	401.0	439.9	510.0	3.8	1.9	3.0
6 Transportation	164.3	198.0	220.3	254.4	3.7	2.1	2.9
7 Services	555.8	667.0	738.2	854.7	3.6	2.0	2.9
8 Government	133.2	150.0	177.7	206.5	2.4	3.4	3.0

pce Personal Consumption Expenditure

	1995	2000	2005	2010	95-00	00-05	05-10
1 Agriculture	12.4	15.0	16.3	19.0	3.9	1.7	3.0
2 Mining and quarrying	1.6	2.0	2.2	2.5	3.9	1.7	3.0
3 Electricity and gas	65.9	80.0	87.1	101.2	3.9	1.7	3.0
4 Manufacturing	329.7	400.0	435.7	506.2	3.9	1.7	3.0
5 Commerce	288.5	350.0	381.2	442.9	3.9	1.7	3.0
6 Transportation	107.2	130.0	141.6	164.5	3.9	1.7	3.0
7 Services	412.1	500.0	544.6	632.7	3.9	1.7	3.0
8 Government	0.0	0.0	0.0	0.0	0.0	0.0	0.0

gov Government Expenditures

	1995	2000	2005	2010	95-00	00-05	05-10
1 Agriculture	0.9	1.0	1.2	1.4	2.4	3.4	3.0
2 Mining and quarrying	0.9	1.0	1.2	1.4	2.4	3.4	3.0
3 Electricity and gas	8.9	10.0	11.8	13.8	2.4	3.4	3.0
4 Manufacturing	71.0	80.0	94.8	110.1	2.4	3.4	3.0
5 Commerce	8.9	10.0	11.8	13.8	2.4	3.4	3.0
6 Transportation	17.8	20.0	23.7	27.5	2.4	3.4	3.0
7 Services	35.5	40.0	47.4	55.1	2.4	3.4	3.0
8 Government	133.2	150.0	177.7	206.5	2.4	3.4	3.0

inv Investment by supplying industry

	1995	2000	2005	2010	95-00	00-05	05-10
1 Agriculture	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2 Mining and quarrying	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3 Electricity and gas	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4 Manufacturing	147.2	200.0	240.2	257.5	6.1	3.7	1.4
5 Commerce	4.4	6.0	7.2	7.7	6.1	3.7	1.4
6 Transportation	5.9	8.0	9.6	10.3	6.1	3.7	1.4
7 Services	7.4	10.0	12.0	12.9	6.1	3.7	1.4
8 Government	0.0	0.0	0.0	0.0	0.0	0.0	0.0

```
                    TINY G-ONLY MODEL, ILLUSTRATIVE FORECAST

                                   Matrix Listing

                             Seller:   1 Agriculture
                                   1995   2000   2005   2010  95-00  00-05  05-10
                              Sales to Intermediate
  1 Agriculture                    17.2   20.0   23.1   26.4    3.1    2.9    2.7
  2 Mining and quarrying            0.9    1.0    1.2    1.3    2.8    2.8    2.8
  4 Manufacturing                  84.2  100.0  115.4  131.0    3.4    2.9    2.5
  5 Commerce                        4.1    5.0    5.5    6.4    3.8    1.9    3.0
  7 Services                        1.7    2.0    2.2    2.6    3.6    2.0    2.9
SUM: Intermediate                 108.1  128.0  147.4  167.7    3.4    2.8    2.6
                           Sales to Other Final Demand
Personal consumption expenditure   12.4   15.0   16.3   19.0    3.9    1.7    3.0
Government consumption              0.9    1.0    1.2    1.4    2.4    3.4    3.0
Exports                            33.1   40.0   45.4   52.8    3.8    2.5    3.0
Imports                           -13.7  -20.0  -20.8  -24.1    7.6    0.8    3.0
Output                            140.7  164.0  189.6  216.7    3.1    2.9    2.7

                        Seller:   2 Mining and quarrying
                                   1995   2000   2005   2010  95-00  00-05  05-10
                              Sales to Intermediate
  1 Agriculture                     3.4    4.0    4.6    5.3    3.1    2.9    2.7
  2 Mining and quarrying            2.6    3.0    3.5    4.0    2.8    2.8    2.8
  3 Electricity and gas            16.7   20.0   22.3   25.7    3.6    2.2    2.9
  4 Manufacturing                  12.6   15.0   17.3   19.6    3.4    2.9    2.5
  5 Commerce                        1.7    2.0    2.2    2.5    3.8    1.9    3.0
  6 Transportation                  0.8    1.0    1.1    1.3    3.7    2.1    2.9
  7 Services                        1.7    2.0    2.2    2.6    3.6    2.0    2.9
SUM: Intermediate                  39.5   47.0   53.2   61.0    3.5    2.5    2.7
                           Sales to Other Final Demand
Personal consumption expenditure    1.6    2.0    2.2    2.5    3.9    1.7    3.0
Government consumption              0.9    1.0    1.2    1.4    2.4    3.4    3.0
Exports                             8.3   10.0   11.4   13.2    3.8    2.5    3.0
Imports                            -6.8  -10.0  -10.4  -12.1    7.6    0.8    3.0
Output                             43.5   50.0   57.5   66.1    2.8    2.8    2.8

                         Seller:   3 Electricity and gas
                                   1995   2000   2005   2010  95-00  00-05  05-10
                              Sales to Intermediate
  1 Agriculture                     5.1    6.0    6.9    7.9    3.1    2.9    2.7
  2 Mining and quarrying            3.5    4.0    4.6    5.3    2.8    2.8    2.8
  3 Electricity and gas             8.3   10.0   11.1   12.9    3.6    2.2    2.9
  4 Manufacturing                  33.7   40.0   46.2   52.4    3.4    2.9    2.5
  5 Commerce                       16.5   20.0   21.9   25.4    3.8    1.9    3.0
  6 Transportation                  8.3   10.0   11.1   12.8    3.7    2.1    2.9
  7 Services                       20.8   25.0   27.7   32.0    3.6    2.0    2.9
SUM: Intermediate                  96.3  115.0  129.6  148.8    3.5    2.4    2.8
                           Sales to Other Final Demand
Personal consumption expenditure   65.9   80.0   87.1  101.2    3.9    1.7    3.0
Government consumption              8.9   10.0   11.8   13.8    2.4    3.4    3.0
```

Perhaps you are wondering how the Compare program knows what elements go into the identity and what are the names of the sectors and final demands. The answer was given to the program in the "matrix listing configuration file" whose name, MATLIST.CFG, was given to Compare in the command

```
\matcfg  matlist.cfg
```

The matrix listing configuration file to produce the matrix listing shown above is in the box below.

The MATLIST.CFG File for TINY

```
Matrix listing identity;out=AM*out+pce+gov+inv+ex+im
# Title file name for the rows of out, the lefthand side vector
out; "sectors.ttl"
# Title file names for matrix columns
AM; "sectors.ttl"
# headers for each term
header for out;    "Output"
header for AM*out; "Intermediate"
header for pce;    "Personal consumption expenditure"
header for gov;    "Government consumption"
header for inv;    "Investment"
header for ex;     "Exports"
header for im;     "Imports"
```

In the MATLIST.CFG file, any line beginning with a # is a comment and anything before the " ; " is likewise a comment. The first line gives the crucial identity on which the matrix listing is built. Recall that Compare has the VAM file and thus knows all the matrix and vector names and dimensions. It knows how to interpret correctly the expression "AM*out." The next line gives the file name of the sector titles for the vector on the left. Then follow the file names for column titles of any matricies for any matrices appearing in the identity. Here we have only one such matrix. Then come headers for each section of the table, a section being a vector or a matrix-vector product.

We return now to the TINY.CFG file to explain the last four lines, namely

```
\center Matrix Listing
\row
\cutoff .001
\matlist 1-8
```

The *\center* command centers the following text on the page. The command *\row* tells Compare to interpret the identity of the matlist configuration file as an identity in the rows. (The other possibility, *\column*, would be used for showing the identity – that holds only in current prices – between the value of the output of an industry and the sum of its intermediate inputs and value-added components.) The *\cutoff* command eliminates the printing of entries which account for less than the specified fraction of the total of the row or column being listed.

Finally, the *\matlist* command instructs Compare to make the matrix listing for the *group* of sectors following the command. This is our first encounter with the *group* concept which is quite useful in working with multisectoral models. A group is just a collection of integers; it can be specified in a rather flexible way. Our specification, 1 - 8, means every sector from 1 to 8. An equivalent specification would be 1 2 3 4 5 6 7 8. If we want just 1 to 3 and 6 to 8, we could write any of following:

1 - 3 6 - 8
1 2 3 6 7 8
1-8 (4 5)

The numbers in the parenthesis are stricken from the list created by the ranges to the left; the parenthesis can also include ranges.

You may now want to ask, “Shouldn’t we connect personal consumption expenditure to labor and capital income?” Of course we should, but to do so goes beyond what we can do in G alone. It requires the Interdyme modeling system, which is similar to Build but for multisectoral models. Everything we have covered in this section is directly relevant to working with Interdyme, but surely you need to pause here and be sure that you have mastered the large amount of information we have already covered. What better exercise could there be than to build your own Tiny model, so do exercise 1 for sure and the others to explore some other ideas.

Exercises

1. Make up the input-output table for your own imaginary economy in 2005. It should have five to ten sectors, but not eight. Use different sector titles. Make forecasts to 2025. Graph the forecasts and make tables of output and other vectors. Make a matrix listing.

2. For the economy of our example, what levels of output and use of primary imputs would be required for the final demand (40, 6, 100, 600, 400, 170, 700, 148)?

3. How much of each of the four factors does one dollar of each of the final demands contain?

4. Was this economy a next exporter or importer of depreciation?

5. What would happen to the prices of each of the eight products if all indirect taxes were eliminated?

6. Greenhouse gases are emitted by the production of the various sectors of our model economy. Measured in tons per billion dollars of output, the emission coefficients for the various sectors of our economy are

   2.1 1.3 6.1 1.8 1.0 4.3 0.8 0.0

   What is the emission of greenhouse gases per billion dollars of final demand for each of the eight products? How much is attributable to a billion dollars of each of the types of final demand -- consumption, government, etc.? Was this country a next exporter or importer of greenhouse gas emissions?

7. The input-output flow table illustrated in the text was for year A. A comparable table for the same country but for a later year, year B, may be found in the files YBF.DAT, YBX.DAT and YBV.DAT in the TINY.ZIP file. (You have to fix up the correct commands to get the data into G.) Price indexes for the eight sectors from year A to B are given by the vector

   (1.01 1.10 1.06 1.07 1.15 1.24 1.18 1.20),

   while the cost of labor increased twenty percent between the two years. (The price indexes are in the file PINDEX.DAT.) What has happened between the two years to total labor requirements for producing one unit of final demand for each product?

8. Return to exercise 7 but now consider that the depreciation and capital income are produced with material inputs in the proportions given by the investment vector of the year in question. Ignore the indirect taxes and imports. The reciprocals of the labor requirements are productivity indexes for the economy in producing the various products supplied to final demand.

   Exercises 7 and 8 illustrate correct ways of studying productivity of the economy in making various final products. As we noted in section 1, it is impossible to know what has happened to productivity in a single *industry*, because the industry may have reduced its primary inputs while increasing its intermediate inputs; and the double-deflation method, supposed to handle this problem, is totally fallacious. The same problem does not arise in looking at total labor required, indirectly as well as directly, for the production of each unit delivered to final demand, for if the direct supplier to final demand has shifted required labor to other industries by buying more intermediate goods, that indirect labor will be automatically picked up. Thus, input-output calculations may offer a way of studying trends in productivity by product which elude methods which do not take into account indirect effects.

9. Read the G help file for the *lint* command. Specify different (but consistent) values of the AM matrix, value-added coefficient vectors, and final demand vector shares for 1995 and 2010, use the *lint* command to interpolate values for other years, and repeat the calculations of the text with these time-varying coefficients. *Consistent* here means that the final-demand share columns sum to 1.0 and the sum of each column of AM plus the sum of the value-added shares in the same industry equals 1.0. Thus, these calculations are, in essence, in current prices, not constant prices.

5. Iterative Solutions of Input-output Equations

Before moving on to the Interdyme software, we must explain one of the mathematical techniques it uses extensively, namely the Seidel iterative solution of the input-output equations. In actual input-output computations, the Leontief inverse is seldom used, for the equations $q = Aq + f$ or $p = pA + v$ can be solved directly from the A matrix in about the same time required to multiply $(I - A)^{-1}$ by f or v. Thus, the effort of calculating $(I - A)^{-1}$ would be pointless. Moreover, for large matrices, many cells of A are zero. This fact can be exploited to reduce the computer storage required for the matrix. But the Leontief inverse will have non-zeroes nearly everywhere, so there is no way to reduce the space required for it. Further, changes to A are easily recorded and applied, but a change of one element in A can easily change all the elements in the inverse. Thus, from the point of view of solving the equations, nothing is gained and a good deal lost by computing the inverse.

How to solve the equations without the use of the inverse is the subject of this section. We will explain two methods of successive approximation, for it is worth knowing that both work even though we mainly use the second. The first, the simple iterative method, takes as a first approximation of q, $q^0 = f$. Then, given the nth approximation, q^n, the next approximation is

$$q^{n+1} = Aq^n + f. \tag{14.5.1}$$

If the process converges so that one q is indistinguishable from the previous one, then the vector to which it has converged is clearly the solution of the equation. In economic terms, we first set the output equal to the final demands. Then we increase it to allow for the intermediate goods needed by the first approximation and then incease it again for the intermediate goods needed for the second approximation, and so on.

It is clear from equation (14.5.1) that if the matrix A is non-negative and f is non-negative, then no element of q ever becomes negative in the course of the iterations. Thus, the conditions on A that insure the convergence also insure that a non-negative f leads to a non-negative q. Thus, our inquiry, initially motivated by considerations of practical computation, also provides an answer to the theoretical question of whether an economy could exist with a given f and A, for the economic interpretation of Aq is dependent on all elements of q being non-negative.

The second method, the Seidel process, takes the same first approximation, and then, to get the second approximation, solves first the first equation for q_1, given all the other elements of q. Then, using this new value of q_1 and the old values of q_3, q_4, *etc.*, solve the second equation for q_2, and so on. If the A matrix is triangular, that is, if all the entries above the main diagonal are zero, this method gives the right answer with one iteration. If it is not triangular, the whole process is repeated until little or no change occurs with each new iteration. While no actual input-output matrix is ever exactly triangular, the sectors can often be taken in an order which makes the matrix almost triangular, and this almost-triangularity speeds the convergence process.

Instead of starting this process with the final demands, it is also possible to start with any guess of q. In dynamic models, a good guess, namely the previous year's q is available. With a good starting point, four or five iterations of the Seidel process is usually sufficient to produce adequately accurate solutions. If twenty percent of the elements of A are non-zero -- a fairly typical situation -- we can make five iterations of the Seidel process in the same time which would be required to multiply f by the inverse if we had it.

If A is not an input-output matrix but just any old matrix you happen to meet on the street, there is not much chance that either of these methods will converge and give a solution. What then makes us so sure that they will converge for an input-output matrix? To discuss *convergence*, we need to be able to say how far apart two vectors are. The concept of the *norm of a vector* gives us that ability. We even need to be able to say how far a given vector is from the solution when we do not know what the solution is. The concept of the *norm of a matrix* enables us to turn that trick. We will now explain these two concepts.

We can say how far apart two vectors are if we can say how "long" a vector x is, that is, how long the line is which connects x with the origin or zero point. For if $\|x\|$ represents the length of any vector, then the length of the difference of two vectors a and b, $\|a-b\|$, serves as a measure of how far apart they are. How shall we weasure the length of a vector? In two dimensions, the usual lenght of the vector (x_1, x_2) is $\sqrt{x_1^2+x_2^2}$. This concept of length readily generalizes to vectors of any dimension by the definition $\|x\| = \sqrt{x'x}$. This formula, called the Euclidean length (or norm), gives one possible way of measuring length.

Why, however, do we bother to take the square root in the Euclidean norm? Because we certainly want *any* way of calculating the length of x to be such that multiplying each element of x by a scalar, λ, multiplies the length of x by the absolute value of λ:

(a) $\|\lambda x\| = |\lambda|\cdot\|x\|.$

Other properties which any definition of length should have are

(b) $\|0\| = 0 \text{ and } \|x\| > 0 \text{ if } x \neq 0$

and

(c) $\|x+y\| \leq \|x\| + \|y\|.$

Property (c) expresses the requirement that the shortest distance between any two points must be a straight line. Let us denote the points by x and $-y$. Then we must have

$$\|x-(-y)\| \leq \|x\| + \|-y\|,$$

since $\|x\|$ is the distance from x to 0 (the origin of the vector space) and $\|-y\|$ is the distance for 0 to -y, while $\|x-(-y)\|$ is the distance directly from x to -y. By applying property (a) to the second term on the right, this requirement may be written more simply as (c) above.

Any way of assigning a number, $\|x\|$, to each vector, x, of the vector space in such a way that (a), (b), and (c) are satisfied is called a *norm* of the space, and $\|x\|$ is read "the norm of x.". It is quite remarkable that we can often prove the convergence of a process in terms of a norm without knowing exactly which norm we are using. Besides the Euclidean norm, there are two more important examples of norms:

the l-norm: $\|x\| = \sum_{i=1}^{n} |x_i|$

the m-norm: $\|x\| = \max_i |x_i|$

You may easily verify that each of these norms has the required three properties, though the values they give as the norm of a given vector may be quite different. For example, the vector (1, -3, 2) has a Euclidean norm of 3.74, while its l-norm is 6 and its m norm is 3. (The l in l-norm refers to Henri Lebesgue, a French mathematician of the early years of the twentieth century.)

Exercise 9: Draw the unit circle for each of these three norms. (The unit circle is the locus of points with norm 1.)

With each of these three norms, if x^k, for k = 0, 1, 2, etc., is a sequence of vectors and x^* is a vector such that

$$\lim_{k \to \infty} \|x^k - x^*\| = 0,$$

then

$$\lim_{k \to \infty} x^k = x^*.$$

That is, convergence of a sequence of vectors in norm implies element-by-element convergence. This property is easily seen for the examples of the three norms and is a characteristic of finite dimensional vector spaces.

What we now want to show is that if q^* is a solution of the input-output equations, so that

$$q^* = Aq^* + f, \tag{14.5.2}$$

then the sequence q^0, q^1, q^2, ... defined by

$$q^{k+1} = Aq^k + f \tag{14.5.3}$$

converges in norm to q^*. Subtracting the first equation, (14.4.2), from the second, (14.4.3), gives

$$q^{k+1} - q^* = A(q^k - q^*),\ k = 1, 2, 3, \ldots . \tag{14.5.4}$$

If we have computed to iteration m, then setting $k = m$ in this equation gives

$$q^{m+1} - q^* = A(q^m - q^*).$$

But setting k = m+1 in (1.4.4) gives

$$q^{m+2} - q^* = A(q^{m+1} - q^*).$$

Together the last two equations imply

$$q^{m+2} - q^{*} = A(q^{m+1} - q^{*}) = A^{2}(q^{m} - q^{*}).$$

For any positive integer, p, similar reasoning applied p times gives

$$q^{m+p} - q^{*} = A^{p}(q^{m} - q^{*}). \qquad (14.5.5)$$

We would like to be able to show that the norm of the vector on the left of (14.5.5) goes to zero as p goes to infinity. To do so, we need to extend the concept of *norm* to matrices. We introduce that extension by a question:

Is there a number, call it $\|A\|$, such that

$$\|Ax\| \le \|A\|\cdot\|x\| \qquad (14.5.6)$$

for all x?

There are indeed such numbers, and we call the least of them (for any norm of the vectors) the *norm* of A. Intuitively speaking, the norm of the matrix A is the greatest "stretch" which multiplication by A performs on any vector. For the l-norm and m-norms of the vectors, the corresponding norms of a matrix are easily computed, as we shall see in a moment. Note that the norms of matrices also have the three basic properties of the norms of vectors:

a) $\|A\| = 0$ if and only if $A = 0$.
b) $\|\lambda A\| = |\lambda|\,\|A\|$
c) $\|A + B\| \le \|A\| + \|B\|$

plus a fourth, which can be easily verified from the definition

d) $\|AB\| \le \|A\|\,\|B\|$.

First, however, note that we can apply this inequality repeatedly to equation (14.5.5). After applying it p times, we have

$$\|q^{m+p} - q^{*}\| = \|A\|^{p}\cdot\|q^{m} - q^{*}\|$$

If we can show that $\|A\| \le 1$ for *some* norm, then $\|A\|^{p} \to 0$ as $p \to \infty$, and therefore $q^{k} \to q^{*}$ as $k \to \infty$, and the iterative calculations converge to the solution.

The norm of the n-by-n matrix A induced by the m-norm of vectors, and therefore called the m-norm of the matrix, is

$$\|A\|_{m} = \max_{i} \sum_{j=1} |a_{ij}|.$$

while the norm of A induced by the l-norm of vectors, and therefore called the l-norm of the matrix, is

$$\|A\|_{l} = \max_{j} \sum_{i=1}^{n} |a_{ij}|.$$

We shall prove the formula for the l-norm, and leave that for the m-norm as an exercise. (The Euclidean norm of A is more complicated and not of immediate concern to us. It is the largest characteristic root of A′A.) For the l-norm, let

$$\alpha = \max_j \sum_{i=1}^{n} |a_{ij}|.$$

Then $\|A\|_l \le \alpha$, because

$$\|Ax\|_l = \sum_{i=1}^{n} |\sum_{j=1}^{n} a_{ij}x_j| \le \sum_i \sum_j |a_{ij}|\cdot|x_j|$$
$$= \sum_j |x_j| \sum_i |a_{ij}| \le \sum_j |x_j| \alpha = \alpha\|x\|_l.$$

On the other hand, let k be the number of the column with the largest sum of absolute values, so that

$$\alpha = \sum_{i=1}^{n} |a_{ik}|$$

and then choose a vertor, x, with $x_k = 1$ and $x_j = 0$ for $j \ne k$. Then $\|x\| = 1$, and

$$\|Ax\|_l = \sum_i |\sum_i a_{ij}x_j| = \sum_i |a_{ik}| = \alpha = \alpha\|x\|.$$

Therefore, $\|A\|_l \ge \alpha$. But we have already shown the opposite inequality, so the only possibility is that $\|A\| = \alpha$.

If an input-output A matrix comes from an observed economy with a positive value-added in every industry, then the column sums of every column are less than 1.0 and therefore the l-norm of the matrix is less than 1. Thus, returning to the iterative solution of the input-output equations, we see that it will indeed converge if such is the source of A. Furthermore, in that case, $(I - A)^{-1}$ will be non-negative, because if we start from an *f* vector which is all zero except for a 1 in some position, the resulting solution will never have any opportunity to acquire any negative elements in the course of the iterative process. But the columns of $(I - A)^{-1}$ are precisely the solutions of such equations, so the whole matrix is non-negative.

The norm of the *A* matrix not only allows us to be sure that the iterative process converges, it also allows us to set an upper bound on how far we are from the solution at any stage. If, as before, q^k indicates approximation k, then

$$q^{k+p} - q^k = q^{k+1} - q^k + q^{k+2} - q^{k+1} + \ldots + q^{k+p} - q^{k+p-1} \tag{14.5.7}$$

But since

$$q^{m+1} = Aq^m + f \quad \text{and} \quad q^m = Aq^{m-1} + f$$

for any positive integer m, subtraction gives

$$q^{m+1} - q^m = A(q^m - q^{m-1}).$$

Repeatedly applying this equation gives

$$
\begin{aligned}
q^{k+1} - q^k &= A(q^k - q^{k-1}) \\
q^{k+2} - q^k &= A^2(q^k - q^{k-1}) \\
&\dots \\
q^{k+p} - q^{k+p-1} &= A^p(q^k - q^{k-1})
\end{aligned}
$$

and substitution in the above equation (1.4.7) gives

$$q^{k+p} - q^k = (A + A^2 + A^3 + \dots + A^p)(q^k - q^{k-1})$$

Taking the norms of both sides and applying properties c and d of the norms of matrices gives

$$
\begin{aligned}
\|q^{k+p} - q^k\| &\leq \|(A + A^2 + A^3 + \dots + A^p)\| \|q^k - q^{k-1}\| \\
&\leq (\|A\| + \|A\|^2 + \|A\|^3 + \dots + \|A\|^p) \|q^k - q^{k-1}\|.
\end{aligned}
$$

Now as $p \to \infty$, $q^{k+p} \to q^*$ and the sum of the geometric progression on the right goes to $\|A\|/(1 - \|A\|)$ because $\|A\| < 1$. Thus, when we have reached iteration k, we know that the distance to the true solution is less than $\|q^k - q^{k-1}\| \|A\|/(1 - \|A\|)$. In other words, when the differences of the successive approximations get small, we can be sure that we are close to the true solution.

Now suppose for a moment that A is a matrix in *physical* units -- with coefficients in units like kilowatt hours per pound -- so that column sums are meaningless and the l-norm perhaps much greater than 1. Further let *w* be an all-positive vector of the hours of labor -- the only primary input -- required per physical unit of output in each industry. Can an economy exist with this technology? In other words, if the vector f of final demands is all positive, will the vector of outputs, *q*, such that $q = Aq + f$ also be all positive? (Mathematically, it is quite possible for some element of q to be negative, but it is economic nonsense to run an industry at a negative level. Coal can be converted into electricity, but all the electricity in the world can't make a ton of coal.)

The answer to these questions lies in the solution of $p = pA + w$ (where *p* is a row vector). If *p* is all positive, then it can be thought of as a vector of prices (with an hour of work as the numeraire) at which each process has a positive value added. If we now change the units of measurement of output of each product to one "hour's worth," the coefficient matrix, say A^*, in these new units corresponding to A in the old units will have columns whose sums are each less than 1. Thus, in these units, the iterative procedure will converge. But the iterative procedure in the original units (with A) would give successive approximations which differ from those with A^* only in their units. Hence the process would converge in the original units as well and $(I - A)^{-1}$ will be non-negative. Since the Leontief inverse is non-negative, any vector of non-negative final demands can be met by non-negative levels of output of all the industries.

6. The Seidel Method and Triangulation.

As mentioned at the outset of the previous section, there is a variation of the iterative method, known as the Seidel method, which converges even faster. In it, one starts with f as the initial guess of the solution just as in the simple iterative method, but then solves the first equation for the first variable and puts this value into the guess, then solves the second equation for the second variable and puts that value into the guess, and so on. Formally,

$$q_i^{(k+1)} = \left(\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + f_i \right) / (1 - a_{ii}).$$

In input-output work, the *f* vector is generally non-negative as are the elements of the *A* matrix. Hence, in the simple iterative method, the approximate solutions form a monotonicly increasing sequence of vectors. The Seidel approximate solutions are also monotonicly increasing but are always larger than the corresponding simple iterative solution. Hence, it also converges to the solution and does so faster than does the simple iterative method.

If all the non-zero elements of A are on or below the main diagonal, A is said to be *triangular*. If A is triangular, one pass of the Seidel process is sufficient to reach the exact solution. If A is merely almost triangular, a few iterations will suffice for a good solution. It general, input-output matrices arrive from the statistical offices more or less triangulated in exactly the wrong way. They start with Agriculture first, later Textiles, then Apparel. The right order for a fast Seidel solution is the reverse, Apparel, Textiles, Agriculture. It is not, however, necessary to physically re-arrange the rows and columns. All that is necessary is to take the rows in the Seidel operation in the order that would make the matrix nearly triangular.

For large matrices, however, it may be convenient to have a mechanical way to generate an approximately triangular order. A simple but effective is to pick as the first industry the one which has the smallest ratio of intermediate to final demand in its row. Then move into final demand all the inputs into this industry and again pick from the remaining sectors the one with the lowest ratio of intermediate to final in its row. Continue until all industries have been selected.

Solving input-output equations by the Seidel method is not only generally much faster than inverting the $I - A$ matrix by Gauss-Jordan reduction, it may even be faster than multiplying $(I - A)^{-1}$ by f when $(I - A)^{-1}$ is already known. How can that be? It is common for the A matrix to be quite sparse. A 300-by-300 matrix may have some 9,000 non-zero elements, not 90,000. It can be stored in a "packed" form in which only non-zero elements are stored, and the Seidel algorithm can be written to use this packed form, so that only as many multiplications and additions are required per iteration as there are non-zero elements. Thus, if the Seidel process requires less than ten iterations in our example, it will require *less than* 90,000 multiplications and additions. The Leontief inverse, however, will generally have 90,000 non-zeroes and thus multiplying it by *f* involves exactly 90,000 multiplications and additions. To economize on both space and solution time, large, sparse matrices are thus best stored in a packed form; and equations involving them should be solved by the Seidel process without ever inverting the matrix.

Exercises

10. Using C, Fortran, Basic or any programming language you know, write a program to compute the triangular order of a matrix. Apply it to the flow matrix used as an example in this chapter. Write the results as a vector of integers, the first being the number of the equation to be taken first; the second, that of the equation to be taken second, etc.

11. Write a program to use the Seidel method to solve input-output equations, taking the equations in the order specified by the vector produced in exercise 7. Apply the program to solve exercise 1 earlier in this chapter. (Bump has a Seidel method. Try to create yours without looking at it.)

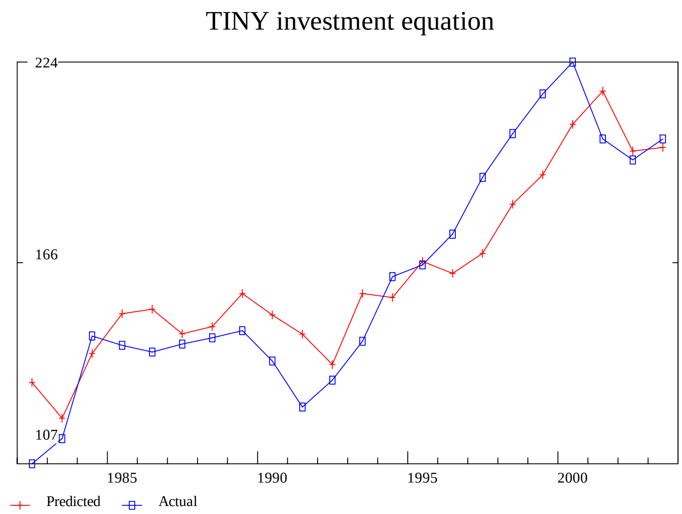
7. Introduction to Interdyme

In section 4, we became acquainted with the VAM file and saw that G could do a number of calculations with the matrices and vectors in these files. By the end, however, we came up against the limit of G by itself. To integrate regression equation and input-output we have to go on to Interdyme. Interdyme is a collection of C++ programs which make it easy to construct interindustry dynamic models involving regression equations, input-output computations with matrix algebra, and lag relationships that provide the dynamics. In this section, we will introduce Interdyme by building a simple Interdyme model of TINY with just one regression equation and rudimentary institutional accounts. We can conveniently begin with the most familiar part, the regression equation and the accounting identities.

The regression equation is estimated in G by the following file

```
catch invtot.cat
save invtot.sav
ti TINY investment equation
dfreq 1
f ub20 = @cum(ub20,1.,.20)
f capstk = @cum(invcum, invtot[1], .20)/ub20
f delGDP = gdp - gdp[1]
con 2000 1.03 = a1
lim 1982 2003
r invtot = ! capstk, delGDP[1], delGDP[2]
save off
gr *
catch off
```

with the following results




```
:                            TINY investment equation
  SEE    =      13.51 RSQ   = 0.8360 RHO =   0.60 Obser  =   22 from 1982.000
  SEE+1  =      11.45 RBSQ  = 0.8187 DW  =   0.79 DoFree =   19 to   2003.000
  MAPE   =       7.24
    Variable name            Reg-Coef  Mexval  Elas   NorRes     Mean   Beta
  0 invtot                 - - - - - - - - - - - - - - - - -   161.06 - - -
  1 capstk                   0.99562   402.8   0.89     1.52   144.50
  2 delGDP[1]                0.35900    17.0   0.11     1.00    47.67  0.231
  3 delGDP[2]                0.03007     0.1   0.01     1.00    47.06  0.020
```

The constraint has produced a coefficient on the *capstk* term very close to 1.0, so capital wearing out will be replace replaced dollar for dollar. The current year value of *delGDP* was omitted both to avoid sinultaneous equation bias and to speed convergence of the model.

The national accounts are given by the file ACCOUNT.SAV as follows:

```
# The Accountant for Tiny
# Personal interest and dividends
fex pintdivrat = pintdiv/capinc
f pintdiv = pintdivrat*capinc
# Personal income
f pi = labinc + pintdiv + pgovtran
# Personal taxes
fex ptaxrat = ptax/pi
f ptax = ptaxrat*pi
# Personal disposable income
f pdisinc = pi - ptax
# Personal saving
fex psavrat = psav/pdisinc
f psav = psavrat*pdisinc
f pcetot = pdisinc - psav
# Government income
f ginc = indtax + ptax
# Government saving
f gsav = ginc - govtot - pgovtran
# Business saving
f bsav = capinc - pintdiv - invtot
# RoW saving
f RoWsav = imtot - extot
```

All of this file should be familiar from the Master file of macromodels and the previous discussion of the Institutional accounts in this chapter.

Both of these .SAV files will be passed through the IdBuild program, the Interdyme version of the Build program. Running and explaining IdBuild reverse the old saying "Easier said than done." It is much easier to run IdBuild than to explain what all it does. Basically, IdBuild converts the G-estimated parts of the model into C++ code. Clicking Model | IdBuild not only runs IdBuild but also compiles the C++ code it has written and combines it with the other C++ modules of the Interdyme system to make an executable program. Thus, running IdBuild just requires two mouse clicks. Recounting what it does will take us a little longer, but some understanding of the results is necessary to play your creative role in writing MODEL.CPP that pulls together the pieces of the Interdyme system.

First, we must note that IdBuild is not so omniscient as Build and needs our help on one point. Build knows about *all* the variables and equations in the macromodel it is building. Therefore, it can arrange for getting all of them into the model, can figure out which ones are endogenous, and can write a file for making quick projections of all those which remain exogenous. Because part of an Interdyme model may be written directly in C++ without going through IdBuild, IdBuild does not necessarily know about all the variables in the model and does not know about the possible definition of some variables in the C++ code. To inform IdBuild of the variables which are used in the C++ code or otherwise needed in the data bank produced by IdBuild but not used in any of the .SAV files processed by IdBuild is the first function of a file called PSEUDO.SAV. In TINY, there is only one variable of this sort, *timet,* which is used in the regressions for projecting exogenous variables. The second function of PSEUDO.SAV is to tell IdBuild which variables which will be defined in the C++ code and therefore do not need regressions in EXOGALL.REG for making up quick, mechanical projections. In TINY, there are five such variables, as we shall see below, namely, *gdp, deprec, labinc, capinc,* and *indtax.* Here is PSEUDO.SAV for TINY:

```
# Time
f timet = timet
# GDP Gross domestic product
f gdp = gdp
# Depreciation income
f deprec = deprec
# Capital income
f capinc = capinc
# Labor income
f labinc = labinc
# Indirect taxes
f indtax = indtax
```

With these files in place, we are ready to look at the MASTER file. It is just the following:

```
#  Master File for TINY
iadd invtot.sav
iadd account.sav
iadd pseudo.sav
end
```

The only commands in the Master file for IdBuild are comments, *iadd* commands to bring in the various .SAV files, and the *end* command to signal the end of the commands.

IdBuild requires a configuration file called BUILD.CFG; in it, I recommend that you make the name of the workspace bank HIST, which is the way the file is supplied.

Once these files are ready, IdBuild can then be run. I would suggest that, for the moment, you do so from within G by the command

```
dos c:\pdg\idbuild master
```

Later, when you have written the MODEL.CPP file, you can run IdBuild and compile and link MODEL.CPP and other parts of the Interdyme program by clicking Model | IdBuild. We will get to that point soon. Right now we should have a quick look at the output of IdBuild. It is reminiscent of that of Build. The EXOGALL.REG file, for example, begins

```
bank hist b
mode f
tdates <StartType> <EndType>
limits <StartReg> <EndReg> <EndForecast>
ti govtot
r b.govtot = timet
gr *
f govtot = depvar
save govtot.xog
sty govtot
save off
...
```

As with a macromodel, areas marked with the <...> in the third and fourth lines must be replaced by dates and the file then passed through G to generate for each variable a .XOG file with exogenous forecast for each of the exogenous variables. The file RUN.XOG will contain an *add* command for each of these files. For TINY, RUN.XOG is

```
dos copy hist.* base.*
wsb base
add govtot.xog
add extot.xog
add imtot.xog
add pgovtran.xog
add pintdivrat.xog
add ptaxrat.xog
add psavrat.xog
wsb ws
```

Note the effect of the first two and the last commands when executed in G. As noted above, HIST is the recommended name (specified in BUILD.CFG) for the workspace bank generated by IdBuild. The first command above copies this G bank to a G bank with the name BASE. The second line makes this bank the workspace bank of G. The subsequent lines add the files with the projections of all the exogenous variables. These projections go into the G workspace bank, namely, into BASE. The last command makes WS the workspace bank of G and thereby frees the BASE bank. As in the case of macromodels, you should check the mechanical exogenous projections, shape them to your preferences, and put revised versions into files with the extension .XG, and make a revised RUN.XOG, calling it something like BASE.XG. For the moment, we will let well enough alone and just use RUN.XOG with the mechanically generated projections.

Like Build, IdBuild also writes RUN.GR, a command file for G which will graph all timeseries variables handeled through IdBuild.

Running IdBuild generates a number of other files to facilitate the writing of the C++ program to run the model. One of these is TSERIES.INC as follows:

```
GLOBAL Tseries timet, invtot, pcetot, gdp, labinc, capinc, deprec, indtax,
pintdiv, pi, ptax, pdisinc, psav, gsav, bsav, RoWsav,
netim, govtot, extot, imtot, pgovtran, ub20, invcum, capstk,
delGDP, pintdivrat, ptaxrat, psavrat, ginc;
```

This is just a listing of all the timeseries variables in the model in a format suitable for use in the model.

Like Build, IdBuild also writes a file of C++ code called HEART.CPP. It is shown in the box below, in somewhat abbreviated form, for TINY. Wherever four dots occur, thus .... , lines have been omitted which just repeat for the other variables listed above in TSERIES.INC the same function above and below the four dots for *invtot* and *ginc*. The Master file had the two following lines:

```
iadd invtot.sav
iadd account.sav
```

Each of these resulted in a corresponding function of C++ code in the HEART.CPP file, namely *invtotf()* and *accountf()*, respectively. These are functions which can be called from MODEL.CPP, the user-written part of Interdyme systerm. The function name is created by adding an *f* to the end of the name of the file. A glance at *accountf()* shows that it is just an adaptation for C++ of the G commands in

ACCOUNT.SAV. The function *invtotf()* is the same for INVTOT.SAV with the extra feature that the numerical values of the regression coefficients have been stripped off and put in a separate file, HEART.DAT, which will automatically be read to supply values to the *coef* variable. The regression coefficients are treated this way so that they can potentially be varied in the course of optimizing the fit of the model, just as was done with macromodels. IdBuild recognizes the special function of the line

```
iadd pseudo.sav
```

in the MASTER file and does not generate a corresponding function in HEART.CPP.

Also in the HEART.CPP file are the C++ functions *tserin()* and *uptseries().* The first of these reads in the historical values and exogenous projections of the time series variables; it is executed at the beginning of any run of the model. The second, *uptseries(),* is called at the beginning of calculation of the forecast for any year. It looks at the starting value of each of the time series variables and if it is -.000001, the value G uses to indicate a missing number, then it replaces that missing value with the value of the series in the previous year. For example, in TINY, no exogenous projection is made for Personal income, since it is endogenous. Thus, its starting value in 2004 would be -.000001. If the model starts with this very bad guess of Personal income, it eventually converges to the correct value, but it will converge much faster if it starts with a good guess, and the previous year's value is usually a pretty good guess. Note that exogenously projected values will not be affected, because they will not be -.000001.

A final file created by IdBuild in CALLALL.CPP , a bit of C++ code which calls all of the functions written by IdBuild in the HEART.CPP file. It is used in conjunction with managing rho adjustments as will be explained below in conjunction with the MODEL.CPP file. Here is the important part of CALLALL.CPP for TINY.

```
void callall()
{
invtotf();
accountf();
}
```

TINY’S HEART.CPP FILE

```
#include "dymesys.h"
#include "heart.h"
extern short t;
extern float **coef;
FILE *fmatrix;
float depend;
#include "tseries.inc"
/* end of standard prolog */
void invtotf()
{
/*  TINY investment equation */
 ub20[t]=cum(ub20,1.,.20);
 capstk[t]=cum(invcum, invtot[t-1],.20)/ ub20[t];
 delGDP[t]= gdp[t]- gdp[t-1];
/* invtot */ depend = coef[0][0]*capstk[t]+coef[0][1]*delGDP[t-1]+coef[0][2]*delGDP[t-2];
invtot.modify(depend);
}
void accountf()
{
 pintdiv[t]= pintdivrat[t]* capinc[t];
 pi[t]= labinc[t]+ pintdiv[t]+ pgovtran[t];
 ptax[t]= ptaxrat[t]* pi[t];
 pdisinc[t]= pi[t]- ptax[t];
 psav[t]= psavrat[t]* pdisinc[t];
 pcetot[t]= pdisinc[t]- psav[t];
 ginc[t]= indtax[t]+ ptax[t];
 gsav[t]= ginc[t]- govtot[t]- pgovtran[t];
 bsav[t]= capinc[t]- pintdiv[t]- invtot[t];
 RoWsav[t]= imtot[t]- extot[t];
}

void tserin()
{
timet.in("timet");
invtot.in("invtot");
....
ginc.in("ginc");
}
void uptseries()
{
//Function to replace missing values or macro vrariable with lagged values.
if(timet[t]< .0000009 && timet[t]> -.0000011) timet[t] = timet[t-1];
if(invtot[t]< .0000009 && invtot[t]> -.0000011) invtot[t] = invtot[t-1];
....
if(ginc[t]< .0000009 && ginc[t]> -.0000011) ginc[t] = ginc[t-1];
}
```

So far, there has been a lot of similarity between building Interdyme models and building macromodels with G and Build. Now we venture into new territory with the writing of the MODEL.CPP file. The box below shows the part of this file that distinguishes TINY from any other model built with Interdyme. I have more than once encountered the reaction, "C++ is hard; I don't have time to learn it, so I'll just stay with the models I can build in Excel." It is true that some of the "far out" tricks of C++ can be rather arcane, but the tricky part of C++ has been done for you in Interdyme. The code that you will need to write is in no way more complicated than the simplest level of Basic, Fortran, or C – and a lot simpler than writing VBA macros for Excel. But with the Interdyme infrastructure, you can easily write the code for the matrix and vector operations that are essential for working with input-output models but are tedious to program in those simpler languages.

The box below shows the C++ code that defines TINY. First of all, you need to know a few things about C++ grammar. Anything following a // on the same line is a comment, useful for humans but ignored by the computer. A comment extending over more than one line is begun with a /* and ended with a */. Every C++ statement ends with a semicolon. More than one statement can be put on one line or a single statement can be broken onto two or more lines between words. C and C++ are case sensitive: x is not the same as X. For any variable *x, x++* means "add 1 to x." Before a variable name, like *Iteration* or *oldinvtot*, can be used, we must declare what kind of variable it is by statements like

```
int Iteration; // declare Iteration to be an integer
float oldinvtot; //declare oldinvtot to be a floating point, real number
```

A group of statements, indicated by enclosing the statements in curly braces like these { }, can be used anywhere a single statement could be used, notably in *for, do, while, if, else*, and *else if* constructions. The **for** loop that fills most of the box of TINY code is a good illustration of this point. It looks something like this:

```
for (t = godate; t<= stopdate; t++) {
  ....
  }
```

where the .... represents many lines of code. The command means: start *t* off equal to *godate*, which other coding part of the Interdyme system will have made the beginning year of the run you are making, something like 1995, then do all the statements represented by the .... with that value of *t*, then increment *t* by 1 – that's the *t++* near the end of the line – and do all the statements represented by the .... with that value of *t*, and keep doing all this as long as *t* is less than or equal to *stopdate*, a number that other parts of Interdyme will have made equal to the last year of your run, something like 2010. Now the generic form of the **for** loop is written

for (*initialization, condition, increment*) *statement*

It starts off one or more variables in the *initialization*, checks that the *condition* is true and, if it is, executes the *statement*, then revises the variable that was initialized as prescribed by the *increment*, checks the *condition*, and executes the *statement*, and so on as long as the *condition* is true. When the *condition* is no longer true, control passes to the next statement below the **for** loop. How the *initialization, condition*, and *increment* work in the example is clear enough, but our particular point here is that the *statement* executed is the compound statement composed of all the simple statements enclosed by the braces that open right after the closing parenthesis of the **for** statement. I should also mention that indentation in C and C++ is solely for the benefit of the human reader; it doesn't matter to the computer.

The construction

if (*condition*) *statement*
else if (*condition*) *statement*
else *statement*

works in an entirely similar way. Any number of **else if** lines may be used or they and the **else** line may be absent altogether. In writing the *conditions*, any of the following operators may be used:

== is equal to
!= not equal
< is less than
> is greater than
>= is greater than or equal
<= is less than or equal
| | or
&& and
! not

Besides the **for** keyword, the keywords **while** and **do** can also be used to write loops. The general form of the **while** is

while(*condition*) *statement*

which executes the *statement* as long as the *condition* is true. There is an example in the TINY code. For the **do** keyword, the syntax is

do *statement* **while** (*condition*);

The *statement* executes until the *condition* becomes false. Since the *conditon* is tested *after* each pass through the loop, the loop will execute at least once. In practice, the **do** loop is seldom used.

Finally, we need to mention three statements for jumping about in the code. The most commonly used is **break**, which breaks out of the innermost loop where it is found. There will be an example in the TINY code below. The **continue** statement shifts control to the end of the innermost loop, while

goto *label;*

sends control to the label. The label is a line with one word ending in a colon, like *top:* or *finish:* . In good C++ style, the **goto** is seldom used; but it is useful when breaking out of a deep nest of loops. (If you have ever used Fortran, please note that the function of the C **continue** statement is totally unlike that of the Fortran CONTINUE statement.)

That is about all you need to know about C++ in general to use Interdyme. There are, however, some Interdyme-specific functions and variables you need to know about. The first of these is the variable *MaxFlag*; it is a single letter, either *y* or *n* for "yes" or "no". It will be *y* only if the user has specified that we are to do an optimization. We will assume that it is *n*. In that case, the *printf()* command writes to the screen the year which is about to be computed. (For the full capabilities of *printf()* or the C++ alternative keyword **out**, you can consult the help file of your C++ compiler.) The program then calls the Interdyme functions *load(t)* to load into the computer's random access memory (RAM) from the computer's hard disk the starting values for year *t* of all vectors and matrices in the VAM file on the hard disk. At the bottom of the **for** loop, the *store(t)* functions stores their newly computed values back to the VAM file. The call to the function *uptseries()* checks all the timeseries variables, and where it finds a value missing in period *t* it inserts the value of that timeseries from period the previous period. As noted above, this replacement gets the iterative process of solution off to a good start.

The Core of TINY's MODEL.CPP File

```
for (t = godate; t<= stopdate; t++) {
    if (MaxFlag == 'n') printf("%d ",t);
    // Load all vectors and matrices.
    load(t);
    uptseries();
    // Start of code particular to TINY:
    Iteration = 0;
    // The loop for convergence on pcetot and invtot.
    while(Iteration < 20){
      Iteration++;
      oldinvtot = invtot[t]; oldpcetot = pcetot[t];
      if(t>= MacEqStartDate)
          invtotf();
      inv = invtot[t]*invc;      pce = pcetot[t]*pcec;
      gov = govtot[t]*govc;      im  = imtot[t]*imc;
      ex  = extot[t]*exc;
      // Add up final demand vectors
      fd = pce + gov + inv + ex + im;
      // Solve input-output equations by Seidel method
      Seidel(AM, out, fd, triang, toler);
      // Compute value-added vectors
      // The Interdyme ebemul() function does element-by-element multiplication
      dep = ebemul(depc,out);  lab = ebemul(labc,out);
      cap = ebemul(capc,out);  ind = ebemul(indc,out);
      // The Accountant for Tiny
      gdp[t] = fd.sum();
      if(t>MacEqStartDate){
          deprec[t] = dep.sum(); labinc[t] = lab.sum();
          capinc[t] = cap.sum(); indtax[t] = ind.sum();
          accountf();
          }
      // Form the convergence test
      invdif = fabs(invtot[t] - oldinvtot);
      pcedif = fabs(pcetot[t]- oldpcetot);
      printf("Iter %2d pce = %7.1f pcedif = %6.2f invdif = %6.2f\n",Iteration,
          pcetot[t],pcedif,invdif);
      if(invdif < .5 && pcedif < .5) break;
      }
    // End of code particular to TINY
    // Here when both Investment and PCE have converged
    // Standard end of the spin() function:
    if(MaxFlag == 'y')
            shiftback(t);
    else{
        // Store the values of vectors and matrices for this period.
        store(t);
        printf("\n");
        }
    }
```

At this point, we begin the code that is particular to TINY. Through the general Interdyme structure, the person running the model provides a value for the variable *MacEqStartDate*, the date at which the macroequations of the model begin to be computed. TINY has only one macroequation, namely that for *invtot(t)*, total investment. Since we have known accounts up through 2003, we will set *MacEqStartDate* equal to 2003. (How we do so, we will see shortly.) Prior to this date, the Interdyme model for TINY produces the same results as did the G-only model. In 2003, only *invtot* and *gdp* may be different. Other macro variables are not affected because the national income accountant function, *accountf()*, is not called until the next year. In practice, we will normally be setting the rho-adjustment factor for all of the macro variable regression equations in this year – how that is done will be explained below – so even those variables will not differ from historical values in this last year of historical macroeconomic data. The fun begins in the next year.

Once *t* has passed the *MacEqStartDate*, the final demands determine output (via the Seidel input-output calculations), but output determines value-added and value-added determines (via the institutional accounts) Personal income, which determines total Personal consumption expenditure, which is one of the final demands. We solve this circularity, just as we did in macromodels, by iteration. That is, we start off with one set of final demand totals – *pcetot[t], invtot[t], govtot[t], extot[t], imtot[t]* – and compute along until will have calculated new values. We then compare the new values with the old, and if the differences exceed the tolerances we set, we go back to compute, with the new values of the variables, final demands, outputs, value added, and the variables just listed, and repeat the process until convergence is reached or the iteration counter exceeds 20, whereupon we go on to the next year. Since *govtot[t], extot[t], imtot[t]* are exogenous, they will not vary from iteration to iteration, so only the values of *pcetot[t]* and *invtot[t]* are checked for convergence. In fact, the form of the regression equation we estimated has no dependence of *invtot[t]* on variables in period *t*, so we do not for the present model need to check *invtot[t]*, but we leave the check because in principle there could be such a dependency. The lines of code that drive the looping process are:

```
Iteration = 0;
// The loop for convergence on pcetot and invtot.
 while(Iteration < 20){
  Iteration++;
  oldinvtot = invtot[t]; oldpcetot = pcetot[t];

  /*****************************************************************
   The substance of the model, which follows here, has been cut out to
   emphasize the testing of convergence. This substance computes new
   values of invtot[t] and pcetot[t].
   *****************************************************************/

  // Form the convergence test
  invdif = fabs(invtot[t] - oldinvtot);
  pcedif = fabs(pcetot[t]- oldpcetot);
  printf("Iter %2d pce = %7.1f pcedif = %6.2f invdif = %6.2f\n",Iteration,
      pcetot[t],pcedif,invdif);
  if(invdif < .5 && pcedif < .5) break;
  }
```

The first line initializes an integer, *Iteration,* to 0. The

```
while(Iteration < 20){
```

line sets up the loop that iteratively solves the circularity just explained. The first line within the loop increments *Iteration* by one, so we see immediately that this loop will not be executed more than 20 times. We then store away the starting values of *invtot[t]* and *pcetot[t]*. Then we execute the substance of the model, which results in changing these values. When that work is complete, we use the standard

C++ function *fabs()* to take the absolute value of the difference between the previous and the new values of these two variables. If both of those values are less than .5, we break out of the **while** loop. More complicated models may have more complicated convergence tests, but TINY's example is pretty typical. Once convergence has been reached, the main thing to be done is to call *store(t)*, which stores back to the computer's hard disk this period's values of the vectors and matrices.

With the iterative solution of the circularity understood, we turn to the the substance of the model. It is easily followed in the code in the box below. The Interdyme code, nestled away in other modules (.CPP files) where you do not need to bother reading it, defines what a Vector and Matrix are and instructs the computer in how to read them, add or subtract them, multiply a Matrix by a Vector, sum up the elements of a Vector, and do both inner-product and element-by-element multiplication of two Vectors. Let me emphasize: Matrix and Vector are not general C++ features; it is the other modules of Interdyme working in the background that give you these handy tools for writing the code for input-output models. In fact, though we have used here only vector addition, Interdyme has a rather complete library of matrix and vector routines.

The Interdyme function *Seidel(AM, out, fd, triang, toler)* solves by the Seidel method the equation

*out = AM*out + fd*

using the order of equations specified by the vector *triang* and with the convergence tolerance specified by *toler*. The value of integer vector, or *ivector*, is given by the file TRIANG.IV. For TINY, its value is

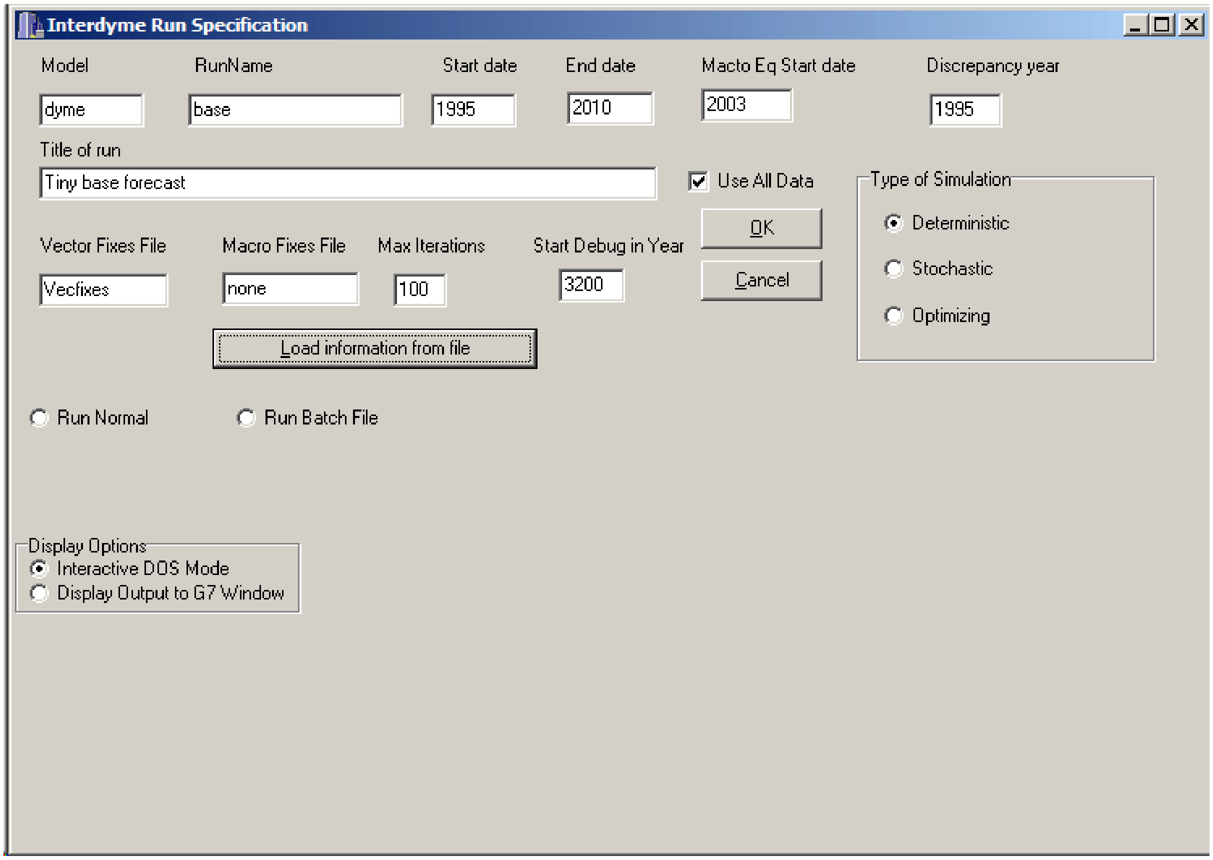
8 7 5 6 4 3 1 2 .

Th Interdyme function *ebemul()* does an element-by-element multiplication of two Vectors and gives a Vector as its output. The floating point absolute value function, *fabs()*, is a standard C++ function.

With the file MODEL.CPP written, running the model is easy. In G, click Model | IdBuild. This one click will not only run IdBuild, which writes HEART.CPP, EXOGALL.REG and RUN.XOG, but also compiles MODEL.CPP, HEART.CPP and any other modules that need to be comiled, links them together and produces the DYME.EXE, the executable file for the program. As described above, you need to use EXOGALL.REG to make projections of the exogenous variables, exactly as with macro models. When the .XOG files and RUN.XOG are ready, you run the Interdyme model by clicking Model | Run Dyme . That brings up a window which you should fill out as shown below. (When you have done this once, you can get the information back by clicking the "Load from file" bar and selecting dyme.cfg.) Note that the "Start date" and "End date" boxes provide *godate* and *stopdate* variables of the C++ code, while the "Macro Eq Start Date" box provides the *MacEqStartDate* variable. Putting the dot in the "Deterministic" radio button in the "Type of Solution" panel on the right will set the *MaxFlag* variable to *n*, so that no optimization is attempted. When all is ready, click the OK button. G will then write the DYME.CFG file and execute the command-line command

```
dyme dyme.cfg
```

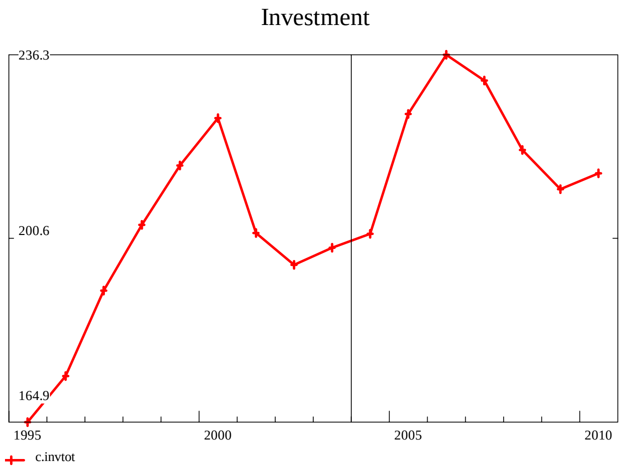
Results flash past quickly in the black screen. They are of interest only if you are having problems or want to see how rapidly the model converges. For the latter purpose, here is what we appears on the screen in the first year when the macro equations are turned on:

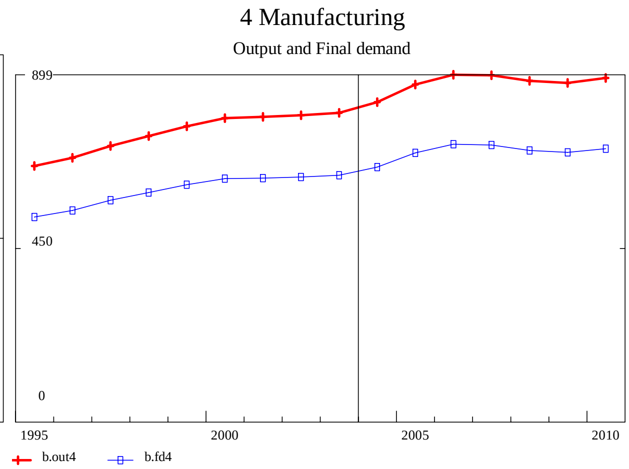



```
2004
Seidel iterations: 4 Iter  1 pce =  1544.4 pcedif =  29.39 invdif =   2.67
Seidel iterations: 3 Iter  2 pce =  1563.5 pcedif =  19.08 invdif =   0.00
Seidel iterations: 2 Iter  3 pce =  1575.8 pcedif =  12.32 invdif =   0.00
Seidel iterations: 2 Iter  4 pce =  1583.8 pcedif =   8.02 invdif =   0.00
Seidel iterations: 2 Iter  5 pce =  1589.0 pcedif =   5.21 invdif =   0.00
Seidel iterations: 2 Iter  6 pce =  1592.4 pcedif =   3.39 invdif =   0.00
Seidel iterations: 2 Iter  7 pce =  1594.6 pcedif =   2.21 invdif =   0.00
Seidel iterations: 1 Iter  8 pce =  1595.9 pcedif =   1.32 invdif =   0.00
Seidel iterations: 1 Iter  9 pce =  1596.8 pcedif =   0.90 invdif =   0.00
Seidel iterations: 1 Iter 10 pce =  1597.4 pcedif =   0.61 invdif =   0.00
Seidel iterations: 1 Iter 11 pce =  1597.9 pcedif =   0.41 invdif =   0.00
```

The Seidel iteration count is printed by the Seidel routine. It indicates that generally very few iterations are needed. The PCE convergence, on the other hand, is rather slow; a faster converging algorithm could be easily devised, but we will not pursue that direction at present.

Here are the graphs of total investment and the output of manufacturing for this simplest version of the TINY Interdyme model.





The results are clearly looking more like model results than they did with all demand exogenous. But before we go further, it is time for a few exercises.

Exercises

1. Make alternative forecasts for some of the exogenous variables in TINY, run the model with them, graph the results, and show them in tables.

2. Estimate a regression equation for total imports as a function GDP, put it into the model, run the model and compare graphically the results with those without your function.

Before you can build a TINY-like model for your own imaginary or real economy, we have to deal with one more basic question: How is the Interdyme program connected with the VAM file? How does it know, for example, that there are vectors in TINY's VAM file named *fd, pce, gov, ind* and so on? The answer is that user has to tell it the names of these vectors and matrices; it, however, will figure out from the VAM file their dimension. This identification has to be done in two steps. First, there is a file called USER.H; the one for TINY is shown in the box below.

USER.H for the TINY I Model

```
// From DYME.CFG and opening screen:
GLOBAL char RunTitle[80],CfgFileName[80],VamFileName[80],GbankName[80],
   VecFixFileName[80],MacFixFileName[80];

GLOBAL char* outfix; // Determines how Seidel will determine output

// Vector declaration:
   GLOBAL Vector out ,pce, gov, inv, ex, im, fd, dep, depc, lab, labc, cap, capc,
   ind ,indc ,pcec, invc ,govc ,exc ,imc, x, y;

// Matrix declaration
   GLOBAL Matrix AM;

   GLOBAL IVector triang;
```

All the variables declared in USER.H are “global,” C - jargon for variables which can be accessed from anywhere in the program. This fact is indicated by the word GLOBAL in front of these variables. To explain exactly what is going on here requires a digression which you can skip if you are familiar with C or don’t want to bother with the details of exactly how this GLOBAL word works, so I will put it in slightly smaller type.

Unlike Basic and Fortran, C and C++ are strictly “typed” languages. That is, the nature of every variable must be declared before it can be used. Typical declarations are:

```
char sex;
int zip;
float height, weight;
char name[40];
```

The variable *sex* is a single character, presumably M or F; the variable *zip* is an integer, something like 20742; *height* and *weight* are floating point numbers, that is, numbers that potentially have a fractional component like 72.5 or 217.8. The variable *name* is a string of up to 40 characters, something like “Thomas”. If a variable is declared inside a function, it is *local* to that function. Other functions know nothing about it. But a variable declared outside of all functions, typically near the top of a file containing C code, is *global* and can be accessed by all functions in that file. A large program such as Interdyme usually consists of a number of files with names ending in .cpp; each of these files is called a *module*, and is compiled separately. If some variable, say *name*, is to be accessed in several different modules, then it must be declared globally in all the modules where it is accessed. But one and only one of the modules should actually make space for it. Which one? To answer this question, the program should mark all the declarations which are **not** to make space as *extern*, like this:

```
extern char name[40];
```

In one and only one module should appear the simple declaration

```
char name[40];
```

That is the module where the space is made. When the compiled modules are “linked” to form one whole executable program, the references to the variable in all the modules where it was external are made to point to the space allocated by the one module where it was not external.

It could potentially be quite a nuisance to remember to mark all but one declaration as extern. That is where the word “GLOBAL” comes in handy. It is not a standard C keyword but is used in Interdyme, G and many other programs. The declarations of all global variables are put into “header” files like USER.H, and they are all marked GLOBAL. These header files are then “included” into all the modules where they are relevant by a *compiler directive* like

```
#include “user.h”
```

In all but one module, this directive is preceded by another:

```
#define GLOBAL extern
```

In these modules, the compiler will replace “GLOBAL” by “extern” before compiling. In that one and only one other module, namely dyme.cpp in Interdyme, the “include” is preceded by

```
#define GLOBAL
```

which defines GLOBAL to be nothing, so that in this module the "extern" is omitted and space is made for the global variables.

When a vector or matrix is declared **locally**, it can be fully "constructed", that is, space allocated for its elements. But there is a problem with declaring an object like a vector **globally**; since the declaration is outside of any function, no computing can be done. But to find out how much space to allocate for the array of numbers in the vector, the VAM file has to be read, but reading the VAM file is computing, so what is to be done? The answer is that once computing has begun, we must *resize* all the global matrices and vectors. That is, we must read the VAM file, find out how big the matrix or vector is, grab enough memory to hold it, and stick the pointer to that memory into the space saved for the pointer by the global declaration.

That may seem complicated to understand, but it is easy to do. Look at MODEL.CPP for TINY with the G or other text editor. The lines concerned with resizing are the following:

```
 // Resize 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"); lab.r("lab"); cap.r("cap"); ind.r("ind");
depc.r("depc"); labc.r("labc"); capc.r("capc"); indc.r("indc");
pcec.r("pcec"); invc.r("invc"); govc.r("govc"); exc.r("exc");
imc.r("imc"); x.r("x"); y.r("y");

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

The "resize" function or method is abbreviated to just .r . The argument to the resize function is the name of the vector or matrix in the VAM file. It may occur to you, as it has to others, that preparing these lines for USER.H and MODEL.CPP from the VAM.CFG is a lot of rather mechanical, error-prone work. If so, you will be pleased to learn that way back when you gave G the command

```
vamcreate vam.cfg hist
```

it also wrote two files, VAM.GLB and VAM.RSZ, as follows

The VAM.GLB file:

```
GLOBAL Vector out, pce, gov, inv, ex, im, fd,
   dep, lab, cap, ind, depc, labc, capc,
   indc, pcec, invc, govc, exc, imc, x,
   y;
GLOBAL Matrix FM, AM, LINV;
GLOBAL Matrix OUTlag;
```

The VAM.RSZ file

```
out.r("out"); pce.r("pce"); gov.r("gov");
inv.r("inv"); ex.r("ex"); im.r("im"); fd.r("fd");
dep.r("dep"); lab.r("lab"); cap.r("cap"); ind.r("ind");
depc.r("depc"); labc.r("labc"); capc.r("capc"); indc.r("indc");
pcec.r("pcec"); invc.r("invc"); govc.r("govc"); exc.r("exc");
imc.r("imc"); x.r("x"); y.r("y");
FM.r("FM"); AM.r("AM"); LINV.r("LINV");
OUTlag.r("out");
```

They are in the \tiny\model directory. You will notice a striking similarity to the corresponding portions of the USER.H and MODEL.CPP files; really all you have to do is bring these two files written by G into USER.H and MODEL.CPP , respectively, with the G or other text editor.   For the model, I removed the FM and LINV matrices, for they will not be used in the Interdyme model. Likewise, I removed *OUTlag* matrix, which is used to store the lagged values of the *out*
vector, because the lagged values are not yet in use.

The rest of the *loop()* function should be regarded as standard which the user of Interdyme should have no need to change.   The *spin()* function, which we have looked at in detail, is where the changes have to be made for different models.

Exercise

3. Build a TINY- like model for your imaginary economy or for a real economy for which you have data readily available.

8. Matrix Tools in Interdyme

Here is a quick overview of the actions, operators, and functions available in Interdyme. Some of them we have seen, but others were not needed in the TINY example. You need not learn the exact syntax of each of them; just make a mental note of the possibilities.

If A is a Matrix or Vector and k is a scalar (a float in C terms), then

k*A	multiplies each element of A by k.
A/k	divides each element of A by k.

If A and B are both Matrices or both Vectors of the same dimension, then

A + B	gives the matrix or vector sum
A - B	gives the matrix or vector difference
ebemul(A,B)	gives the element-by-element product
ebediv(A,B)	gives the element-by-element quotient, the elements of A being divided by the corresponding elements of B. If, an element of B is zero, the corresponding element of A is returned in that position.

If A has the same number of columns as B has rows, then

A*B	gives the matrix product.

If A and B have the same number of columns, then

A/B	gives the same thing as ~A*B, that is, the transpose of A multiplied by B, but without actually forming the transpose of A. (Think of the / as being a ' to denote transposing the matrix.)

Interdyme understands parentheses; if all the dimensions are appropriate, the following is an acceptable statement:

```
A =k* (B+C+D)*(E + F + G*(H+I));
```

If x and y are both Vectors with the same number of elements:

dot(x,y)	gives the inner product as a float.

If A is a Matrix and x a Vector with the same number of elements as A has columns, A%x gives the result of converting x to a diagonal matrix and then post-multiplying A by this diagonal matrix. Said perhaps more simply, it multiplies each column of A by the corresponding element of x.

If A is a Matrix and x a Vector with the same number of elements as A has rows,

x%A gives the result of converting x to a diagonal matrix and then pre-multiplying A by this diagonal matrix. Said perhaps more simply, it multiplies each row of A by the corresponding element of x.

The first of these % operations is useful in computing a flow matrix from a coefficient matrix and a vector of outputs. The second can compute a flow matrix in current prices from one in constant prices.

For a Matrix A, Vector v, float z, and int k,

v.set(z)	sets all elements of Vector v to z.
A.set(z)	sets all elements of Matrix A to z.
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.
z = v.sum()	puts the sum of the elements of v into z.
v.First()	gives the number of the first row of v if v is column and vice versa.

If A is a square, non-singular matrix,

!A	gives the inverse of A.
A.invert(i,j)	transforms A into its inverse by Gauss-Jordan pivoting. The pivot operations start in row i and stop when the pivot has been in row j. If these arguments are omitted, the pivoting starts in the first row and continues through the last, to produce the true inverse.

The difference here is that !A does not change A but creates a new matrix for the inverse while A.invert() transforms A into its inverse. Thus, if memory space is scarce, the invert action may be preferable. The algorithm in both cases is Gauss-Jordan pivoting with no niceties. Don't trust it if your matrix poses any problems for inversion.

If A a is either a Vector or Matrix object, then

~A	gives the transpose of A.
A.rows()	gives the number of rows as an integer.
A.columns()	gives the number of columns as an integer.
A.firstrow()	gives the number of the first row as an integer.
A.lastrow()	gives the number of the last row as an integer.
A.firstcolumn()	gives the number of the first column as an integer.
A.lastcolumn()	gives the number of the last column as an integer.

If A is a square Matrix and q and f are Vectors of the appropriate dimension, the equation

q = Aq +f

can be solved by the Seidel iterative method (if it converges) by the function

Seidel(A, q, f, triang, toler)

where triang is an array of integers giving the order in which the rows of A should be selected in the Seidel process, and toler is a float giving the tolerance which is accepted in the iterative solution. Similarly, the equation

p = pA + v

can be solved by PSeidel(A, p, v, triang, toler);

If you need a "scratch" Matrix A or Vector B is not in the VAM file, you can declare it locally in the function where it is needed by:

```
Matrix A(n,m);
Vector B(n);
```

were *n* is the number of rows and *m* is the number of columns.

In the process of debugging a program, it is sometimes useful to display a Matrix or Vector A on the screen. To do so, use

```
A.Display("message", fieldwidth, decimals);
```

To write a Matrix A to a file use

```
writemat(A, filename,fieldwidth, decimals);
```

To write a Vector A to a file use

```
writevec(A, filename, fieldwidth, decimals);
```

For completeness, we mention a function which will be explained in following sections. A Matrix A can be balanced to have the row sums given by Vector a and column sums given by Vector b by the function

```
int ras(A, a, b)
```

If the sum of the elements of a and b are not equal, the user is required to pick which governs.

Finally, I should mention that all of these matrix routines are available in a matrix package call BUMP (Beginner's Understandable Matrix Package) which can be used in C++ independently of the rest of Interdyme. The code of BUMP is carefully explained so that beginners with C++ can learn from it how to write such functions. Understanding how it works, however, is not necessary for using the functions in Interdyme any more than it is necessary to know how Excel is programmed in order to use it.

9. Vector Elements in Regression Equations

So far, we have used only one behavioral, regression-estimated equation, the one for investment. In this section, we will add regression equations for all components of Personal consumption expenditures. In the process, we will introduce several new techniques. So far, we have used only macrovariables, that is, those in the standard G bank, in regressions; now we will see how to use elements of vectors in regression. To ensure that the total of the predicted values stands in a reasonable relation to disposable income, we will need to learn about static vectors and apply some vector arithmetic.

In the Tiny\Model directory, you will find the file PCE.DAT shown below.

The PCE.DAT File

```
vmatdat r 1 9   1 8  5
pce 1995 1996 1997 1998 1999 2000 2001 2002 2003
#"  Date" "   pce1$" "   pce2$" "   pce3$" "   pce4$" "   pce5$" "   pce6$" "   pce7$" pce8$
1995    14.169   1.908   76.759   283.944   288.323   109.289   443.041   0.0
1996    14.101   1.917   78.365   299.580   295.352   115.091   446.638   0.0
1997    14.127   1.934   76.941   317.095   304.522   121.077   454.966   0.0
1998    14.228   1.972   77.331   343.489   317.501   123.079   466.458   0.0
1999    14.519   2.016   77.277   375.369   334.601   126.768   478.526   0.0
2000    15.000   2.000   80.000   400.000   350.000   130.000   500.000   0.0
2001    14.947   2.001   77.527   407.622   353.149   127.153   506.602   0.0
2002    14.821   1.985   77.343   419.457   356.482   121.248   508.665   0.0
2003    14.974   1.928   75.216   435.677   364.071   115.282   507.851   0.0
```

Open G, assign the HIST.VAM file as bank b, and make it the default vam file. Then introduce the data in the PCE.DAT with an "`add pce.dat`"statement, and check that it has been correctly read with a *show* command.

From this data, we now want to estimate simple consumption equations by regressing each component of the Personal consumption expenditure vector, *pce,* on personal disposable income (*pdisinc*) and its first differene (*dpdis*). You will also find in the \tiny\model directory the PCE.REG file shown, in abbreviated form, on the right. (The . . . . show where there are similar triplets of commands for regressions for sectors 2, 3, 4, and 5 in the full file on the disk.) You can now execute this command file in G either by "add pce.reg" or by opening the file in the editor and clicking "Run". You will see that G has no problem figuring out that *pce1, pce2, ..., pce7* are elements of the *pce* vector in the default VAM bank.

The PCE.REG File

```
lim 1995 2003
catch pce.cat
save  pce.sav
f dpdis = pdisinc - pdisinc[1]
ti PCE on Agriculture
r pce1 = pdisinc, dpdis
gr *

. . . .

ti PCE on Transport
r pce6 = pdisinc, dpdis
gr *
ti PCE on Services
r pce7= pdisinc, dpdis
gr *
save off
catch off
```

IdBuild, however, is not so clever. It knows nothing about the VAM bank. In response to the command "iadd pce.sav", it will therefore give a number of error messages such as "Cannot find pce1." The solution, however, is simple. We just need to tell IdBuild that *pce* is a vector. We do so with the command

```
isvector pce
```

in the MASTER file. Here is the complete MASTER file for the TINY model with the PCE equations. The new material is in bold type.

Master File for TINY with PCE Equations

```
isvector pce
iadd pce.sav
isvector clear
iadd invtot.sav
iadd account.sav
iadd pseudo.sav
end
```

(In the printed box, the first three lines — isvector pce, iadd pce.sav, isvector clear — are in bold.)

From it, IdBuild will produce a HEART.CPP file with the section shown to the right elow for the PCE equations. Note first that *pce* Vector must be passed to the function *pcef*; Secondly, note that the left side of equation stores the value computed by the equation directly into the appropriate element of the *pce* vector. There is no provision here for the automatic application of fixes or rho adjustments. How fixes or adjustments may be applied we will see in a later section. Finally, back in the Master file above, note the "`isvector clear`" command in the third line. If it were not there, IdBuild would write the following *invtotf* and *accountf* functions so that they also had to be passed – quite unnecessarily – the *pce* vector.

Excerpt from HEART.CPP File

```
void pcef(Vector& pce)
{
 dpdis[t]= pdisinc[t]- pdisinc[t-1];
/*  PCE on Agriculture */
 pce[1] = coef[0][0]+coef[0][1]*pdisinc[t]+coef[0][2]*dpdis[t];
/*  PCE on Mining */
 pce[2] = coef[1][0]+coef[1][1]*pdisinc[t]+coef[1][2]*dpdis[t];
/*  PCE on Gas and Electricity */
 pce[3] = coef[2][0]+coef[2][1]*pdisinc[t]+coef[2][2]*dpdis[t];
/*  PCE on Manufacturing */
 pce[4] = +coef[3][0]+coef[3][1]*pdisinc[t]+coef[3][2]*dpdis[t];
/*  PCE on Commerce */
 pce[5] = +coef[4][0]+coef[4][1]*pdisinc[t]+coef[4][2]*dpdis[t];
/*  PCE on Transport */
 pce[6] = coef[5][0]+coef[5][1]*pdisinc[t]+coef[5][2]*dpdis[t];
/*  PCE on Services */
 pce[7] = coef[6][0]+coef[6][1]*pdisinc[t]+coef[6][2]*dpdis[t];
}
```

Equations for Personal consumption expenditures (PCE) need a property not required of the equations for most other variables, namely, they must add up properly. More precisely, sum of the predicted values from the PCE equations plus Personal savings must equal Personal disposable income. We could, of course, just let savings be a residual, but it is too important for the macroeconomic properties of the model to be treated so casually. So we generally want to have an equation for personal savings – and for any other items in the difference between total PCE and disposable income, of which there are none in TINY.) The best way to achieve this equality is to add up the predicted pieces, compare the sum with the desired total, and spread the difference over the components in some pre-determined shares.   The shares I hare used are proportional to the income coefficients of the various regression equations. The shares were chosen in this way because the discrepancy between the sum of the predicted values and the desired total can be thought of as a little more or little less income to be divided among the various goods purchased. Sometimes we have used the standard error of estimate of the different equations on the grounds that the changes should be greatest where the uncertainty about the right value is greatest.

The mechanics of how the discrepancy is allocated is, however, independent of how the shares were determined. We create a text file, which we shall call PCESPREAD.DAT, which has the shares we want to use, however we got them. They should, of course, sum to 1.0. The box to the right shows this file for TINY.

The PCESpread.dat file

```
0.003175
0.000236
0.000000
0.481880
0.248311
0.036523
0.229874
0.000000
```

In the file USER.H, we need to add at the end the line

```
GLOBAL Vector PCESpread;
```

to declare globally the Vector which will hold the shares for spreading.

To make use of the new equations, we need to make a few changes in MODEL.CPP. On the disk, the modified file is called MODEL2.CPP; copy it to MODEL.CPP to make the changes take effect. The box on the next page shows excerpts from the new MODEL.CPP with the changed areas in bold and, as usual, the locations of lines which have been cut out to help you concentrate on the essentials indicated by four dots (. . . .).

The first order of new business in the loop() function is to resize the *PCESpread* vector which has been declared globally and then to read in its data. That job is accomplished by the lines

```
// Resize and read the vector for spreading the PCE discrepancy.
PCESpread.resize(NSEC);
PCESpread.ReadA("PCESpread.dat");
```

Note that the resizing is done by the “resize” function of the Vector, not by the “r” function. The “r” function looks in the VAM file to find the dimension of the Vector, so it won’t work for the *PCESpread* Vector, because it is not in the VAM file. The “resize” function is given the size directly as its argument. In TINY, *NSEC* has already been given the value of 8.

The predicted values of the elements of the *pce* vector are computed from the regression equations by the function call

```
pcef(pce);
```

The lines

```
 // Sum up the calculated PCE elements
pcesum = pce.sum();
pcediscrep = pcetot[t] - pcesum;
// printf("\npcediscrep = %10.2f\n",pcediscrep);
// Spread discrepancy by the proportions of PCESpread vector.
pce = pce + pcediscrep*PCESpread;
```

sum up the calculated PCE elements, subtract the sum from the desired total, and spread the discrepancy among the sectors in the proportions given by the PCESpread vector. The line

```
// printf("\npcediscrep = %10.2f\n",pcediscrep);
```

is now just a comment without effect on the operation of the program. But if the // at the beginning were removed, it would print the discrepancy at each pass through the loop. I used it to satisfy myself that the discrepancies were fairly small. Instead, however, of totally removing it, I left it as a comment which may be a useful illustration of how to include such debugging printing.

You should now estimate the PCE equations, make the required changes in USER.H and MODEL.CPP, run IdBuild, then do Model | Run dyme and graph the resulting elements of the *pce* vector.

To review, the two new techniques introduced in this section were the *isvector* command to IdBuild and the use of a static vector not in the VAM file.

The *isvector* command, by the way, also allows vector elements to be used on the **right-hand side** of regression equations. For example, in a more detailed model than TINY, the output of the Railroad industry may be used in the equation for Railroad construction.

Excerpts from the MODEL2.CPP File

```
. . . .
#include "user.h"     // All global variables for user model.
. . . .
const int NSEC=8; // Number of sectors in I-O table, most vectors.
void loop(void)
{
   . . . .
   triang.ReadA("triang.iv");   // Read the triangularization vector

   // Resize and read the vector for spreading the PCE discrepancy.
   PCESpread.resize(NSEC);
   PCESpread.ReadA("PCESpread.dat");

   if(MaxFlag == 'n')
          spin();
  . . . .
  }

void spin(){
   . . . .
   float oldpcetot,oldinvtot,pcedif,invdif;
   float pcesum, pcediscrep;
   for (t = godate; t<= stopdate; t++) {
          . . . .
      while(Iteration < 20){
             Iteration++;
             oldinvtot = invtot[t]; oldpcetot = pcetot[t];
             if(t>= MacEqStartDate)
             // pce = pcetot[t]*pcec; Removed
             // Compute with the estimated PCE equations
             pcef(pce);
             // Sum up the calculated PCE elements
             pcesum = pce.sum();
             pcediscrep = pcetot[t] - pcesum;
             // printf("\npcediscrep = %10.2f\n",pcediscrep);
             // Spread the discrepancy by the proportions of the PCESpread vector.
             pce = pce + pcediscrep*PCESpread;
             invtotf();
            . . . .
            }
    . . . .
    }
```

10. Systems of Detached-Coefficient Equations

The use of the *isvector* command to IdBuild is perfectly satisfactory for elements of vectors on the right-hand side of the regression equation; but for variables on the left-hand side, it has its limits. In the first place, automatic rho-adjustment is not possible. Secondly, it cannot be used to pass the whole system of import equations to the Seidel routine so that imports dependent on domestic demand can be calculated simultaneously with product outputs. Nor can it be used to pass a whole system of consumption functions, such as the PADS system to be explained in a later chapter, to a function to compute predicted values as a fairly complicated function of the parameters. All of the problems are overcome in through the use of a system of “detached-coefficient” equations. Actually, this method is older than the *isvector* method and goes back to models developed in the early 1960's. The “detached-coefficient” name comes from the fact that the regression equations are “detached” from the variables names and stored in a separate file by G. The added complexity of this method, however, is that program has to be written to interpret the equations. For Seidel with import equations and for PADS, however, the code is already written and part of the Interdyme system.

In this section, we will see how to use the detached-coefficient method for the PCE equations we have already estimated. We begin from the PCEEQN.REG file shown in the box to the right. It is a command file for G; as usual, the four dots indicate the omission for printing here of a repetitve portion of the text of the file. There are two new commands, *punch* and *ipch*. The *punch* command

punch pce.eqn 7 4 2003

opens a file to be called PCE.EQN to receive the regression coefficients. There will be 7 equations with up to 4 coefficients each. The last year of data used in estimating them will be 2003. After each equation is estimated there is an *ipch* command such as

```
ipch pce 1 L
```

which writes to the open “punch” file the rho and the regression coefficients for the most recently estimated regression equation. In the file, it will be labeled as the equation for the *pce* vector, element 1. The “L” will be written to file to indicate the type of equation. It could be any one letter or number; we will write the program to interpret these letters correctly. The L stands for “Linear”. The file ends with the command

punch off

to close the file to which G has been writing the regression results. The name of the command, *punch*, refers to punching the results into cards and indicates just how ancient this method is. The cards are gone, but the name works fine.

The PCEEQN.REG File

```
catch pce.cat
lim 1995 2003
vam hist b
dvam b
f dpdis = pdisinc - pdisinc[1]

punch pce.eqn 7 4 2003

ti PCE on Agriculture
r pce1 = pdisinc, dpdis
ipch pce 1 L
gr *

ti PCE on Services
r pce7 = pdisinc, dpdis
ipch pce  7 L
gr *

punch off

catch off
```

The PCE.EQN File

```
 7 4 2003
pce 1 L 3
 1 2 3
     0.280107     10.1751   0.00312443  -0.00190926
pce 2 L 3
 1 2 3
     0.456294     1.62189  0.000232365  0.000253049
pce 3 L 3
 1 2 3
     0.150868     76.9213 -6.45766e-06   -0.0127951
pce 4 L 3
 1 2 3
     0.214085    -303.312     0.474066    -0.155993
pce 5 L 3
 1 2 3
     0.268659    -14.8633     0.244285   -0.0810024
pce 6 L 3
 1 2 3
     0.572774     66.8459    0.0359306    0.0773437
pce 7 L 3
 1 2 3
   -0.0435048     164.664     0.226147    -0.179012
```

The PCE.EQN file produced by G from the PCEEQN.REG file shown above is shown in the box on the left. The three numbers on the first line are taken from the *punch* command that opened the file and have already been explained. For each equation there are then three lines. The first gives, somewhat redundantly, the name of the vector of the dependent variable, the number of the element in the vector, the type of equation, and the number of regression coefficients for it. The second line specifies which coefficients will be given in the next line, and the third line gives first the rho estimated for the equation and then the regression coefficients in the order specified by the second line.

In our case, the second line may seem unnecessary since it is both very simple and always the same. It can, however, add considerable flexibility. We might have had, for example, a more general form of regression in which there were five possible independent variables with a relative price term as the fourth and a time trend as the fifth variable. In that case, had the equation for element 7 used only the intercept and the time trend, the command to G would have been

```
r pce7 = time
ipch pce 7 L 1 5
```

and in PCE.EQN the resulting lines would have been something like

```
pce 7 L 2
 1 5
   -0.0435048      164.664     12.345
```

where 12.345 is the coefficient on *time*. Since the coefficient matrix is originally set to zero, this device allows one type of equation to be used for all variants of an equation that differ from a base type only by omitting one or more of the variables. Note that the numbers at the end of the “ipch” command are optional. If they are not specified, then G assumes that they are the integers up to the number of variables in the equation.

Now with the equations estimated and stored away in the PCE.EQN file, we have to get them into the Interdyme program and tell Interdyme how to interpret the coefficients. To do so, we will use the C++ concept of a “class”, namely, of a class called “Equation.”. Generally, a class is a combination of data and ways of working with the data. We have already met the Matrix and Vector classes. Instances of a class are called “objects.” For example, in USER.H we need to put the line

```
GLOBAL Equation pceeqn;
```

It will make the object *pceeqn* an instance of the class Equation. Then in MODEL.CPP, we put the lines

```
// Resize, read, and store the PCE equations; reads the pce.eqn file.
pceeqn.r("pce.eqn");
```

The second of these calls the *r* function (sometimes called *method*) of the *pceeqn* object to read the file PCE.EQN, claim enough space for storing the data of the PCE equations, and read the data from the file into that space. How does the computer know how to claim the space and read the data? Basically, the same way it knows how to invert a matrix object, namely, the creators of the Interdyme system wrote code that tells it how to read the data for *any* object that is an instance of the *Equation* class. That code is in the file EQUATION.CPP if you really want to look at it, but you can use the object very well without reading it. Any object of the Equation class knows how to read the data, how to dish out the coefficients of the equations and some other data to using programs, and how to apply rho adjustments to the predicted values. It does NOT know how to use the coefficients to compute the predicted values. The user, you, must write those instructions. We will see how in a moment. First, here is an except from MODEL.CPP which shows the new code for reading the equation data in context in the loop() function.

```
// Resize and read the vecor for spreading the PCE discrepancy.
PCESpread.resize(NSEC);
PCESpread.ReadA("PCESpread.dat");

// Resize, read, and store the PCE equations; reads the pce.eqn file.
pceeqn.r("pce.eqn");

// pceeqn.rhostart is read as the last year used in estimating the PCE equations.
// We want it to be the year in which to start the rhoadjustment.  If we are
// doing a historical simulation, that should MacEqStartDate if it is before the
// date recorded in the pce.eqn file.
if(pceeqn.rhostart > MacEqStartDate) pceeqn.rhostart = MacEqStartDate;
```

In the *spin()* function, the new equations are called to calculate the *pce* vector by the line

```
pcefunc();
```

shown in context in the lines below.

```
// Compute the estimated PCE equations with equation system
if(t>= MacEqStartDate){
      pcefunc();
      // Sum up the calculated PCE elements
      pcesum = pce.sum();
      pcediscrep = pcetot[t] - pcesum;
      // printf("\npcediscrep = %10.2f\n",pcediscrep);
      // Spread the discrepancy by the proportions of the PCESpread vector.
      pce = pce + pcediscrep*PCESpread;

      invtotf();
      }
inv = invtot[t]*invc;
....
```

As already noted, the function *pcefunc()* to calculate the predicted values must be written by the user. Consequently, it is in MODEL.CPP, and we had better have a good look at it. It is shown in full in the box below.

The first thing to be explained is the difference between the equation number, for which the program uses the variable *i* and the sector number, for which it uses the variable *j* . In the case of TINY, the PCE vector has 8 elements, one for each sector, but there are only 7 equations, because one sector (sector 8) is always zero. In larger models or for other dependent variables, the difference between the number of sectors and the number of equations can be much greater. The equations are read and stored in the order estimated, which is not necessarily the order of the sector numbers. For each equation, however, the corresponding sector number is stored in the *sec* variable Thus,

j = pceeqn.sec(i);

will put into *j* the sector number corresponding to equation *i* . The single character indicating the type of equation is stored in the *type* variable. Thus which = pceeqn.type(i);

will put into the **char** variable *which* the type of equation *i* . In our case so far, it will be the letter L . Finally, the regression coefficients of equation *i* are given by pceeqn[i][1], pceeqn[i][2], pceeqn[i][3], and so on. In our case, these are the constant term, the coefficient of *pdisinc* and *dpdis*, respectively. With this explanation, the code down as far as the comment

// Apply rho adjustment

should be reasonably clear. The working of the rho adjustment requires more explanation.

Program to Calculate PCE from Detached-Coefficient Equations

```
// pcefunc() -- PCE functions for TINY
int pcefunc(){
       int n, i,j,t1;
       float cons;
       char which;

       n = pceeqn.neq; // Number of equations
       // pdisinc is personal disposable income. It is a global macro variable.
       // Compute variables used in several equations.
       dpdis[t] = pdisinc[t] - pdisinc[t-1];

       // Loop over the equations
       for(i = 1; i <= n; i++){
              j = pceeqn.sec(i); // j is the sector number of this equation.
              which = pceeqn.type(i);
              if(which == 'L'){
                     cons = pceeqn[i][1] + pceeqn[i][2]*pdisinc[t] +
                     pceeqn[i][3]*dpdis[t];
                     }
              else{
                     printf("Unknown equation type %c in pcefunc, category %d.\n",
                            which,i);
                     tap(); //Pause so that error message can be read.
                     continue;
                     }
              // Apply rho adjustment                 }
              // Note the use of i and j in the following statement.
           pce[j] = pceeqn.rhoadj(cons,pce[j],i);
         }
       pce.fix(t);
       return(n);
       }
```

.In macro models, the calculation of the initial error from which the additive rho-adjustment factor begins is simple enough. It is calculated in the first period of the run. Matters are more complicated in the multisectoral model case because not all data is equally up-to-date. We may, for example, have macroeconomic variables through 2005 but detailed PCE data only through 2003. The starting error for rho adjustments must therefore be calculated for PCE components in 2003 and used in 2004, 2005, and later years. For a macrovariable, however, actual data can be used for 2004 and 2005, and the starting rho-adjusment error calculated in 2005 and used thereafter. That is why the variable *rhostart* is part of and Equation object. As read from the data file, it gives the last year of estimation, which is presumably the last year of data and therefore the last year in which the base error of the rho-adjustment process can be calculated. A further complication is that, since predicted values of one variable may depend upon calculated values of other variables, the rho-adjustment errors cannot be set until the model has converged for the period. Finally, sometimes we want to do a historical simulation and set all the rho-adjustment errors in the first period of the run. In that case, we uncheck the “Use all data” box on the screen which appears when we click Model | Run dyme.

Bearing all that in mind, what happens when the program executes the statement

```
pce[j] = pceeqn.rhoadj(cons,pce[j],i);
```

near the end of the program shown in the box? The *rhoadj* function of the pceeqn object is passed the calculated value (*cons*), the value already in the vector (*pce[j])*, and the equation number, (*i*). Simply put, the function then figures out what it is supposed to do and does it. More specifically, if we are using all data and data is still available, it returns the actual value. If we are past the end of the data, it adds on the rho-adjusted error to the value predicted by the equation and returns that. If the signal that the model has converged (*setrho*) has been set and we are in the period in which the base error should be computed, it does so, saves it, and returns the actual value. If the model has converged, but we are already past the year for calculating the error, it adds on the error for this year to the computed value and updates the error for the next year by multiplying this year’s error by the appropriate rho factor.

That leaves one question – well, at least one – How does the *rhoadj* function know that the model has converged? Why does it need to know that? Because only then can it calculate the starting error for rho adjustment in the appropriate year, or in later years, multiply this year’s errors by the appropriate rho to get them ready to be added to the values predicted by the equations in the next year. The signal that the model has converged is the variable *setrho*. It really should be called *SetRhoAdjusmentErrors*, but that is too long for lazy programmers (like me) who abbreviated it to just *setrho.* This *setrho* can have one of two values, ‘y’ for yes or ‘n’ for no. Near the top of the MODEL.CPP file, we see the lines

```
 float pcesum, pcediscrep;
setrho = 'n';
int Iteration;
```

so we know it starts off as ‘n’. What happens next is shown in code snippet from MODEL.CPP in the box below.   The **while** loop begun in the first line continues until convergence is reached (or the limit on iterations is hit). On reaching convergence, the program breaks out of the **while** loop, sets *setrho* to ‘y’, calls the two functions involving rho adjustment – namely *pcefunc()* and *invtot()* – and then puts *setrho* back to ‘n’.

Handling of *rhostart* in MODEL.CPP

```
while(Iteration < 20){
    Iteration++;
    oldinvtot = invtot[t]; oldpcetot = pcetot[t];

    . . . .

    invdif = fabs(invtot[t] - oldinvtot);
    pcedif = fabs(pcetot[t] - oldpcetot);
    printf("Iter %2d pce = %7.1f pcedif = %6.2f invdif = %6.2f\n",
         Iteration, pcetot[t],pcedif,invdif);
    // Break out of while loop if convergence has been reached
    if(invdif < .5 && pcedif < .5) break;
    }
// Set error for rhoadjustment
setrho = 'y';
if(t >= pceeqn.rhostart)  pcefunc();
invtotf();
setrho = 'n';
```

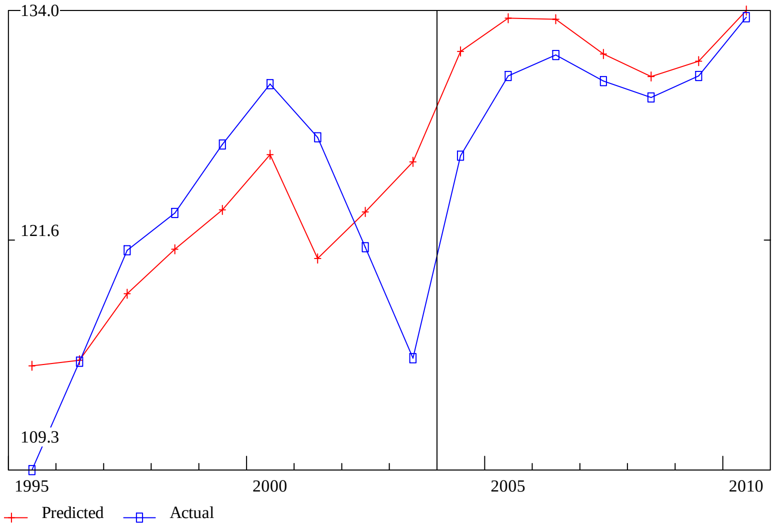
As with macromodels, there other kinds of fixes besides rho adjustment. How to specify them will be explained in section 12. Here we just note that the statement

pce.fix(t);

at the end of the *pcefunc()* function will apply the fixes to be explained in that section.

The fact that the rho-adjustment is working is seen very clearly in the graph below. For it, we made the VAM file BASE.VAM the default VAM file, copied the variable *pdisinc* from the BASE bank into the G workspace, and then estimated the regression in test mode, the default mode. Thus, the "actual" line in the graph is history up to 2003 and model forecast thereafter, while the "predicted" line is the prediction from the equation *without* rho adjustment but made with the model-predicted values of disposable income. The way the model forecast remembers the big error of the equation in 2003 indicates clearly that the rho adjustment is working.

PCE on Transport



11. Import Equations

So far we have considered imports as exogenous. In fact, of course, they are very dependent on domestic demand, so it makes sense to relate them to domestic demand and then calculate them from the estimated equations *simultaneously* with the calculation of output. In this section, we will see how to do exactly that. The new code is all in MODEL4.CPP, which you should copy to MODEL.CPP.

The IMPR.DAT file, shown to the right, will, when introduced in G with the *add* command, provide values for the *im* vector from 1995 to 2003. Up until now, imports have been treated as *negative* numbers because it is convenient to do so in the input-output table. In this file however, imports are *positive* numbers, as they usually are in statistical sources. We will use them henceforth as positive numbers and make the required changes in our program.

It is natural to suppose that imports of product i, m_i, are a linear function of domestic demand for the product, d_i, thus

$$m_i = \alpha_i + \beta_i d_i .$$

Conceptually, domestic demand for a product is the row total for that product in the input-output table *without subtracting imports.* Thus, conceptually, it does not depend on imports. Statistically, however, it is most easily calculated as domestic output, q_i, plus imports (as a positive number), thus:

$$d_i = q_i + m_i.$$

The impR.dat File

```
vmatdat r 1 9   1 8  5
im 1995 1996 1997 1998 1999 2000 2001 2002 2003
#Date     1       2          3      4          5 6     7         8
1995    19.753   8.118  0  108.065   0 0  14.470  0
1996    19.320   8.236  0  113.841   0 0  14.795  0
1997    18.808   8.612  0  124.300   0 0  15.561  0
1998    18.114   8.619  0  130.503   0 0  16.307  0
1999    18.732   9.068  0  145.711   0 0  17.461  0
2000    20.000  10.00   0  170.000   0 0  20.000  0
2001    19.068   9.302  0  155.832   0 0  18.902  0
2002    18.992   9.122  0  156.815   0 0  18.665  0
2003    20.105   9.763  0  166.286   0 0  19.018  0
```

Substitution of d_i from the second of these equations into the first and solving for m_i gives imports as a linear function of domestic output:

$$m_i = a_i + b_i q_i$$

To estimate these regressions, we need data on domestic output, *out*; it is found in the OUT.DAT file. (In real economies, it is quite normal to have data on output in years for which the complete input-output table is not available. You may, however, wonder how I made it up for TINY. In Section 13, we will introduce a series of input-output coefficient tables for 1995 – 2003. From the data already given for PCE and imports, and by moving the 2000 final demand vectors for investment, government, and export by the corresponding historical totals, final demand vectors for these years were calculated; with the time-varying input-output table, the implied domestic outputs shown in the OUT.DAT file were calculated.)

The OUT.DAT File

```
vmatdat c 1 9   1 8  5
out    1995    1996    1997    1998    1999    2000    2001    2002    2003
Ag    127.66  134.03  143.60  149.83  156.71  164.00  165.33  166.35  167.31
Mine   43.04   44.49   46.14   47.39   48.67   50.00   51.07   51.96   51.96
Elec  177.22  182.53  185.93  191.08  196.78  205.00  203.96  205.63  205.78
Mfg   613.61  641.88  680.28  718.33  758.87  787.00  795.93  810.36  832.24
Com   330.46  338.79  350.13  364.29  383.29  401.00  404.00  407.86  416.37
Trans 166.86  174.08  182.33  186.06  192.30  198.00  194.85  189.79  185.23
Serv  585.82  593.32  607.69  622.64  640.34  667.00  676.61  682.71  686.39
Gov   133.19  133.97  135.85  137.98  144.51  150.00  153.59  160.44  167.37
```

The G command file to do the regressions is shown on the left below, and the resulting IMPORT.EQN file is shown on the right. The program to compute the imports from these equations is given in the third box and may be found in model4.cpp.

The IMPORTS.REG File

```
# Estimate the import equations
bank tiny
vam hist b
dvam b
add impR.dat
add out.dat
vc out = LINV*fd

# Regress imports on outputs
lim 1995 2003 2003
punch import.eqn 4 2 2003
ti Agricultural Imports
r im1 = out1
gr *
ipch im 1 L
ti Mining Imports
r im2 = out2
gr *
ipch im 2 L
ti Manufacturing Imports
r im4 = out4
gr *
ipch im 4 L
ti Service Imports
r im7 = out7
gr *
ipch im 7 L
punch off
```

The IMPORT.EQN File

```
 4 2 2003
im 1 L 2
 1 2
     0.253335      18.6513   0.00365866
im 2 L 2
 1 2
    -0.104578     0.740326     0.170614
im 4 L 2
 1 2
   -0.0630504     -67.3531     0.282823
im 7 L 2
 1 2
     0.235647     -14.1671     0.049055
```

The Import Function from the MODEL.CPP File

```
// Importfunc() -- Import functions for TINY
int importfunc(Vector& q){
       int n, i,j,t1;
       float imp;
       char which;

    if(t < MacEqStartDate) return(0);
       n = impeqn.neq; // Number of equations
       // pdisinc is personal disposable income. It is a global macro variable.

       // Loop over the equations
       for(i = 1; i <= n; i++){
           j = impeqn.sec(i); // j is the sector number of this equation.
           which = impeqn.type(i);
           if(which == 'L'){
              imp = impeqn[i][1] + impeqn[i][2]*q[j];
              }
           else{
              printf("Unknown equation type %c in importfunc, sector %d.\n",
                           which,i);
              tap(); //Pause so that error message can be read.
              continue;
              }
           // Apply rhoadjustment                     }
           // Note the use of i and j in the following statement.
           im[j] = impeqn.rhoadj(imp,im[j],i);
           }
       im.fix(t);
       return(n);
       }
```

We now need a version of Seidel that will compute imports simultaneously with outputs.   It is shown in the box below and may be found at the end of the model4.cpp file. The prototype for it is found near the top of the MODEL4.CPP file:

```
// Prototype for Seidel used in TINY
short Seidel(Matrix& A, Vector& q, Vector& f, IVector& triang,float toler);
```

so that when a reference is made to it later, the C++ compiler will know how to set up the call to it and how to handle the return. In this same file, in the *spin()* function, right after the PCE and investment functions have be calculated, we find the lines:

```
inv = invtot[t]*invc;
gov = govtot[t]*govc;
ex  = extot[t]*exc;
fd = pce + gov + inv + ex;
 Seidel(AM, out, fd, dump, triang, toler);
 imtot[t] = im.sum();
```

The first three compute final demand vectors as we have done up to now. Then the *fd* vector is calculated as the sum of these components. Note that imports have **not** been subtracted.

In the Seidel routine, a potentially infinite loop is started by the code of which the outline is:

```
iter = 0;
while(1){
       . . . .
       iter++;
       if(dismax < toler || fdismax < .000001) break;
       if(iter > 25 )return(ERR);
       }
return(OK);
```

The key to getting out of this loop successfully is clearly to get the *dismax* variable down to less than *toler,* where *toler* is the error tolerance for the Seiderl process which was given in the call to *Seidel* and *dismax* is the maximum discrepancy between the output of a sector calculated on the current iteration of the Seidel process and the output of the same sector calculated in the previous iteration. The exit from the *while* loop when the iteration count passes 25 is just a safety net to avoid a hung computer if the process fails to converge.

The first thing done under the *while* loop is to compute imports, *im*, with a call to the import function. These imports are then subtracted from the given *fd* vector to get the net final demand vector, *f*, with which the rest of this iteration of the Seidel process takes as its final demand. Note that the calculation of the import vector may involve not only the import equations but also rho-adjustments and perhaps other fixes. Convergence on the *q* or output vector will imply convergence on the import vector as well.

The most of the rest of the code of Seidel process just implements the process as explained previously and should be fairly transparent. There is, however, one non-standard wrinkle which needs explanation. It gives the process the capability of accepting a pre-determined output for one or more products. Petroleum extraction in the United States is a good example; demand beyond some capacity level must be imported or dealt with in some way as a shortage. If *i* is a normal, non-constrained product, then in column 18 of the SECTORS.TTL file its row should have the letter e . The *e* means that the product uses the *e*quation to determine output. Here are a few lines from TINY’s SECTORS.TTL:

```
Agricul     ;1   e   "Agriculture"
Mining      ;2   e   "Mining and quarrying"
Elect       ;3   e   "Electricity and gas"
```

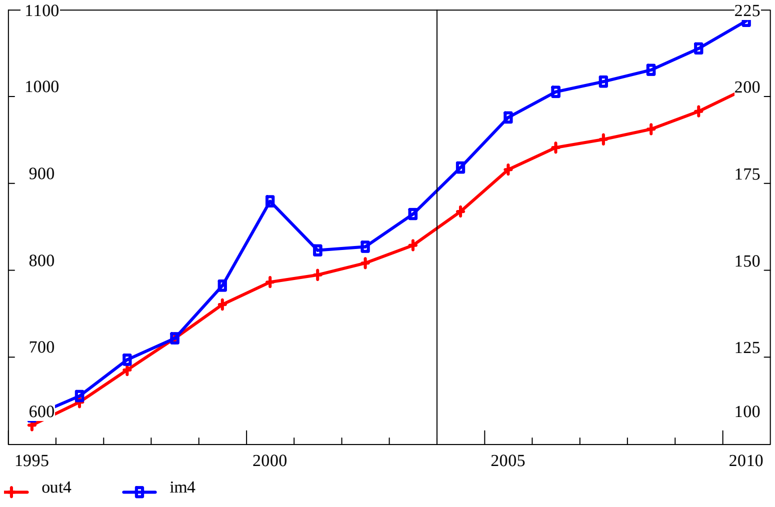
Whatever letter is in that column will become the value of *outfix(i)*. When our routine finds a vaule of 'e' in *outfix(i)*, it will routine set the output of product *i* equal to the demand which has been calculated. If, however, *i* is a constrained product, the routine keeps the initial, presumably exogenously set, value of *q[i]* and puts the difference between it and calculated demand into the vector called *dump*. What should be done with *dump* is then up to the model builder who is using this feature. Since TINY is not using it, we will do nothing with dump.

At this point you may want to refresh your memory of the Seidel process, but after doing so, all the coding in the Seidel routine shown below should make sense. When it does, build and run the new new model and make graphs of imports and outputs. (You will need to copy MODEL4.CPP to MODEL.CPP .)

```
/* Seidel() Version 2, with imports .*****************************************


short Seidel(Matrix& A, Vector& q, Vector& fd, Vector& dump, IVector& triang,
    float toler){
    short i,j,k,first,last,n,iter,im1,imax;
    float discrep,dismax,fdismax;
    double sum;
    Vector f(fd.rows()); //scratch vector

    iter = 0; // Iteration count
    n = A.rows();
    first = A.firstcolumn();
    last =  A.lastcolumn();

    if(q.rows() < A.rows()){
        printf("In Seidel, the solution vector is not large enough.\n");
        return(ERR);
        }
    dump.set(0.);

    while(1){
       // Compute imports
       importfunc(q);
       f = fd - im;
        dismax = 0;
        for(k = triang.First(); k < triang.First()+ triang.numelm(); k++){
            i = triang[k];
            sum = f[i];
            for(j = first; j <= last; j++)
                sum += A[i][j]*q[j];
            sum -= A[i][i]*q[i]; // Take off the diagonal element of sum
            sum = sum/(1.- A[i][i]);
            if(outfix[i] != 'e'){ // Override the equation
                  // Dump error into dump
                dump[i] = q[i] - sum;
                sum = q[i];
                }
           else dump[i] = 0;
            discrep = fabs(sum - q[i]);
            if(discrep > dismax){
                dismax = discrep;
                imax = i;
                if(fabs(sum)>1.0e-3) {
                    fdismax = fabs(sum);
                    fdismax = dismax/fdismax;
                    }
                }
            q[i] = sum;
            arith("in Seidel",i);
            }
        iter++;
        if(dismax < toler || fdismax < .000001) break;
        if(iter > 25 ){
            printf("No convergence in %d iterations. Discrep = %7.2f in sector %d.\n",
                iter, dismax, imax);
            return(ERR);
            }
        }
    printf("Seidel iterations: %d ",iter);
    return(OK);
    }
```

Manufacturing

Output and Imports



The above graph shows the similarity that we now find in the forecasted course of outputs and imports for Manufacturing. The scale for the two curves, however, is different; the scale for output is shown on the left and that of imports on the right. The imports – the blue, upper line marked with squares – are actually much less than output, but the pattern of movement is similar. (The right vertical axis for graphs drawn with the G *mgr* command may be set by the *hrange* command.)

12. Speeding Up Solutions with Read and Write Flags

So far, we have been reading and writing the value of every vector and every matrix from the VAM file in every period. That is not necessarily a smart thing to do. Take the *AM* vector for example. We know perfectly well that at present we do nothing to change it in the program, so writing it back to the VAM file at the end of each year's calculation is a waste of time – not much time in the case of TINY, but in larger models it could take a second or so. In the case of the *out* vector, we certainly want to write the value after it has been computed, but in the forecast period the value of *out* in the VAM file on the first run of the model will be all zero. That is not a good starting value for the Seidel process. It would be much better just to start with the previous period's value. Thus, we would want to read it only in the initial year.

Such considerations have lead to the introduction of read and write flags for the vectors and matrices which are set when they are resized in MODEL.CPP. For example, so far we have had as the first line of the resizing commands

```
out.r("out");pce.r("pce");gov.r("gov");
```

which is equivalent to

```
out.r("out",'y','y');pce.r("pce",'y','y');gov.r("gov",'y','y');
```

The second argument – the first 'y' – in one of these function calls is the read flag; the third, the write flag. Possible values for the read flag are:

‘y’ yes, always read the vector or matrix
‘n’ no, never read the vector or matrix
‘i’ read the vector or matrix in the initial year of the run
‘a’ read the vector or matrix if there is data available for it.

Possible values of the write flag are:

‘y’ yes, always write the vector or matrix
‘n’ no, never write the vector or matrix

The default value of both flags is ‘y’; that is to say, if no flags are specified, both are assumed to be ‘y’. If only one is specified, it is interpreted as the read flag. To specify a write flag, we must also specify the read flag, even if it is ‘y’. For example, to set the read flag for the *AM* matrix to ‘y’ and the write flag to ‘n’, we would use the call

```
AM.r(“am”, ’y’,’n’);
```

To set the read flag for the *out* vector to ‘i’ but the write flag to ‘y’ , it is enough to use the call

```
out.r(“out”, ‘i’);
```

The read flag ‘a’ can be quite convenient to read the values of a vector in years when it is known but not in later years. Its use, however, requires telling the program the last year of data for the vector or matrix in question. To do so, the first step is to remove the comment indicator (//) in front of the line

```
// lastdata(); // Use of lastdata function requires that the LastData file exist.
```

which is found just above the resize commands for vectors. The call to the function *lastdata()* will read a file which must be called LASTDATA. Here is a possible LASTDATA file for TINY:

```
pce 2003
im 2003
out 2000
```

It simply lists one or more of the vectors in the VAM file and the last year for which data is available for each.

Because TINY is so tiny, you will not notice any acceleration of the solution by the use of these flags. For big models, however, they can make a small but measurable difference in solution time. You should experiment with them in TINY, but since there is not much in the way of results to show, I will not give a more explicit application.

13. Changing input-output coefficients and prices

Up until now, our input-output coefficient matrix has remained constant, as have the coefficients for the value-added components depreciation, labor income, capital income, and indirect taxes. Consequently, there has been no need to consider changes in relative prices. We must not leave TINY without introducing coefficient change and relative prices, for input-output analysis has often been unjustly reproached for assuming constant coefficients. In fact, it is precisely input-output analysis which allows us to specify changing input-output coefficients.

I prevailed upon the TINY statistical bureau to construct a coefficient matrix for 1995 and drew upon the collective wisdom of a group industry specialists to project the table to 2010. The results of these labors, together with the matrix for 2000 with which we have been working, are contained in the files AM1995.DAT, AM2000.DAT, and AM2010.DAT. For intervening years, we will have to rely upon linear interpolation. The same groups provided us with past history and projections of the labor income coefficients for the same two years. They are in the file LABC.DAT.

The TINYP.PRE File

```
# File to create VAM file for TINYP, TINY with Prices
vamcreate vamp.cfg hist
vam hist b
dvam b
# Bring in data on outputs and final demands
add out.dat
add pceR.dat
add impR.dat
# Bring in the intermediate flow matrix
add flows.dat
# Bring in the final demand vectors
add fd.dat
# Bring in the value added vectors
add va.dat
fdates 2000 2000
# The following are element-by-element vector divisions
vc depc = dep/out
vc capc = cap/out
vc indc = ind/out
# Compute share vectors of final demands
vc invc = inv/invtot{2000}
vc govc = gov/govtot{2000}
vc exc  = ex/extot{2000}
# Read I-O coefficient matrices
add AM1995.dat
add AM2000.dat
add AM2010.dat
# Interpolate matices for missing years
fdates 1995 2010
lint AM
show AM r 4
# Keep value added coefficients, except labc, constant
dfreq 1
f one = 1.
index 2000 one depc
index 2000 one capc
index 2000 one indc
#Read labc for 1995, 2000, and 2010
add labc.dat
lint labc
# Keep the structure of some final demand columns constant
index 2000 one invc
index 2000 one govc
index 2000 one exc
# Compute final demand vectors for historical years
fdates 1995 2003
vc inv = invtot*invc
vc gov = govtot*govc
vc ex  = extot*exc
store
```

We need to expand the VAM file so much that it is perhaps better to start from a new VAM configuration file, VAMP.CFG, which you will find on the disk. (The P in the filename stands for Prices.) We have added the vector of prices, *prices*, final demands in nominal prices, *fdN, pceN, govN, invN, exN, and imN*, the interindustry flow matrix in nominal terms, *FMN*, and two vectors, *fix* and *cta*, whose functions will be explained in the next section on fixes.

Corresponding to this new configuration is a new G command file, TINYP.PRE, to create the initial VAM file for the model. In some ways it is simpler than before; we do not need to compute the AM matrix but rather just read it for 1995, 2000, and 2010. The one new command follows these reads. It is simply

```
lint AM
```

where *lint* stands for "linearly interpolate." It will linearly interpolate the values of each and every cell between the years where non-zero values are given. The same command is applied a few lines further down to the *labc* vector. The *lint* command works over the range specified by the last previous *fdates* command. When you have run this file through G, you will have created a new HIST.VAM file. Copy it to BASE.VAM with the following G command:

```
dos copy hist.* base.*
```

The MODEL.CPP file for the model with prices is on the disk as MODEL5.CPP; copy it to MODEL.CPP. Near the top of the file, resizing commands for the new vectors and matrix have been added. The new elements in the heart of the *spin()* function are shown in bold in the box below.

Excerpt from MODEL5.CPP File

```
for (t = godate; t<= stopdate; t++) {
   . . . .
    // Start of code particular to TINY:

    // Compute total value-added coefficient vector.
    vac = depc+labc+capc+indc;
    // Compute prices
    PSeidel(AM, prices, vac, triang, 0.000001);
    // The loop for convergence on pcetot and invtot.
    while(Iteration < 20){
        . . . .
      }
    // Set error for rhoadjustment
    setrho = 'y';
    if(t >= pceeqn.rhostart)  pcefunc();
    if(t >= impeqn.rhostart)  importfunc(out);
    setrho = 'n';
    // Here when both Investment and PCE have converged
     // Compute flow matrix in current prices
     FM = AM % out;
     FMN = prices % FM;
     // Put output and final demands into nominal prices
     outN = ebemul(prices,out);
     fdN = ebemul(prices,fd);
     pceN = ebemul(prices,pce);
     invN = ebemul(prices,inv);
     exN  = ebemul(prices,ex);
     imN  = ebemul(prices,im);
     govN = ebemul(prices,gov);
     // End of code particular to TINY
    . . . .
    }
```

The first addition is to compute the total value added coefficient vector and then compute the *prices* vector by a call to *PSeidel()*. This routine, which may be found in SEIDEL.CPP, solves the equations

$p = p\,A + v$

where p and v are row vectors. If the sectors of an input-output matrix have been arranged in a good, almost triangular order for solution of the output equations by the Seidel method, the *opposite* order will be good for solving for prices. Hence, *PSeidel()* uses the same *triang* vector as does *Seidel()* but takes the sectors starting from the end of the list and working towards the beginning. The remaining new commands are just to calculate the values of the nominal vectors and of the nominal flow matrix, *FMN*. This last calculation uses Interdyme's % operator. It is used between a vector and a matrix. If the vector is on the **right**, it multiplies each **column** of the matrix by the corresponding element of the vector. If the vector is on the **left**, it multiplies each **row** of the matrix by the corresponding element of the vector.

A final note on the economic interpretation of the prices we have calculated may be helpful. They are, in fact, relative prices and the *numeraire* with price equal to 1.00 is the quantity of labor which could be bought for one dollar in the year 2000.

To obtain truly nominal prices, we need to model inflation, and to model inflation, as we already know from the Quest model in Part 2, we need to model employment and unemployment. Further, we need to model savings behavior more carefully and consider the role of money and interest rates in both savings and investment. All these things can be modeled using the techniques we already have., so they make a better exercise for you than subject for me.

We can introduce optimizing in Interdyme models in exactly the same way as we did in Quest. All we need do is to specify an objective function and the regression coefficients to be varied in exactly the same way as in a macro model and then check the “Optimizing” radio button on the panel of window which comes up when we click Model | Run dyme. At present, optimization works only with respect to the coefficients of macro equations, not for the coefficients of systems of detached-coefficient equations. There are no problems of principle involved in extending the code to handle coefficients of those equations also, just writing some housekeeping code. Finally, although there is a “Stochastic simulation” radio button on that same panel, it does not at present (April 2006) work. Much of the necessary code has been brought over to Interdyme, but somewhat more work is needed to activate it.

It is tempting to go on elaborating the fictitious TINY economy with all these features, but you surely have a real model you are eager to get to work on, so we will give TINY no further features, for there are no more major new techniques that need to be explained. We do, however, need to deal with how to use fixes to discipline TINY or any model if it misbehaves.

Exercise

Make up employment and labor force data for TINY, and estimate and put into the model employment equations. Have the model compute unemployment. Modify the savings equation so that the model provides employment for roughly 96 percent of the labor force, though the exact level may vary cyclically.

14. Fixes in Interdyme

This section has been copied verbatim from the Interdyme manual of 2000 March 9 and has not yet been integrated into the context of this book.

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 which we have already seen 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 G 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 G bank is passed to MacFixer through the form which the above G 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 G 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 G 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
  1993  4.1
  1995  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 G 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 G 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 G 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 G 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 G 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 G main menu. From the information on the form which this command creates, G 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.

Answers

3.2 The new outputs are
166.14 55.21 222.30 763.57 426.48 206.41 812.58 148.00
and the primary resources required to produce them are
317.24 1351.84 268.34 226.58.

3.3 The net export of depreciation is -5.73.

3.4 The new price vector is (.92 .96 .89 .90 .71 .93 .96 1.00).

3.5 Net export of greenhouse gas production is -31.87.

15. A Historical Note

All of us tend to presume that the world was made the way we found it; if there were input-output tables in it when we arrived, then they must have always been there. Of course, that is not the case. In fact, they are so much connected with the work of one man, Wassily W. Leontief, that without his remarkable contribution they would probably not have been developed until decades later. Born in St. Petersburg in 1906, he was already a university student when the Bolsheviks began taking over the educational program. He joined a group protesting this process, was caught pasting up a poster, spent a while in jail and was periodically jailed and interrogated thereafter. Though deeply interested in the economy of his country and in the efforts at economic planning, he clearly had little to hope for from the Bolshevik government. Even as an undergraduate, however, his paper on "The Balance of the Economy of the USSR" describing efforts in Russia to investigate interindustry relations came to the attention of professors in Germany. When he graduated from the University of Leningrad in 1926, he was offered the possibility of graduate study in Germany, but it was already difficult to get out of the Soviet Union. By an extraordinary turn of fate, he developed a bone tumor on his jaw. It was removed, but the surgeon warned him that he would surely soon die. Armed with the surgeon's written statement, he argued to the officials that he should be allowed to leave the country since he would certainly be useless and possibly expensive to the government. The argument worked, and in 1925 he arrived in Germany with the tumor in a bottle. It was there re-examined and found ... benign! His work in Germany led, via Nanjing, to an appointment at the National Bureau of Economic Research in New York. His theoretical writings came to the attention of the Harvard faculty which offered him an instructorship. He accepted the Harvard offer on the condition that he be given a research assistant to help him build what we would now call an input-output table. The reply informed him that the entire faculty had discussed his request and had unanimously agreed that what he proposed to do was impossible and, furthermore, that even if it were done, it would be useless. Nonetheless, they were so eager to have him come that they would grant the request and hope that he would use the resources for better purposes. He didn't. In 1936, his first results were published; in 1939 a book *The Structure of the American Economy* appeared. It had input-output tables for the United States for 1919 and 1929. The theoretical parts of the book had the major ideas of input-output analysis: coefficients, simultaneous solution, and price equations. During World War II, Leontief constructed, with support of the U.S. Bureau of Labor Statistics (BLS), a 96-sector table for 1939 and, by 1944 was able to study changes in employment patterns which could be expected after the end of the war. In 1947, a second edition of the book appeared with the addition of a 1939 matrix and a comparison of input-output and single-equation projections.[2] In 1973, he was awarded the Nobel prize in economics for this work. Leontief remained active until shortly before his death in 1999 at the age of 93.

In 1949, a group at the BLS began work on a 400-sector table for 1947. A 190-sector table was published in 1952, but financing -- which had come through the Defense budget -- for the more than fifty people working on the project was discontinued early in the Eisenhower administration, so that neither the full table nor the extensive documentation of the details of its production were ever published.

In other countries, making of tables spread rapidly. They were incorporated in the United Nation's standard System of National Accounts prepared by Richard Stone. In 1950, the first international conference on input-output methods was sponsored by the United Nations; the eleventh (without U.N. support) was held in 1995.

---

[2] The spelling of Leontief's name in Latin letters was for German speakers; English speakers almost invariably mispronounce it, though he never corrected anyone. In *Wassily*, the *W* is pronounced V, the *a* is long as in "father," and the accent is on the *si* which is pronounced "see". In *Leontief* the accent is on *on* and the *ie* is pronounced like the ye in "yet". The final *f* is a soft v.

In the late 1950's, Soviet authors, eager to make input-output acceptable in their country, put together a table for the Soviet Union in 1924 and argued that all the essential ideas had originated in the Soviet Union. The difference, however, between what they could find in the literature of that period and Leontief's comprehensive treatment only heightens an appreciation of his contribution.

Gradually, it has come to be recognized that an input-output table is not only useful for economic analysis and forecasting but is also an essential step in making reliable national accounts. The statistical offices of most major industrial countries, therefore, prepare input-output tables, often on a regular basis. Annual tables for France, the Netherlands, Norway, and Japan are prepared as a part of annual national accounting. In the USA, a comprehensive table is made every five years in the years of economic censuses (years ending in 2 and 7) and is used in revising and "benchmarking" the national accounts.

In 1988, the International Input-Output Association was organized as a group of individuals interested in using input-output techniques. In 1989, it began publishing its own journal, *Economic Systems Research*.

The Interdyme modeling system, like the G program, was developed by the Inforum group in the Department of Economics at the University of Maryland. It has been used in developing and linking dynamic input-output models of about twenty countries. Most of these models have been developed and used mainly in the country concerned.

Chapter 15. Matrix Balancing and Updating - the RAS Method

1. The RAS Algorithm

Making an input-output matrix from scratch for a country is a major undetaking often involving a group of ten or more persons for a number of years. By the time the project is finished, the matrix refers to a year that is apt to seem part of ancient history. Hence the question arises, Given an input-output table for a base year, is there a way to update it to a more recent year with less work than making the table from scratch? In this updating, one usually has some data for the more recent year. One wants the matrix for this year, which we may call the target year, to conform to all those data.

Usually those data include at least industry outputs, major GDP components, and value-added by industry. The value-added by each industry can then be subtracted from its output to give the total intermediate inputs by each industry. Thus, we would know the row total for each industry and the column total for each final demand column and for the intermediate use of each industry. An obvious check on the accuracy of this information is that the sum of the row totals equals the sum of the column totals. We will assume that this condition has been met, although meeting it is not always easy except by a rough scaling. Thus, we have the margins or frame for the table for the target year.

An initial guess of the inside of the table for the target year can then be made by assuming constant coefficients for the input-output coefficients and for the shares in each of the final demand vectors. More sophisticated initial estimates could also be made. One could use, for example, consumption functions to "forecast" the purchases of households. However the initial inside elements of the table are estimated, it is almost certain that they will not have the right row and column sums. Adjusting them to make them conform to these control totals is generally done by what has come to be called the RAS procedure, a name derived from notation in Richard Stone's description of the method in *A Computable Model of Economic Growth* (Chapman and Hall, London, 1962). The idea had been mentioned by Leontief in the 1941 edition of *The Structure of the American Economy*, but the idea seemed to pass unnoticed until applied by Stone.

The method is extremely simple in practice. First scale all of the rows so that each has the correct total. Then scale all the columns so that each has the correct total. The row sums are then probably no longer correct, so scale them again, and then scale the columns again, and so on until the scaling factors have converged to 1.0. The matrix at that point has the desired row and column sums. If A^t denotes the flow matrix at stage t of the operation, R^t denotes the row scaling factors at step t arrayed as the diagonal elements of an otherwise zero matrix, and S^t denotes the column scaling factors similarly arrayed, then the flow matrix at the beginning of stage t+1 is

$$\boldsymbol{A}^{t+1} = \boldsymbol{R}^t \boldsymbol{A}^t \boldsymbol{S}^t. \tag{1}$$

The expression on the right gave rise to the name RAS, which should be pronounced as the three letters, though foreign speakers of English often turn it into one syllable, "ras".

2. Convergence of the Algorithm

The practice is simple, but will the process converge? To answer that question, we will need some notation. Let the original matrix be $\boldsymbol{A}$, whose elements we will denote by a_{ij}, let b be the positive vector of required row sums and c be the positive vector of required column sums. The first condition is that $\boldsymbol{A}$ be non-negative. The second is simply that there must exist at least one matrix with zeroes everywhere that ***A*** has zeroes and positive numbers everywhere that ***A*** has positive numbers which has row sums equal to *b* and column sums equal to c.

Notice that this second condition did *not* assume a solution of the form we are seeking, that is, derived from ***A*** by scaling the rows and columns. It does, however, have some important implications. The first is that the sum of the elements of *b* must be the same as the sum of the elements of *c*. A further implication is that, if it is possible rearrange the rows and columns of ***A*** so that an all-zero block appears, then the corresponding subtotals of *b* and *c* must be consistent with those blocks remaining zero while the other cells are positive. For example, if

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}$$

then we must also have $b_1 < c_1$. In practice, one insures that the first implied condition (the equality of the sum of row sums and column sums) is met before beginning the RAS calculations. If they fail to converge, then one looks for inconsistences along the lines of the second implication.

The proof of the convergence of the RAS procedure under these general conditions is requires a complicated notation. The essence of the proof, however, can be seen in the special case in which ***A*** is all positive, and we will limit ourselves to that case. (For the general case, see M. Bacharach, *Biproportional Matrices.* Cambridge University Press.)

We will start the process by scaling the rows, then the columns, and so on. In the first row scaling, we choose the first-round row-scaling factors by

$$r_i^{(1)} = b_i / \sum_j a_{ij}. \tag{2}$$

where the superscript on the *r* refers to the iteration number. Then we compute the first-round column scaling factors by

$$s_j^1 = c_i / \sum_i r_i^{(1)} a_{ij}. \tag{3}$$

Then we come back to compute the second-round row scaling factors,

$$\begin{aligned} r_i^{(2)} &= b_i / \sum_j r_i^{(1)} a_{ij} s_j^{(1)} \\ &= b_i / \sum_j \frac{b_i}{\sum_k a_{ik}} a_{ij} s_j^{(1)} \\ &= 1 / \sum_j \frac{a_{ij} s_j^{(1)}}{\sum_k a_{ik}} \end{aligned} \tag{4}$$

Thus, we can see that the second-round row factors are reciprocals of convex combinations of the first-round column factors, that is, they are reciprocals of a weighted average of those first-round column factors with positive weights which sum to 1. Thus,

$$\underset{i}{Max}\ r_i^{(2)} \leq 1/(\underset{j}{Min}\ s_j^{(1)}) \text{ and } \underset{i}{Min}\ r_i^{(2)} \geq 1/(\underset{j}{Max}\ s_j^{(1)}) \ . \tag{5}$$

By similar reasoning,

$$\underset{j}{Max}\ s_j^{(1)} \leq 1/(\underset{i}{Min}\ r_i^{(1)}) \text{ and } \underset{j}{Min}\ s_j^{(1)} \geq 1/(\underset{i}{Max}\ r_i^{(1)}) \ . \tag{6}$$

The inequalities in (6) imply

$$1/\underset{j}{Max}\ s_j^{(1)} \geq (\underset{i}{Min}\ r_i^{(1)}) \text{ and } 1/\underset{j}{Min}\ s_j^{(1)} \leq (\underset{i}{Max}\ r_i^{(1)}) \ . \tag{7}$$

Then combining the first inequality of (5) with the second of (7) and the second of (5) with the first of (7) gives

$$\underset{i}{Max}\ r_i^{(2)} \leq 1/(\underset{j}{Min}\ s_j^{(1)}) \leq \underset{i}{Max}\ r_i^{(1)} \text{ and } \underset{i}{Min}\ r_i^{(2)} \geq 1/(\underset{j}{Max}\ s_j^{(1)}) \geq \underset{i}{Min}\ r_i^{(1)}. \tag{8}$$

In other words, the biggest element of *r* diminishes from iteration to iteration while the smallest rises. Since ***A*** is all positive, all of the inequalities in (5) through (8) will be strict inequalities unless all the elements of ***r*** are equal or all the elements of ***s*** are equal. But if they are all equal, they must be all be equal to 1, for otherwise the scaling would increase or decrease the total of all elements in the matrix, contrary to the fact that, after the first row scaling, the sum of all elements remains equal to the common sum of the vectors ***r*** and ***s***. Since the sequences of $\boldsymbol{r}^{(k)}$ and $\boldsymbol{s}^{(k)}$ vectors both lie in closed, bounded sets, they have limit points. Can these limit points be other than the vectors that are all 1's? No, because at any such point, one more iteration of the process would bring a finite reduction of the maximum element (and a finite increase in the minimum element) of each vector. (This is where we use the all positive assumption to have strict inequalities in (8).) Thus, for points sufficiently close to these limit points, the next iteration must also bring lower maximal and higher minimal elements than those of the limit point, contrary to the limit point being a limit point. Therefore the unique limit of each sequence of vectors is a vector of ones.

Thus the convergence is proven for the case of all positive ***A***. The proof is similar for ***A*** with some 0 elements, but in this case, it may require several iterations to get a finite reduction in the maximal elements of ***r*** and ***s***.

In practice, the condition that the sum of ***b*** equals the sum of ***c*** is checked and assured before the iterative process begins. The initial ***r*** and ***s*** vectors should be reported by the program because they often indicate discrepancies between ***b*** and ***c*** vectors and the initial ***A*** matrix. Once the iterations start, the largest and smallest elements of the ***r*** and ***s*** vectors should be reported every five or ten iterations. It is common to observe “wars” between a row control and a column control when one element looms large in both its row and column but the control totals for the two are quite different. Such “wars” are symptomatic of a failure of the second assumption and an indication that the ***b*** and ***c*** vectors should be revised.

It should be noted that the RAS procedure works for rectangular matrices just as well as for square ones. It is also useful in making input-output tables and the bridge matrices used to convert investment by investor to investment by product bought or consumption by consumer categories to consumption by product categories used for productive categories.

3. Preliminary Adjustments Before RAS

It often happens in updating or making tables that one has better information about some cells than about others. For example, in updating the a table with a Glass row, we may have quite good information on the sales of glass products to Beer, because we have information on the production of glass beer bottles. In this case, we can simply remove the "relatively well-known flow" from both its row and column control, perform the RAS balancing on the remaining flows, and then put back in the known flow.

The problem with this procedure is that the "relatively well-known flows" tend to be big flows. If they are not quite consistent with the row or column controls, then removing them requires that all of this inconsistency should be attributed to changes in the remaining small flows. Thus, the small flows can be pushed about rather considerably. This problem can be reduced by a preliminary scaling of the relatively well-know flows before removing them from the process. To describe this adjustment, let R_i be the sum (in the base year) of the relatively well-known flows in row i ; S_i , the sum of the other flows; and B_i , the row control. Then let

$$\alpha_i = \frac{R_i}{R_i + S_i} \tag{9}$$

and define z_i as the solution of

$$S_i z_i + R_i z_i^{\alpha_i} = B_i . \tag{10}$$

The value of z which satisfies (10) is readily found by Newton's method. We then scale all the relatively well-known flows by $z_i^{\alpha_i}$ and all the other flow by z_i . By (10), the row will then have the correct sum. By (9), $0 \le \alpha_i \le 1$. If $0 < \alpha_i < 1$ then $z_i^{\alpha_i}$ is closer to1.0 than is z_i; that is to say, the relatively well-known flows are scaled *less* than the other flows. They are however, scaled somewhat. If they account for a small fraction of the total of all flows in the row, they will be scaled but little; if they account for much of the row, they will be scaled almost as much as the other flows.

After this preliminary scaling, the known flows can be removed for the rest of the RAS process. While this scaling may seem a bit arbitrary, in practice it has given plausible results in many applications. In fact, it worked so well that the first person working with it, Thomas Reimbold, felt that the z must stand for *Zauber*, "magic" in his native language, and the procedure is therefore often referred to as the *Zauber* process.

Chapter 16. Trade and Transportation Margins and Indirect Taxes

1. Trade and Transportation Margins

A perennial problem in applied input-output analysis is the treatment of trade and transportation margins and of indirect taxes. The problem is nicely illustrated with transportation costs. If output is valued at the producer's price — the price at the factory gate, so to speake — then the cost of transporting the goods to the user must be considered to be paid separately by the purchasing industry. Thus, the cost of the rail services used in hauling the coal used by electric power plants shows up as an input of rail transportation into electric generation. The cost of hauling generation equipment to and from the utilities' repair facilities would appear in the same cell. Similarly, the cost of hauling coal to a steel mill and of hauling iron ore to the same mill will appear in the same cell.

The problems with this treatment are (1) it puts quite diverse activities into the same cell and (2) the table does not reflect the way the rail industry thinks about its business. It thinks in terms of products hauled — and prepares statistics on products hauled, not on industries to which it delivers. (Despite these problems, this treatment is the one most commonly followed.)

All of the problems apply with equal force to all the other transportation margins and to wholesale and retail trade margins.

One alternative is to change the measure of output of the industry to include the cost of delivering the product to the user. One disadvantage of this treatment is that it removes the numbers in the input-output table one step further from the numbers in terms of which people in the industry think, namely in producer prices. Another problem is that transportation margins may be very different for a dollar's worth of product delivered to different users. The transportation cost of oil delivered to an electric utility by pipeline from a marine terminal may be very different from delivering by truck or rail to a small industrial user.

A better alternative is to add another dimension to the input-output tables. Thus, corresponding to each cell of the tables we have considered so far there would be a vector. The first entry in the vector would be the transaction in producer prices; the second entry would show the rail margin; the third, the truck margin; the fourth, the air freight; and so on through the wholesale and retail trade margins. In effect, we would have a table with layers, the first layer for the producer price transaction, the second for the rail margins, and so on. In fact, the benchmark tables for the United States are prepared with all this information. It has not been commonly used because the size of the matrices involved has been, until fairly recently, large relative to the power of the computers available. That constraint has now been effectively removed, and we may ask, How would we in fact compute with such a layered table?

If $\boldsymbol{A}$ represents the coefficient matrix in producer prices and $\boldsymbol{T_i}$ represents the ith layer of transportation and trade margin coefficients, then the fundamental input-output equations become

$$q = Aq + \sum_i S_i T_i q = (A + \sum_i S_i T_i) q = f$$

where $\boldsymbol{S_i}$ is a matrix with 1's in the row which produces the service distributed by layer i and elsewhere all zero. The matrix $(A + \sum_i S_i T_i)$ is, in fact, the matrix in producer prices with which it has been traditional to compute. What is gained by distinguishing the layers is not a correction of the traditional computations but rather a better description of what the flows are and a better basis for studying changes in coefficients in the $\boldsymbol{T_i}$ matrices.

2. Indirect Taxes, Especially Value Added Taxes

Indirect taxes such as property taxes or franchise taxes are always and without problems treated as a component of value added, along with depreciation, profits, interest, and labor compensation. Excise taxes such as those on gasoline, alcohol, and tobacco are usually similarly treated, but with less justification, because some uses of these products are exempt. For example, gasoline used to power agricultural machinery or exported whiskey or cigarettes are exempt. Thus, these taxes should also be treated as a layer of the table, since they are not uniform for all cells. Retail sales taxes are usually treated as a component of value added by Retail trade. This treatment assumes that the tax is proportional to the retail margin in all products in all cells. In fact, there are different tax rates on different products, and some products are sold by retail establishments for intermediate use without retail sales tax.

The greatest problems, however, have probably been created by the value added tax (VAT) in the tables of countries which use this tax, a group that now includes all members of the European Union and numerous other countries. Producers pay VAT on the value of their sales but may deduct the VAT paid on their purchases. VAT is not charged on certain products, such as health services. Nor is it charged on exports. Many European input-output tables have been published in producer prices plus non-deductible VAT. That practice meant that the cell for paper products sold to the hotel industry did not contain VAT, because the VAT on those sales was deductible from the VAT owed by the hotels. The cell for paper products sold to hospitals, however, contained VAT, because the hospitals owed no VAT from which the VAT on the paper products could be deducted. Similarly, since households owe no VAT, they cannot deduct the VAT on the paper products they buy, so the VAT is included in the cell showing the sales of paper products to households. Thus, the cells in the paper products row of such a matrix have very diverse levels of VAT content. That means that the valuation of the product across the row is not homogeneous. It takes more wood pulp to make a dollar’s worth paper towels used by a hotel than to make a dollar’s worth of paper towels used by a hospital or household, because a significant portion of their dollar goes to VAT. This heterogeneity in the pricing in the row is obviously detrimental to the accuracy of the input-output calculations. The solution to the VAT problem is simply to create a VAT layer of the table.

Chapter 17. Making Product-to-Product Tables

1. The Problem

Makers of input-output tables often find data on inputs not by the *product* into which they went but by the *industry* that used them. An *industry* is a collection of establishments with a common principal product. But besides this principal product, any one of these establishments may produce a number of secondary products, products primary to other industries. Establishments classified in the Cheese industry may also produce ice cream, fluid milk, or even plastic moldings. Consequently, the Cheese industry may have inputs of chocolate, strawberries, sugar, plastic resins, and other ingredients that would appal a connoisseur of cheese. The inputs, however, are designated by what the product was, not by what industry made them. Similarly, data on the final demands, such as exports and personal consumption expenditure, is by product exported or consumed, not by the industry which made it. Thus, input-output matrices usually appear in two parts. The first part, called the Use matrix, has products in its rows but industries in its columns. The entries show the use of each product (in the rows) by each industry (in the columns.) The second, called the Make matrix, has industries in the rows and products in the columns; the entries show how much of each product was made in each industry.

How can we use these two matrices to compute the outputs of the various products necessary to meet a final demand given in product terms?

One way is to consider that each product will be produced in the various industries in the same proportion as in the base year of the table. This assumption is used, for example, in computable general equilibrium models based on social accounting matrices that explicitly show the Make and Use matrices. This assumption, however, can produce anomalous — not to say silly — results. In the above example, an increase in the demand for cheese would automatically and immediately increase demand for chocolate, strawberries, and sugar. That is nonsense. There must be a better way to handle the problem.

This highly unsatisfactory situation has led to efforts to make a product-to-product matrix. Indeed, the problem is so well recognized that the “Transmission programme of data” of the European system of accounts requires that all national statistical offices of the member states of the European Union transmit “symmetric” input output tables to Eurostat every five years. No real advice, however, is offered by Eurostat to the statistical offices on how to make these product-to-product tables. This paper offers a valuable tool for the process. (“Symmetric” is here intended to mean that the same concepts are used in both rows and columns. Its use as applied to these *matrices* is both highly confusing and not descriptive. Since it is the *nature* of the rows and columns that is the same, not their measure, *symphysic* would be both a better characterization and less confusing.)

To make such a matrix, we need to employ an additional assumption. There are basically two alternatives:

9. The product-technology assumption, which supposes that a given product is made with the same inputs no matter which industry it is made in.
10. The industry-technology assumption, which supposes that all products made within an industry are made with the same mix of inputs.

The *System of National Accounts 1993* (SNA) reviews the two assumptions and finds (Section 15.146, p. 367) "On theoretical grounds, ... the industry technology assumption performs rather poorly" and is "highly implausible." (Section 15.146, p 367) “From the same theoretical point of view, the product (commodity) technology model seems to meet the most desirable properties .... It also appeals to common sense as it is found *a priori* more plausible than the industry technology assumption. While the product technology assumption thus is favoured from a theoretical and common sense viewpoint, it may need some kind of adjustment in practice. The automatic application of this method has often shown results that are unacceptable, insofar as the input-output coefficients appear as extremely improbable or even impossible. There are numerous examples of the method leading to negative coefficients which are clearly nonsensical from an economic point of view." (Section 15.147)

Since 1967, the Inforum group has used a "semi-automatic" method of making "some kind of adjustment" in calculations based on the product-technology assumption, as called for by the SNA. We have used it with satisfactory results -- and without a single negative coefficient -- on every American table since 1958. The method was published in Almon 1970 and in Almon *et al.* 1974. Despite this long and satisfactory use of the method, it seems not to have come to the attention of the general input-output community. In particular, the authors of the section quoted from the SNA seem to have been unaware of it. The purpose of this note is to record the method where it is more likely to come to the attention of anyone working in input-output. At the same time, it expands the previous exposition with an example, provides a computer program in the C++ language for executing the method, and presents some of the experience of applying the method to the 1992 table for the USA.

2. An Example

An example will help us to visualize the problem. The Table 1 below shows the Use matrix for a 5-sector economy with a strong concentration in dairy products, especially cheese and ice cream.

Table 1. The Use Matrix

USE	Industries				
Products	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	0	0	0	0	0
Ice cream	0	0	0	0	0
Chocolate	4	36	0	0	0
Rennet	14	6	0	0	0
Other	28	72	30	5	0

We will call this matrix U. The use of chocolate in makings cheese and rennet in making ice cream alerts us to the fact that the columns are industries, not products. (Rennet is a substance used to make milk curdle. It is commonly used in making cheese but never in ice cream.) The Make matrix, shown in Table 2 below, confirms that cheese is being made in the ice cream industry and ice cream in the cheese industry.

Table 2. The Make Matrix

MAKE	Products				
Industries	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	70	20	0	0	0
Ice cream	30	180	0	0	0
Chocolate	0	0	100	0	0
Rennet	0	0	0	20	0
Other	0	0	0	0	535
Total	100	200	100	20	535

This matrix shows that of the total output of 100 of cheese, 70 was made in the Cheese industry and 30 in the Ice cream industry, while of the total ice cream output of 200, 180 was in the Ice cream industry and 20 in the Cheese industry. It also shows that, of the total output of 90 by the cheese industry, 78 percent (70/90 = .77778) was cheese and 12 percent ice cream. We will need the matrix, M, derived from the Make matrix by dividing each cell by the column total. For our example, the M matrix is shown in Table 3.

Table 3. The M Matrix

M	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	0.7	0.1	0.0	0.0	0.0
Ice cream	0.3	0.9	0.0	0.0	0.0
Chocolate	0.0	0.0	1.0	0.0	0.0
Rennet	0.0	0.0	0.0	1.0	0.0
Other	0.0	0.0	0.0	0.0	1.0

Now let us suppose that, in fact, cheese is made by the same recipe wherever it is made and ice cream likewise. That is, we will make the "product-technology assumption." If it is true and the matrices made well, then there exists a "recipe" matrix, R, in which the first column shows the inputs into cheese regardless of where it is made, the second column shows the inputs into ice cream regardless of where it is made, and so on. Now the first column of U, U_1, must be $.70*R_1 + .10*R_2$,where R_1 and R_2 are the first and second columns of R, respectively. Why? Because the Cheese plants make 70 percent of the cheese and ten percent of the ice cream. In general,

$$U = RM' \tag{1}$$

where M' is the transpose of M. It is then a simple matter to compute R as

$$R = U(M')^{-1}.$$

For our example, $(M')^{-1}$ is given in Table 4.

Table 4. M′ Inverse

1.5	-0.5	0.0	0.0	0.0
-0.2	1.2	0.0	0.0	0.0
0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.0	1.0

and R works out to be

Table 5. The R or "Recipe" Matrix

R	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	0	0	0	0	0
Ice cream	0	0	0	0	0
Chocolate	0	40	0	0	0
Rennet	20	0	0	0	0
Other	30	70	30	5	0

This R is very neat. All the rennet goes into cheese and all the chocolate goes into ice cream. Unfortunately, as indicated by the quotation from the SNA, it is rare for the results to turn out so nicely.

Indeed, just a slight change in the U matrix will show us what generally happens. Suppose that the U matrix had been just slightly different, with 1 unit less of chocolate going into cheese as shown below and one less unit of rennet used in ice cream.

Table 6. An Alternative Use Matrix

Alternative U	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	0	0	0	0	0
Ice cream	0	0	0	0	0
Chocolate	3	37	0	0	0
Rennet	15	5	0	0	0
Other	28	72	30	5	0

Table 7 shows what the R matrix would have been:

Table 7. An Impossible R Matrix

Impossible R	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	0.0	0.0	0.0	0.0	0.0
Ice cream	0.0	0.0	0.0	0.0	0.0
Chocolate	-1.7	41.7	0.0	0.0	0.0
Rennet	21.7	-1.7	0.0	0.0	0.0
Other	30.0	70.0	30.0	5.0	0.0

Here we find the infamous small negative flows. It is not hard to see how they arise. While it is conceivable that the Cheese industry does not produce chocolate ice cream, it is also very easy for the table makers to forget to put into the Cheese industry the chocolate necessary for the ice cream it produces, or to put in too little. Wherever that happens, negatives will show up in the R matrix.

The negatives have driven at least some statistical offices to the industry-technology assumption. The so-called commodity-to-commodity matrix, C, derived from this assumption is

$$C = UN', \tag{3}$$

where N is the matrix derived from the Make matrix by dividing each row by the row total. For example, the Cheese column of C is $C_1 = .77778U_1 + 0.14285U_2$ because 77.778 percent of the product of the first industry is cheese and 14.285 percent of the product of the second industry is cheese. The result of applying this assumption to our example is Table 8.

Table 8. The Mess Made by the Industry Technology Assumption

C Indust. Tech.	Cheese	Ice cream	Chocolate	Rennet	Other
Cheese	0	0	0	0	0
Ice cream	0	0	0	0	0
Chocolate	8.254	31.746	0	0	0
Rennet	11.746	8.254	0	0	0
Other	32.063	67.936	30	5	0

This "solution" has made matters worse. The original U matrix had 4 units of chocolate going into the Cheese industry, which admittedly made some ice cream. Now this industry-technology product-to-product matrix asserts that *8.25 units of chocolate went into producing pure cheese*! Not into the Cheese *industry* but into the *product* cheese! And 8.25 units of rennet went into producing curdled ice cream! To call the result a product-to-product table would be little short of scandalous.

Fortunately, we do not have to choose between this sort of massive nonsense and negative flows. It is perfectly easy to rely mainly on the product-technology assumption, yet avoid the negatives, as we will now show.

3. The No-Negatives Product-Technology Algorithm

We wrote the basic equation relating U, M, and R as equation (1) above. It will prove convenient to rewrite equation (1) as

$$U' = MR'. \tag{4}$$

Using U'_i to denote the i^{th} column of U' and R'_i to denote the i^{tb} column of R, we can write

$$U'_i = MR'_i\ . \tag{5}$$

Notice that this is an equation for the distribution of product i in row i of the Use matrix as a function of M and distribution of the same product in row i of the R matrix. We can simplify the notation by writing

$$\mathrm{u} = U'_i \quad \text{and} \quad r = R'_i\ ; \tag{6}$$

then the previous equation becomes

$$u = Mr \tag{7}$$

or

$$0 = -Mr + u \tag{8}$$

and adding r to both sides gives

$$r = (I - M)r + u\ . \tag{9}$$

Save in the unusual case in which less than half of the production of a product is in its primary industry, the column sums of the absolute values of the elements of (I - M) are less than 1, and the convergence of the Seidel iterative process for solving this equation is guaranteed a by well-known theorem. (If the share of the total production of a particular product coming from the industry to which it is primary is x, then the absolute value of the diagonal of (I - M) for that product is | 1 - x | and the sum of all the absolute values of off-diagonal elements in the column is |1 - x |, so the total for the column is 2*| 1 - x |, which is less than 1 if x > .5.) We start this process with

$$r^{(0)} = u \tag{10}$$

and then define successive approximations by

$$r^{(k+1)} = (I - M)\, r^{(k)} + u\ . \tag{11}$$

To see the economic interpretation of this equation, let us write out the equation for the use of a product, say chocolate, in producing product j, say cheese:

$$r_j^{(k+1)} = u_j - \sum_{\substack{h=1 \\ h \neq j}}^{n} m_{jh} r_h^{(k)} + (1 - m_{jj}) r_j^{(k)} \tag{12}$$

The first term on the right tells us to begin with the chocolate purchases by the establishments in the cheese industry. The second term directs us to remove the amounts of chocolate needed for making the secondary products of those establishments by using our present estimate of the technology used for making those products, $r^{(k)}$. Finally, the last term causes us to add back the chocolate used in making cheese in other industries. The amount of chocolate added by the third term is exactly equal to the amount stolen, via second terms, from other industries on account of their production of product j:

$$(1 - m_{jj}) r_j^{(k)} = \sum_{\substack{h=1 \\ h \neq j}}^{n} m_{hj} r_j^{(k)} \tag{13}$$

because

$$\sum_{h=1}^{n} m_{hj} = 1. \tag{14}$$

It is now clear how to keep the negative elements out of r. When the "removal" term, the second on the right of (12), is larger than the entry in the Use matrix from which it is being removed, we just scale down all components of the removal term to leave a zero balance. Then instead of adding back the "total-stolen-from-other-industries" term, $(1 - m_{jj})r_j$, all at once, we add it back bit-by-bit as it is captured. If a plundered industry, say Cheese, runs out of chocolate with only half of the total chocolate claims on it satisfied, we simply add only half of each plundering product's claim into that product's chocolate cell in the R matrix. We will call the situation where the plundered industry runs out of the product being removed before all claims are satisfied a "stop".

The process can also be applied to the rows of the value added part of the matrix. It is not certain, however, that the column sums of the resulting value-added table will match the value added as calculated from product output minus intermediate input. This value-added matrix will generally require RAS balancing to make it consistent with the product-to-product intermediate table.

4. When Is It Appropriate to Use This algorithm?

This algorithm is appropriate where the product-technology assumption itself is at least approximately true. Essentially, it allows there to have been slightly different technologies in industries where assuming *strictly* the average product technology would produce negatives. It is appropriate where the negatives arise because of inexactness in making the tables or because of slight differences in technologies in different industries. Applied to the Use matrix of either Table 1 or Table 6, this method gives the "neat" Recipe matrix of Table 5 with no rennet in ice cream and no chocolate in cheese. It never produces negative entries nor positive entries where Use has a zero. The row totals are unaffected by the process. It is, moreover, equivalent to deriving Recipe from equation (1) if no negatives would arise, so that if the product-technology assumption is strictly consistent with the Use and Make tables, the method produces the true matrix. It may even produce a correct Recipe matrix from a faulty Use matrix — as it has perhaps done in our example — so that equation (1) could be used to revise the estimate of the Use matrix.

Certain accounting practices, however, may produce situations which appear to be incompatible with the product-technology assumption, even though the underlying reality is quite compatible. For example, local electric utilities generally buy electricity and distribute it. In the U.S. tables, they are shown as buying electricity (not coal), adding a few intermediate inputs and labor, and producing only a secondary product, electricity, which is transferred, via the Make matrix, back to electricity. Looked at mechanically, this method of making electricity is radically different from that used in the Electricity industry, which uses coal, oil, and gas to make electricity, not electricity itself. If our algorithm is applied thoughtlessly to this situation, it cannot be expected to give very sensible results.

Fortunately, it is easy to generate signs of this sort of problem. One can compute the new Use matrix implied by equation (1) with the Recipe matrix found by the algorithm and the given Make matrix. This "NewUse" matrix can then be compared with the original Use matrix and the causes of the differences investigated. We will follow this procedure in next section on the experience of using the method on the 1992 tables for the USA.

To fix the problem in the above example about electricity, we have only to consider the output of the State and local utilities as production of their own primary product, which is then sold, via the Use matrix — not transferred via the Make matrix — to the Electricity industry. In essence, we use the industry technology assumption for the local electric utilities — and for all other industries where all of the output is secondary. The industry technology assumption may also be preferable for transfers to some catch-all sectors such as "Miscellaneous food preparations" (SIC2099), which includes such disparate products as vinegar, yeast, Chinese noodles, and peanut butter. It is probably just as reasonable to suppose that a product transferred into this industry is made with the average technology of the industry where it is made as with the average technology of this catchall sector. Indeed, this sort of industry can produce the reverse of the negatives problem. For example, because of the importance of peanut butter in this industry, it has significant inputs of oil seeds. Now the no-negatives algorithm will not pull oil seeds out of the "Macaroni, spaghetti, vermicelli, and noodles" industry, (SIC2098), (which used no oil seeds) just because it transferred some Chinese noodles to 2099. But neither will it take out an adequate amount of flour for those noodles, because flour is quite unimportant in the 2099 input mix. This problem shows up only indirectly by substantial oil-seed inputs to many food industries in the NewUse which transferred products to 2099 but, in fact, used no oil seeds. That is a signal to switch to the industry technology for these transfers by converting them to sales in the Use matrix.

Thus, in the use of this method, a number of iterations may be necessary. Changes in concepts, in treatments of some transactions, and occasionally in underlying data may be necessary. Although the calculation of the non-negative Recipe matrix is totally automatic, it may be necessary to make several runs to get acceptable results.

In this process, it must be recognized that a nice, clean accounting system may not be operational, that is, it may not provide by itself a simple, automatic way to go from final demand vectors specified by products to total outputs of those products. We may have to change slightly some of the concepts in the accounting system to make it operational. In making the change required for the Electricity example, we have messed up the neat accounting concept of the Electricity column of the Use matrix as a picture of what came into a particular group of establishments. We have, however, taken a step toward creating what might be called an operational Use matrix. I do not say, therefore, that statistical offices should not produce pure accounting Use matrices. But I do feel that they should also prepare the operational use matrix and the final product-to-product matrix, for in the process, they will learn about and deal with the problems which the users of the matrix will certainly encounter. They may even discover and correct errors in their work before they are discovered by their users.

This process is totally inappropriate for handling by-products such as hides produced in the meat packing industry or metal scrap produced in machinery industries. Their treatment is a different subject.

5. A Brief History of the Negatives Problem

The idea to compute R from equation (1) seems to have been first put in print by Van Rijckeghem (1967). He realized that there could be negatives but did not think they would be a serious problem. The idea of using equation (1) in this way, however, must have been in the air, for by early 1967, I had used it, without thinking that it was original, found negatives, and started work on the algorithm presented here.

The problem was encountered by ten Raa, Chakraborty and Small [1984] in the course of work which was primarily concerned with identifying by statistical means true by-products. They note the existence of the method presented here but write,

> [Almon] iterates truncated Neumann series in which matrix multiplications are carried out only to a limited extent to avoid negatives. This arithmetic manipulation goes without justification, is arbitrary and depends on the choice of [make matrix]-decomposition as well as the iteration scheme.

I do not believe that any of this comment is correct. The Neumann series is the expansion $(I - A)^{-1} = I + A + A^2 + A^3 + \dots$. The algorithm used here makes no use of this series; rather it uses the Seidel procedure. There are no matrix multiplications, nor is there is any equivalence between a “limited” number of terms in the Neuman series and the Seidel solution. The procedure is carried to convergence. We have seen that the procedure has a perfectly reasonable economic interpretation; indeed, it arose from the economic interpretation of the Seidel procedure. The only thing perhaps "arbitrary" is that 0 is considered a reasonable input flow while negatives are considered unreasonable. I do not know what the "[make matrix]-decomposition" refers to, but I can assure the reader that the solution does not depend on the "iteration scheme." While I could not see how it could, given that it is carried to convergence, I changed the program and ran the "robberies" in the opposite order. The answers were identical.

The ingenious attempt of ten Raa [1988] to modify elements of the matrices in such a way as to find a most probable U matrix consistent with a non-negative R should be mentioned even though it ended, in the author’s view, in frustration.

Rainer and Richter [1992] have documented a number of steps which they took towards making what I have called here the operational Use and Make matrices. Such steps should certainly be considered and applied if need. These authors still ended up with hundreds of negative flows in the R matrix because they were using just equation (1). At that point, the process described here could have been applied.

Steenge and Konijin [1992] point out that if the R matrix computed from equation (1) has any negatives in it, then it is possible to change the levels of output of the various industries in such a way that more of all products is produced *without* using more of all inputs. They feel that it is implausible that such a rearrangement is possible and observe that perhaps the negatives "should not be regarded as rejecting the commodity technology assumption, but as indicators of flaws in the make and use tables." (p. 130). I feel that there is much merit in that comment. It seems to me that the right time and place to use the algorithm presented here is in the process of making the tables. If there are not good statistical grounds for preferring the original Use matrix, the recomputed NewUse might well be argued -- following the reasoning of Steenge and Konijn -- to be a better estimate.

The caveat here is that there may well be cases where it really would be possible to increase the outputs of all products while using less of some product. For example, if there are shoes made in the Plastics products industry without any use of leather, while the Footwear industry uses leather, then by moving shoe production from Footwear to Plastic products it may be possible to produce more of all products while using less leather. Where such cases arise, a different solution is necessary, for example, moving the shoes made in the Plastics products industry together with their inputs into the Footware industry or insisting that the two kinds of shoes are separate if substitutable products.

6. Application to the U.S.A Tables for 1992

The method described here has been applied to all of the USA tables since 1958 with experiences broadly similar to those described here for the 1992 table. This table has 534 sectors, counting some construction sectors which have no intermediate sales. Of these 534, 425 have secondary production. Of the 283,156 possible cells in a 534 X 534 matrix, the Use matrix has 44,900 non-zero cells, and the Make matrix has 5,885. The matrix was produced in two versions. In one, certain activities, such as restaurant services of hotels, were removed from the industry where they were produced (Hotels) and put into the sector where these activities were primary (Restaurants). In the other, these activities were left in the industry where they were conducted. The first version was designed to make the product-technology assumption more valid, and it has been used here. The matrix also puts true by-products (such as hides from meat packing) in a separate row, not one of the 534 considered here.

To try to convey a feeling of what it is like to work with the algorithm, we will look at the process midway along, rather than at the very beginning or the somewhat polished end. That is, some adjustments in the Use and Make matrix from which the algorithm starts will have already been made. As a result of this application, further adjustments will be suggested before the next application.

Before this application of the algorithm, the output of industries which had only secondary production had been changed, for reasons explained above, to be primary and the flows moved from the Make to the Use matrix.

In the following rather detailed descriptions, necessary to give a picture of what the process is really like, I will, to avoid confusion, capitalize the first letter of the first word in industry names but not in product names.

The industry Water and sewer systems failed to satisfy the requirement that at least half of the output of a product should be in the industry where it is primary. Indeed, some 85 percent of this product's output comes from Other state and local enterprises, and the iterative procedure failed to converge for a few rows until this secondary transfer was converted into a primary sale. Production of secondary advertising services, which occurred in many sectors, was also converted to a primary product of the producing industry and "sold" via the Use matrix to the Advertising industry. Secondary production of recreational services in agricultural industries was similarly converted. Much of the output of the several knitting industries had been treated originally as secondary production, and these had been changed to primary sales before the calculations shown here. Finally, the diagonals of many columns of the Use matrix are large, in part because intra-firm services, such as those of the central offices, often appear there. Thus the same sort of service that is on the diagonal of industry i is also on the diagonal of industry j. In this case, the product-technology assumption does not apply, not because it is untrue, but because of the way the table was made. Until we are able to obtain tables without this problem, we have just removed half of the diagonals from the Use table before calculating Recipe, and have then put back this amount in both of these matrices and in the NewUse matrix.

The data in both Use and Make tables were given to the nearest 1 million dollars, and all dollar figures cited here are in millions. The convergence test in the iterative process was set at one tenth of that amount, .1 million dollars. The iterative process converged for most rows of the R matrix in less than five iterations. The most iterations required for any row was 15.

The resulting Recipe matrix looks very similar in most cells to the original Use table. The Recipe matrix contains, of course, only non-negative entries and can have strictly positive entries only where U has positive entries. It may, however, as a result of the "robbing" process, have a zero where U has a positive entry. In all, there were only 95 cells in which Recipe had a zero where Use had a positive entry.

Although it is the Recipe matrix that we need from this process, it is also interesting, as noted above, to compare the original Use matrix with what we may call NewUse, computed by the equation 1 by NewUse = Recipe*Make'. The difference between Use and NewUse shows the changes in the Use matrix necessary to make it strictly compatible with product-technology assumption, the given Make matrix, and the calculated Recipe matrix. If there was no "stop" in a row, the two matrices will be identical in that row. There were 118 such identical rows, 109 of them having no secondary output.

In the other rows, these differences turn out to be mostly small but very numerous. The first and most striking difference is that NewUse has almost twice as many non-zero cells as does Use. Nearly all of these extra non-zeros are very small, exactly the sort of thing to be reasonably ignored in the process of making a table. But it is precisely this "reasonable ignoring" that leads to the problem of many small negatives in the product-to-product tables calculated without the no-negatives algorithm.

To get a closer look at how Use and NewUse compare, we may first divide each column by corresponding industry's output and then look at the column sums of the absolute values of the differences of individual coefficients in the column. This comparison is shown in Table 9. Clearly the vast majority of industries show only small differences compatible with "reasonable ignoring" of small flows in the Use matrix. They, therefore, cast no serious doubt on the product-technology assumption or the usability of the Recipe matrix obtained by the no-negatives algorithm. If what we are interested in is the R matrix, we can ignore the small differences between Use and NewUse.

Table 9. Comparison of Use and NewUse

Sum of Absolute Differences	Count
.050 - .250	17
.030 - .050	24
.020 - .030	54
.010 - .020	117
.000 - .010	312

Table 10. Largest Differences between Use and NewUse

Sum |dif|	Column numb.	Column Name	Largest single difference Row	| dif |	Row name
0.250	272	Asbestos products	31	0.023	Misc. nonmetallic minerals
0.232	88	Sausages	3	0.151	Meat animals
0.167	125	Vegetable oil mills, nec	15	0.074	Oil bearing crops incl s
0.118	493	Auto rental & leasing	232	0.025	Petroleum refining
0.088	128	Edible fats and oils, nec	15	0.043	Oil bearing crops incl s
0.088	126	Animal & marine fats	126	0.038	Animal & marine fats &
0.086	87	Meat packing plants	3	0.057	Meat animals
0.079	285	Primary metals, nec	22	0.006	Iron & ferroalloy ore m
0.079	225	Manmade organic fibers	212	0.036	Indl chem: inorg & org
0.074	450	Transportation services	232	0.019	Petroleum refining
0.068	123	Cottonseed oil mills	5	0.048	Cotton
0.065	357	Carburetors, pistons,	391	0.011	Electronic components
0.060	99	Pickles, sauces	1	0.011	Dairy farm products
0.060	95	Canned & cured sea food	19	0.039	Commercial fishing
0.059	139	Yarn mills & textile fini	212	0.035	Indl chem: inorg & org
0.055	459	Sanitary services, steam	413	0.018	Mechanical measuring devices
0.051	248	Leather gloves	244	0.012	Leather tanning

There are, however, a few cases that should be looked at more closely. Table 10 shows a list of all of industries which had a sum of absolute differences greater than .050. We will look at the top five.

For Asbestos products, the cause of the difference is quickly found. The fundamental raw material for these products comes from industry 31 Misc. non-metallic minerals. Over forty percent of the output of asbestos products, however, is produced in industry 400 Motor vehicle parts and accessories, but this industry buys neither miscellaneous non-metallic minerals nor asbestos products. In other words, it seems to be making almost half of the asbestos products without any visible source of asbestos. This anomaly seems to me to be an oversight in making the Use matrix which should be simply corrected. If our only interest is the Recipe matrix, the algorithm seems to have computed pretty nearly the right result from the wrong data. On the other hand, if we want to correct the Use table, NewUse, gets us started with the right entry for Misc. non-metallic minerals into both Motor vehicle parts and Asbestos products. To keep the right totals in these two columns of Use will require manual adjustments.

The second largest difference between Use and NewUse shown in Table 10 is in the input of meat animals into Sausage. The Sausage industry is shown in the Use matrix to buy both animals ($655) and slaughtered meat ($9688). It had a primary output of $13458 and a secondary output of $2612 of products primary to Meat packing. Meat packing had a secondary output of $4349 of sausage. Now in Meat packing, the cost of the animals is over eighty percent of the value of the finished product, so the purchases of animals in the Sausage industry is insufficient to cover even the secondary meat output of this industry, not to mention making any sausage. In making Recipe, the input of animals directly into sausage is driven to zero and cut off there rather than being allowed to become negative. Then when NewUse is made, the direct animal input for all the secondary production of meat packing products is put in, thus making a flow some six times as large as the purchase of meat animals by the Sausage industry in the original Use matrix.

What I believe to be really happening here is that Sausage plants are mostly buying halves of slaughtered animals from meat packers, selling off the best cuts as a secondary product, and using the rest to make sausage. Over in the Meat packing plants, the same thing is happening. Fundamentally, there is only one process of sausage making. The question is how to represent it in the input-output framework. The simplest representation of it in the Use matrix would be to have packing houses sell to sausage plants only the meat that would be directly used in sausage. The rest, the choice cuts sold off as meat by Sausage mills, would simply be considered sold by the packers without ever passing through the Sausage mills. The industry output of Sausage mills is reduced but cost of materials (namely, meat) is reduced by exactly the same amount, so there is no need to adjust other flows. Product output of meat is reduced, but not the industry output. Thus, a slight adjustment in the accounting makes it broadly compatible with the product-technology assumption. The seventh item in Table 10, by the way, is just the other side of this problem.

The third largest of the discrepancies lies in row 16, oil-bearing crops, of industry 125 Vegetable oil mills n.e.c (not elsewhere classified). The differences in the underlying flows is not large, \$298 in Use and \$251 in NewUse, but it turns up in Table 10 because the cost of these oil crops is such a large fraction of the output of the Vegetable oil mills. A comparison of the oil-bearing crops row of Use and NewUse shows that NewUse has a number of small positive entries for industries where, as for Cheese, Use has a zero and where, moreover, it is highly implausible that there was any use of oil seeds. On the other hand, most of the large users of oil seeds, like Vegetable oil mills have had their usage trimmed back. The key to what is going on is found in industry 132 Food preparations n.e.c.. In Use, this industry bought \$558 from oil bearing crops, nearly twice the consumption of the vegetable oil mills themselves. Peanut butter, as noted above, is in this catchall industry. That fact, by itself, is not a problem. The problem is that about a quarter of the production of products primary to this industry are made in other industries. In fact, most of the food manufacturing industries have some secondary production of the miscellaneous food preparations. Probably "preparations" made in the Cheese industry are quite different from those made in the Pickles industry. And it certainly makes no sense to spread oil seed inputs all over the food industries. Here we have a clear case of the inapplicability of the product-technology assumption if all these secondary products are considered to be truly the same product. On the other hand, as argued above, the very heterogeneity of the products makes it appropriate to consider each as a primary product of the industry which produces it and then "sell" it, via the Use matrix, to Food preparations for distribution. In the next pass at making Recipe, this change is to be made.

The vegetable oil industries also present another interesting case of apparent but perhaps not real violation of the product-technology assumption, which shows up in the fifth item in Table 10. Industry 125 Vegetable oil mills n.e.c. has inputs of oil-bearing crops, cotton, and tree nuts totaling \$437. It uses these oil sources to produce a primary output of \$572. Industry 128 “Edible fats and oils” produces \$92 of products primary to 125 without a penny of any of these inputs! Surely this is flat violation of the product-technology assumption. But is it really? "Edible fats and oils" buys lots of the products primary to Vegetable oil mills. Thus, it is entirely possible to have two bottles of chemically identical oil made of identical raw materials by identical refining processes but with one bottle made entirely in Vegetable oil mills while the oil in the other bottle was pressed in those mills and then sold to Edible fats and oils for finishing. We might call this situation “trans-market product technology.” Our algorithm gave the right answer for the Vegetable oil mills column of Recipe, that is, it combined output of products primary to the oil mills with the inputs of oil sources which this industry had.

The fourth largest discrepancy in Table 10 is for the gasoline input into Automobile renting and leasing. Use shows \$1131; Recipe ups that to \$1197.2; but NewUse cuts it back to \$565.5. What happened? The problem is that slightly more than half of the output Auto renting is produced in Credit agencies, with a minuscule input of gasoline. When NewUse is made, more than half of the gasoline in Recipe is allocated over to Credit agencies. Here we are confronted with a failure of the product-technology assumption not because of different processes for producing the same product but because two quite different products have been called one and the same in the accounting system. The output of the Credit agencies, long-term leasing, is quite distinct from the short-term renting, which is were the gasoline was used. The best solution would be to recognize the difference of the two products. Short of that, the worst of the problem can be fixed by turning the secondary transfer from Credit agencies to Automobile rental into a primary flow. The present Recipe matrix, incidentally, is about right in the gasoline row but makes no connection between a final demand for automobile renting and leasing and the output of credit agencies.

From these five or six cases, we see that our algorithm cannot be expected to give usable results on the first try. The problems are likely to lie, however, neither in the fundamental economic reality nor in the algorithm, but in an accounting system which needs a few modifications in Use and Make to make it operational in our sense. Most importantly, the algorithm gives us the means to identify the places that need attention and a way of progressing systematically through the problems. It also provides a way of producing a final, non-negative Recipe matrix that implies a NewUse matrix close enough to the modified Use matrix that the differences can be safely ignored.

Making an input-output table requires fussing over details, and making a good Recipe matrix with the algorithm presented here is no different in this respect from any other part of the process. Use of the algorithm reveals and pinpoints problems. Moreover, the important problems are likely to be small in number. We have covered all of those causing a difference of as much as .100 between columns of Use and NewUse. To get to a Recipe table we would be ready to accept might require another week's work. But in the total effort which went into making this table, that is minuscule. Most importantly, the use of the algorithm gives us a way to work on the problems rather than just wring our hands over negatives.

In this sense, this algorithm has performed satisfactorily over many years on every U.S. table since 1958. The use of the method seems to me to deserve to become a standard part of making input-output tables and, in particular, for making product-to-product tables.

7. The Computer Program

The C++ code for this algorithm, using functions from BUMP, the Beginner's Understandable Matrix Package, for handling matrices and vectors, is given below. It is reproduced here because the code shows more clearly than the verbal or formulaic description exactly what is done. The program and the supporting BUMP code made be downloaded from the Inforum Internet site: www.inforum.umd.edu. The main program here reads in the matrices that were used in the examples. The main program for the actual calculations of the full-scale American matrices is significantly larger and has various diagnostic output, such as that shown in Table 10. It is available on request.

In using the algorithm, it is important for documenting what has been done to have a method of input of the original Use and Matrix matrices that preserves the original version at the top of the input file and introduces the modifications as over-rides later in the file. It is also important to have software, such as ViewMat, which will show corresponding columns of several large matrices side-by-side in a scrolling grid. ViewMat is also available on the Inforum Internet site.

```
#include <stdio.h>            // for printf();
#include <math.h>             // for abs()
#include "bump.h"
int purify(Matrix& R, Matrix& U, Matrix& M, float toler);

void main(){
```

```
        Matrix Use(5,5), Make(5,5), R(5,5), NewUse(5,5);
        Use.ReadA("Use.dat");
        Make.ReadA("Make.dat");
        purify(R,Use,Make,.000001);
        R.Display("This is R");
        writemat(R,”Recipe”);
        NewUse = R*(~NewUse);
        writemat(NewUse,”NewUse”);
        tap();
        printf("\nEnd of calculations.\n");
        }

/* Purification produces a produt-to-product (or Recipe) matrix R from     a Use matrix U and
a Make matrix M.  M(i,j) shows the fraction of       product j made in industry i.  U(i,j)
shows the amount of product i used in industry j.  The product-technology assumption leads us
to expect that there exits a matrix R such that U = RM'.  If, however, we compute R =
U*Inv(M') we often find many small negative elements in  R.  This routine avoids those small
negatives in an iterative process.
*/

int purify(Matrix& R, Matrix& U, Matrix& M, float toler){
        int row, i, j, m, n, iter, imax;
        const maxiter = 20;
        float sum,rob,scale,dismax,dis;
        n = U.rows();  // n = number of rows in U
        m = U.columns(); // m = number of columns in U
        Vector C(m), P(m), Flow(m), Discrep(m);
        // Flow is row of U matrix and remains unchanged.
        // P becomes the row of the purified matrix.
        // C is the change vector at each iteration.
        // At the end of each iteration we set P = Flow + C, to start // the next iteration.

        // Purify one row at a time
        for(row = 1; row <= n; row++){
                C.set(0.); // C, which will receive the changes, is
                // initialized to zero.
                // P = Flow + C will be the new P.
                pulloutrow(Flow,U,row);
                P = Flow;
                iter = 0;
                start: iter++;
                for(j = 1; j<=m; j++){
                // Calculate total claims from other industries on
                // the inputs into industry j.
                        sum = 0;
                        for(i = 1; i <= m; i++){
                                if(i == j) continue;
                                rob = P[i]*M(j,i);
                                sum += rob;
                                C[i] += rob;
                                }
                        // Did we steal more from j than j had?
                        if (sum > Flow[j] && sum > 0){
                                // scale down robbery
                                scale = 1. - Flow[j]/sum;
                                for(i = 1; i <= m; i++){
                                        if(i == j) continue;
                                        C[i] -= scale*P[i]*M(j,i);
                                        }
                                sum = Flow[j];
                                }
                        C[j] -= sum;
                        }
                // Check for convergence
                imax = 0;
                dismax = 0;
                for(i = 1; i <= m; i++){
                        dis = fabs(P[i] - Flow[i] - C[i]);
                        Discrep[i] = dis;
                        if(dis >= dismax){
                                imax = i;
                                dismax = dis;
                                }
                        }
                P = Flow + C;
                C.set(0);
                if(dismax > toler){
                        if(iter < maxiter) goto start;
```

```
				printf(
				"Purify did not converge for row %d. Dismax = %7.2f. Imax = %d.\n",
				row,dismax,imax);
				}
		putinrow(P,R,row);
		}
	return(OK);
	}
```

References

Almon, C. (1970) Investment in input-output models and the treatment of secondary products, *Input-Output Techniques, vol. 2, Applications of Input-Output Analysis*, pp.103-116 (Amsterdam, North Holland Publishing Co.)

Almon, C., Buckler, M., Horwitz, L., and Reimbold, T.,(1974) *1985, Interindustry Forecasts of the American Economy* (Lexington, Lexington Books) pp.151-154.

European system of accounts: ESA 1995, Transmission programme of data. Eurostat.

Rainer, N. and Richter, J. (1992) Some Aspects of the Analytical Use of Descriptive Make and Absorption Tables. *Economic Systems Research,* 4, pp.159 - 172

Steenge, A.E. and Konijin. A new Approach to Irreducibility in Multisectoral Models with Joint Production. *Economic Systems Research,* 4, pp 125-132

The *System of National Accounts 1993* (published by the United Nations, the World Bank, the IMF, the OECD, and the European Union)

ten Raa, Thijs, D. Chakraborty, and J.A. Small (1984) An Alternative Treatment of Secondary Products in Input-Output Analysis, *Review of Economics and Statistics*, pp. 88-97.

ten Raa, Thijs (1988) An Alternative Treatment of Secondary Products in Input-Output Analysis:Frustration", *Review of Economics and Statistics,* pp. 535-538.

Van Rijckeghem (1967) "An Exact Mehod for Determining the TechnologyMatrix in a Situation with Secondary Products," *Review of Economics and Statistics,* pp. 607-8.