Getting Started
ioanalysis v_0.3.2

R | E | A | L[1]
University of Illinois at
Urbana-Champaign

Abstract

This paper is an introduction to using the R package `ioanalysis`. It introduces the package by describing the `InputOutput` object that is the foundation of this package. All of the examples are using an attached example data set named `toy.IO`. Once the `InputOutput` object has been created, we proceed through basic operations, followed by more advanced computations. `ioanalysis` is available on CRAN at https://cran.r-project.org/package=ioanalysis. All applications are used through RStudio.

[1]Regional Economics Applications Laboratory: http://www.real.illinois.edu/

Contents

1 Getting Started

`ioanalysis` is based around the `InputOutput` object created with `as.inputoutput()`. Input-output tables come in many different forms, layouts, and orientations, so the first step is to organize the data in a way that the program can understand. Currently, the package is not set up to handle supply and use matrices or physical matrices. Once the `InputOutput` object is made, the remaining functions in the package become simpler and user friendly. To illustrate how to create this object, and example data set call `toy.FullIOTable` is attached to the package.

Below is code recommended for first time users. This is the best way to become acquainted with the package and its features.

```
install.packages("ioanalysis") # Adding the package to your computer
library(ioanalysis)            # Loading the package into your R session
help(package = "ioanalysis")   # Listing available functions
```

Recall that if there is confusion of how to use a function such as `easy.select()`, you can always run the following code for a description of function inputs and uses:

```
?easy.select
```

The rest of the documentation is assuming the reader is familiar with Input-Output (IO) tables. For those who need more details, visit the appendix. The labels of each matrix is located there as well.

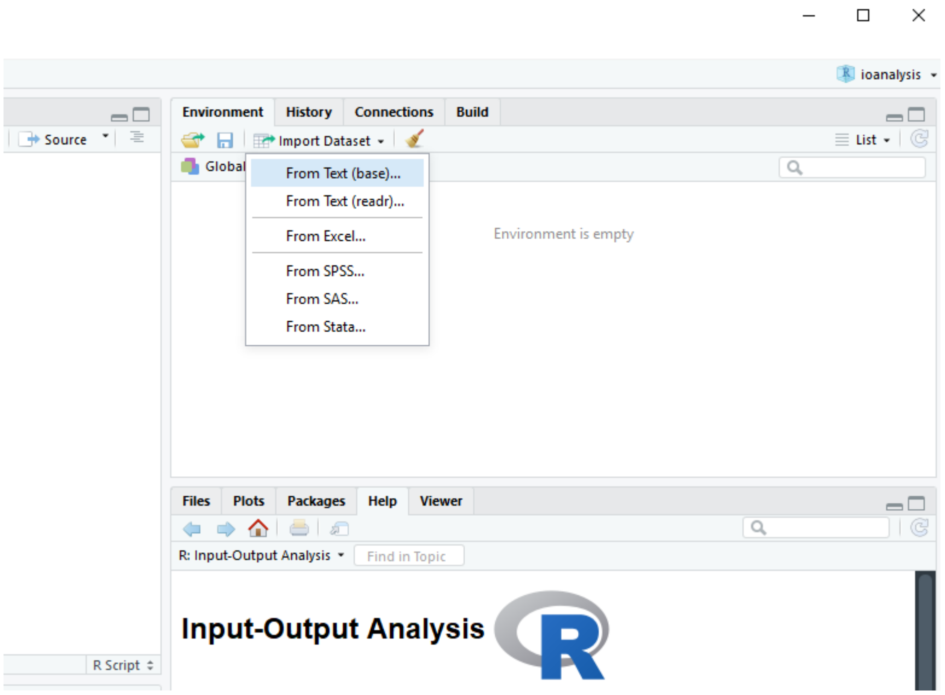
1.1 Loading your Data

There are a few ways to get your data loaded into R. Note that depending on the size[2] of your data, loading it into R could take quite some time. Perhaps the easiest way is to use RStudio's "Import Dataset" feature as demonstrated in figure 1.

Another way to load your data is to call upon your data at its source directly. Please note the delimitation of you data and use the corresponding import function in R. Here is an example for tab delimited and comma delimited.

[2]For large packages, consider using `fread()` from the package `data.table`.

Figure 1: Importing Data



```
readLines("C:/PATH/TO/DATA/DATANAME.csv", n = 10) # Examine first 10 rows
?read.table # Details on functions for loading data
io1 <- read.csv("C:/PATH/TO/DATA/DATANAME.csv", header = TRUE)
io2 <- read.table("C:/PATH/TO/DATA/DATANAME.txt", header = TRUE)
```

If your data is not saved as one object, you may need to make multiple objects to hold your data. Once all of your data is loaded into R, you can begin making your `InputOutput` object.

1.1.1 as.inputoutput()

This package is based around an `InputOuptut` object that holds on to all the matrices that compose a full Input-Output system. The reason for this is twofold. Firstly, this allows for easy use in the remainder of the functions used in the package. Secondly, it acts as a safeguard against user error.

There are several matrices that can be stored in the `InputOutput` object. However, only three matrices are required to create the object: the **intermediate transactions** matrix (`Z`), the **region-sector label** matrix (`RS_Label`), and the **total production** matrix (`X`). If only these three matrices are supplied, the final demand matrix (`f`) matrix is calculated for you.

Table 1: Required Matrices

Matrix	Required	Description
`Z`	Yes	$n \times n$ matrix of **intermediate transactions**: units of currency
`RS_label`	Yes	$n \times 2$ label matrix: first column region, second sector
`f`	No	$n \times p$ matrix of **final demand**
`f_label`	No	$2 \times n$ label matrix: first row region, second regional accounts
`E`	No	$n \times er$ matrix of **exports**
`E_label`	No	$2 \times er$ label matrix: first row region, second export type
`X`	Yes	$n \times 1$ matrix of **total production**
`V`	No	$p \times n$ matrix of **value added**
`V_label`	No	$q \times 1$ label matrix of value added types
`M`	No	$m \times n$ matrix of **imports**
`M_label`	No	$m \times 1$ label matrix of import types
`A`	No	$n \times n$ **technical input coefficient** matrix
`B`	No	$n \times n$ **technical output coefficient** matrix
`L`	No	$n \times n$ **Leontif** inverse
`G`	No	$n \times n$ **Ghoshian** inverse

Let $n = \#$ regions $\times$ # sectors, $r = \#$ regions, $e = \#$ export types, $m = \#$ import types, and $p, q =$ arbitrary length.

Almost every matrix[3] that is provided needs a label matrix to accompany it. This is so the package knows how to match one matrix to another. If you only have one region, there still needs to be a respective column or row populated in the label matrix; the call functions do not like working with empty cells. To get us started, let us look[4] at the attached example input-output matrix.

```
data("toy.FullIOTable") # Loading the dataset into the current session
View(toy.FullIOTable)   # Taking a look at the dataset
?toy.FullIOTable        # Shows how the data was generated
```

Here we see that we have a well-populated input-output table with all but the last four matrices from table 1. Now we get to create out first `InputOutput` object. The data was designed to be annoying to illustrate potential issues when making the object. To construct the object, you can either assign the matrices individually (recommended) prior to using `as.inputoutput()` or you can directly put in the arguments. In the following example, we use both.

[3] f is not capitalized because `F == FALSE`

[4] Caution: If you use `View()` on a large dataset, R will take a long time to load the image.

```
Z <- matrix(as.numeric(toy.FullIOTable[3:12, 3:12]), ncol = 10)
f <- matrix(as.numeric(toy.FullIOTable[3:12, c(13:15, 17:19)]),
            nrow = dim(Z)[1])
E <- matrix(as.numeric(toy.FullIOTable[3:12, c(16, 20)]), nrow = 10)
X <- matrix(as.numeric(toy.FullIOTable[3:12, 21]), ncol = 1)
V <- matrix(as.numeric(toy.FullIOTable[13:15, 3:12]), ncol = 10)
M <- as.numeric(toy.FullIOTable[16, 3:12])
fV <- matrix(as.numeric(toy.FullIOTable[15:16, c(13:15,17:19)]), nrow = 2)

# Note toy.FullIOTable is a matrix of characters: non-numeric
toy.IO <- as.inputoutput(Z = Z, RS_label = toy.FullIOTable[3:12, 1:2],
                         f = f, f_label = toy.FullIOTable[1:2,
                                                          c(13:15, 17:19)],
                         E = E, E_label = toy.FullIOTable[1:2, c(16, 20)],
                         X = X,
                         V = V, V_label = toy.FullIOTable[13:15, 2],
                         M = M, M_label = toy.FullIOTable[16,2],
                         fV = fV, fV_label = toy.FullIOTable[15:16, 2])

class(toy.IO)
```

A few words of caution. R performs a calculation roughly every 8e-10 seconds (of course, depending on your computer). The number of computations per matrix is inversion is n^3, where n is the dimension of your square matrix. If you have a 5000×5000 matrix, it should take roughly 100 seconds to invert. `as.inputoutput()` calls upon `leontief.inv()` to calculate L and `ghosh.inv()` to calculate `G`. Both of these functions will print an estimated time of inversion if the estimated time is longer than six seconds.

To save time in future sessions using your `InputOutput` object, you can save the file as an `.rda` file. These files are much faster to load in R compared to say `.csv` files. When saving the file, R will save the file to your working directory. You can either manually move the saved file to where you want it to be saved, or change the working directory itself.

```
getwd()                            # Get working directory
setwd("C:MY/FAVORITE/FOLDER")      # Setting new wokring directory
getwd()                            # Checking to see if it worked
save(toy.IO, file = "toy.IO.rda") # Saving the file
```

If your input-output matrix is large, the generated `InputOutput` matrix will be large. R stores its current working session on RAM. If your object exceeds the physical limit your computer can store, it will not create the object and produce an error: `Error: cannot allocate vector of size XY.Z Mb`.

1.2 Table Operators: Adjusting Your Data

Some functions such as `agg.region()`, `export.coef()` and `import.coef()` require each region has exactly the same set of sectors. If your data does not have this feature, there are ways to balance the regions and sectors. The first step is to check your data for mismatches. If you have a mismatch, you can locate where the mismatches occur. Then there are two options on how to proceed; you can either combine sectors or combine regions.

1.2.1 check.RS()

`check.RS()` uses the RS_label matrix in the `InputOutput` object to identify whether or not each region has the exactly the same sectors. If you know each region has the same sectors, it is still useful to run this function because it will tell you if there are any misspellings. This package was not written to consider "`label`" as the same thing as "`LaBeL`". This function produces a logical output where `TRUE` indicates a balanced region-sector Input-Output table.

```
> check.RS(toy.IO)
 [1] TRUE
```

1.2.2 locate.mismatch()

If `check.RS` produces a `FALSE` output (each region does not have the same sectors) the next step is to identify where the mismatches occur. `locate.mismatch()` identifies which sectors are not found in all regions. If a sector is not in all regions, then a report is generated to indicate which regions have that sector, which regions do not have that sector, and where that sector is in the repository. To do so, we provide an example that has a mismatch.

```
toy.IO$RS_label <- rbind(toy.IO$RS_label,
                         c("Valhalla", "Wii"),
                         c("Valhalla", "Pizza"),
                         c("Valhalla", "Pizza"),
                         c("Valhalla", "Minions"))
MM <- locate.mismatch(toy.IO)

  WARNING:

  Pizza occurs more than once in Valhalla at 12 13
  Consider using agg.sector(toy.IO, sectors = 1, regions = 3,
                            newname = "Pizza")
  to combine these sectors into one or renaming sector.

> MM$Spaceships
$location
[1] 3 8

$regionswith
[1] "Hogwarts" "Narnia"

$regionswithout
[1] "Valhalla"
```

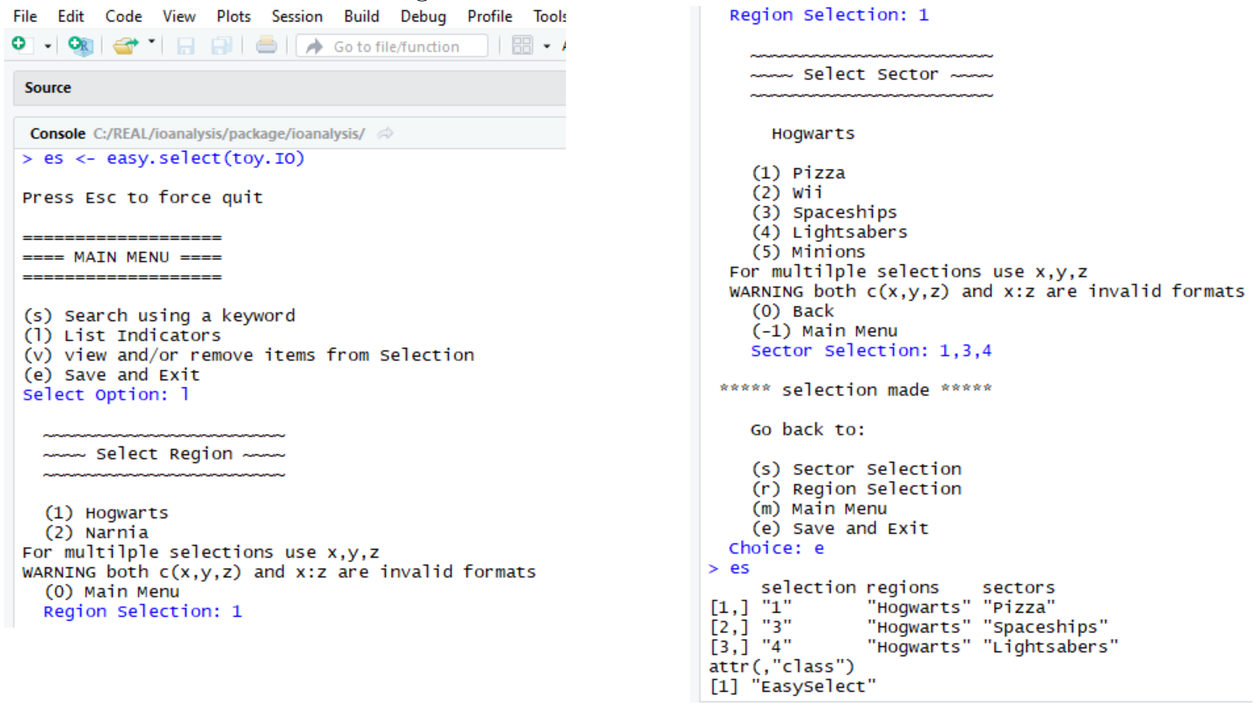
1.2.3 agg.sector() and agg.region()

Both aggregation functions can be used to handle mismatch discrepancies or to consider alternative compositions of the `InputOutput` object. As show in the previous example, if there is a duplicate of named sectors within a region, suggested code to consolidate is generated. It is important to note that these two functions generate entirely new `InputOutput` objects, which should be taken into size considerations. `agg.sector()` works as follows:

```
newIO <- agg.sector(toy.IO, sectors = c(1,2), newname = "Party.Supplies")
```

Here you can see that you need the original `InputOuput` object, the sectors you want to aggregate, and a name to give to the new sectors created. Here we are aggregating the "Pizza" and "Wii" sectors together. Instead of using the argument `sectors = c(1,2)`, it is equivalent to use `sectors = c("Pizza", "Wii")`. If you wish to use the name of the sector instead, you must spell each sector correctly with correct capitalization and punctuation. Additionally, you cannot mix spelling out a sector, and using another sector's numerical location. `agg.region()` works exactly the same as agg.sector, however, it requires each region has the same sectors.

Figure 2: Easy Select Example


File Edit Code View Plots Session Build Debug Profile Tools
Go to file/function
Source
Console C:/REAL/ioanalysis/package/ioanalysis/

```
> es <- easy.select(toy.IO)

Press Esc to force quit

===================
==== MAIN MENU ====
===================

(s) Search using a keyword
(l) List Indicators
(v) View and/or remove items from Selection
(e) Save and Exit
Select Option: l

  ~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~ Select Region ~~~~
  ~~~~~~~~~~~~~~~~~~~~~~~

  (1) Hogwarts
  (2) Narnia
For multilple selections use x,y,z
WARNING both c(x,y,z) and x:z are invalid formats
  (0) Main Menu
  Region Selection: 1
```

```
Region Selection: 1

  ~~~~~~~~~~~~~~~~~~~~~~~
  ~~~~ Select Sector ~~~~
  ~~~~~~~~~~~~~~~~~~~~~~~

    Hogwarts

  (1) Pizza
  (2) Wii
  (3) Spaceships
  (4) Lightsabers
  (5) Minions
For multilple selections use x,y,z
WARNING both c(x,y,z) and x:z are invalid formats
  (0) Back
  (-1) Main Menu
  Sector Selection: 1,3,4

***** selection made *****

  Go back to:

  (s) Sector Selection
  (r) Region Selection
  (m) Main Menu
  (e) Save and Exit
Choice: e
> es
     selection regions    sectors
[1,] "1"       "Hogwarts" "Pizza"
[2,] "3"       "Hogwarts" "Spaceships"
[3,] "4"       "Hogwarts" "Lightsabers"
attr(,"class")
[1] "EasySelect"
```

1.2.4 easy.select()

This is an interactive function that is designed to greatly simplify locating regions and sectors, resulting in an `EasySelect` object that can be used in functions [5] to select region and sector combinations. You can choose to select region-sector combinations by searching over regions first the sectors (or sectors first the regions), or by using a list to select desired regions and then desired sectors. If you choose multiple regions and then choose a sector that is not in all regions, a warning message is produced to let you know. Below is an image of what to expect.

The `EasySelect` object is a matrix with three columns. The first column is for the package to locate which column corresponds to which region-sector combination. The middle column is the name of the region and the last column is the corresponding sector. The last two columns are not used by the package, but are instead intended for the user to ensure the desired region-sector combinations were correctly chosen. For use in future examples, the package comes with a example data set called `toy.ES` that can be brought into the current session by the command `data("toy.ES")`.

[5]Comprehensive list: `extraction()`, `key.sector()`, `linkages()`, `multipliers()`, `upstream()`, and `vs()`

2 Coefficients, Matrices, and Indicators

Now that our `InputOutput` object has been constructed and adjusted, we can start working with the data to perform intraperiod analysis. What follows is a list of functions that produce coefficients that can be either by region-sector combinations, or combinations there of. Some of those functions require preliminary steps. Those steps have been made into their own function allowing the user to have access to as well.

2.1 Primary Functions

2.1.1 multipliers()

There are five different multipliers available in the package: `output`, `input`, `income`, `employment.closed`, and `employment.physical` as described in (Blair & Miller, 2009). It is possible to call multiple multipliers at once. In the literature there are a wide variety of methods to calculate each multiplier, so pay close attention to the formulation. To better understand what these multipliers tell us, consider an *output* multiplier of 1.2. This is interpreted as: if final demand was to increase by \$1, then total output would increase by \$1.20.

The `multipliers` function outputs a list over each specified region containing a table with rows for each specified sector and columns for the specified multipliers. Here, the numbers are interpreted as \$1 increase in the final demand for that specific region-sector combination. The following example illustrates how to use an `EasySelect` object.

```
> data(toy.IO); class(toy.IO)
[1] "InputOutput"
> data(toy.ES); class(toy.ES)
[1] "EasySelect"
> mult <- multipliers(toy.IO, toy.ES, multipliers = c("input", "output"))
> names(mult)
[1] "Hogwarts" "Narnia"
> mult$Hogwarts
              output    input
Wii         1.524694 1.658562
Spaceships  1.486583 1.383500
Lightsabers 1.447051 1.383186

> mult2 <- multipliers(toy.IO, region = "Narnia", sectors = c(1,2),
                       multipliers = "income", wage.row = 1)
```

Recall that this is randomly generated data, so your results will likely differ. Notice in `mult2` that regions are called by a character and sectors are called by numerics. It is equivalent to call `regions = 2, sectors = c("Pizza", "Wii")`, but calling `c(1, "Wii")` is invalid because it is now viewed as a vector of characters. The multipliers are calculated as follows:

1) `output` is calculated using the row sums Leontief inverse matrix by

$$O_j = \sum_{i=1}^{n} l_{i,j}$$

where $l_{i.j}$ is the i^{th} row and j^{th} column element of the Leontief inverse.

2) `input` is calculated using exactly the same technique as `input`, but instead uses the Ghoshian inverse matrix by

$$I_j = \sum_{i=1}^{n} g_{i,j}$$

where $g_{i.j}$ is the i^{th} row and j^{th} column element of the Ghoshian inverse.

3) To use the `wage` multiplier, the function needs to know which row corresponds to wages in the value added matrix (V). Clearly, the function will refuse to run if there is not a value added matrix. The multiplier is calculated as:

$$W_j = \sum_{i=1}^{n} \omega_i l_{i,j}$$

where $\omega_i = \frac{w_i}{X_i}$ is the wage divided by total output for each region-sector combination.

4) The first employment multiplier is `employment.closed` which is for input-output matrices that are closed to labor; that is, one of the sectors is employment. The function can handle multiple sectors of employment, for example if there is low-skill and high-skill labor. Here, the multiplier is

$$E_j^c = \sum_{i=1}^{n} \epsilon_i l_{i,j}$$

where ϵ_i is the rows of the i^{th} column in the matrix of technical input coefficients corresponding to labor and $l_{i.j}$ is the i^{th} row and j^{th} column element of the Leontief inverse. Note that ϵ_i can be a vector.

5) `employment.physical` works exactly the same as `employment.closed` except it calls from the physical matrix. The physical matrix is not a standard matrix, so you will have to add it to your `InputOutput` matrix manually.

$$E_j^p = \sum_{i=1}^{n} \rho_i l_{i,j}$$

is the rows of the i^{th} column in the physical matrix corresponding to labor and $l_{i.j}$ is the i^{th} row and j^{th} column element of the Leontief inverse, where ρ_i is the element in the physical corresponding to employment.

2.1.2 linkages()

Linkages are an indicator of how the input-output system would respond if you increase sector j's output by $1 as described in (Blair & Miller, 2009). *Backward linkages* (BL) tell us how much sector j's quantity demanded for inputs increases due to its increased production. *Forward linkages* tell us how much the output sector j's increases the supply of sector j's product as other industries demand more. Then for both type of linkages.

Then there is the distinction of *total*, *direct*, and *indirect* linkages. Direct linkages only consider the "first order" dependency or partial effects. In terms of BL, how much will j's demand for inputs increase without the other sectors adjusting to the increased production. Total effects is how the system adjusts to sectors j's increase in production. Indirect effects is simply the difference of total and direct effects.

Furthermore, if the system is multi-regional there are *intraregional*, *interregional*, and *aggregate* effects. Considering total BL, intraregional is how much sector j's demand increases due to changes within the region, interregional is changes due the remaining system, and the aggregate effect is simply the sum. Note that aggregate effect is what is produced with default settings.

`linkages()` outputs a list over regions. Each element is a table of the specified region, where the rows are for each sector and the columns are the types of specified region. The system to help identify the different linkages is as follows:

Table 2: `linkages()` Affixes

Prefix		Infix		Suffix	
`BL`	Backward Linkages	`.intra`	intraregional	`.tot`	total effects
`FL`	Forward Linkages	`.inter`	interregional	`.dir`	direct effects
		`.agg`	aggregate		

```
> data(toy.IO); class(toy.IO)
[1] "InputOutput"
> link <- linkages(toy.IO, intra.inter = TRUE)
> names(link)
[1] "Hogwarts" "Narnia"
> link$Hogwarts
             BL.intra.tot FL.intra.tot BL.inter.tot FL.inter.tot
BL.agg.tot FL.agg.tot
Pizza            1.315163     1.272749    0.2938851    0.3446877
1.609048   1.617437
Wii              1.263724     1.247795    0.2609708    0.2749895
1.524694   1.522785
Spaceships       1.268119     1.399085    0.2184646    0.2892265
1.486583   1.688311
Lightsabers      1.234348     1.354568    0.2127030    0.3129667
1.447051   1.667535
Minions          1.270830     1.164668    0.1262444    0.2279104
1.397074   1.392579
```

Note that this data is randomly generated and does not represent a real economy. In reality, aggregate linkages should fluctuate above and below 1. Here we see this is not the case, but instead the linkages fluctuate around 1.5.

There is an additional option to normalize the linkages, where the default is not to normalize. Note that if you do normalize, then the total effect is not the same as the sum of direct and indirect effects. Similarly the aggregate effect would no longer be the sum of intra- and interregional effects. Normalizing is equivalent to adding the denominator in the following identities. If you want to visualize the non-normalized version of the equations, simply put your thumb over the denominator.

$$BL_j^{total} = \frac{\sum_{i=1}^n l_{i,j}}{\frac{1}{n}\sum_{j=1}^n \sum_{i=1}^n l_{i,j}} \qquad FL_j^{total} = \frac{\sum_{j=1}^n g_{i,j}}{\frac{1}{n}\sum_{j=1}^n \sum_{i=1}^n g_{i,j}}$$

Notice instead we could use matrix notation, therefore stacking the above identities into a matrix as follows, where $\mathbb{1}$ is a column vector of ones:

$$BL.total = \frac{\mathbb{1}'L}{\frac{1}{n}\mathbb{1}'L\mathbb{1}} \qquad FL.total = \frac{G\mathbb{1}}{\frac{1}{n}\mathbb{1}'G\mathbb{1}}$$
$$BL.dir = \frac{\mathbb{1}'A}{\frac{1}{n}\mathbb{1}'A\mathbb{1}} \qquad FL.dir = \frac{B\mathbb{1}}{\frac{1}{n}\mathbb{1}'B\mathbb{1}}$$

Let J be the multiregional matrix corresponding to the appropriate matrix as from above (i.e. for BL.total J = L). Then let $J_{r,r}$ denote the block of J corresponding to the column and row of the domestic region, $J_{s,r}$ correspond to all other regions' rows but domestic rows, $J_{\cdot,r}$ correspond to all rows, but only the domestic columns, and etc. Also, assume $\mathbb{1}$ is the appropriate dimension. Then we have

$$BL.intra = \frac{\mathbb{1}'J_{r,r}}{\frac{1}{n^2}\mathbb{1}'J_{r,r}\mathbb{1}} \qquad FL.intra = \frac{J_{r,r}\mathbb{1}}{\frac{1}{n^2}\mathbb{1}'J_{r,r}\mathbb{1}}$$

$$BL.inter = \frac{\mathbb{1}'J_{s,r}}{\frac{1}{n^2}\mathbb{1}'J_{s,r}\mathbb{1}} \qquad FL.intra = \frac{J_{r,s}\mathbb{1}}{\frac{1}{n^2}\mathbb{1}'J_{r,s}\mathbb{1}}$$

$$BL.intra = \frac{\mathbb{1}'J_{\cdot,r}}{\frac{1}{n^2}\mathbb{1}'J_{\cdot,r}\mathbb{1}} \qquad FL.intra = \frac{J_{r,\cdot}\mathbb{1}}{\frac{1}{n^2}\mathbb{1}'J_{r,\cdot}\mathbb{1}}$$

2.1.3 key.sector()

`key.sector()` uses the `linkages()` function to indicate the relative dependence or independence of each sector ((Blair & Miller, 2009)). There are options to specify which region-sector combinations you wish to specify. If you want to refine further, you can use an `EasySelect` object. there is an option to specify which kind of linkages (`"direct"` for example) to use in evaluating the relative dependence. Furthermore, you can use intraregional and interregional linkges. Lastly, there is the option to choose the critical value you wish to use for comparison.

```
> data(toy.IO); class(toy.IO)
[1] "InputOutput"
> key <- key.sector(toy.IO, type = "total", crit = 1.5)
> key$Narnia
              BL.tot   FL.tot key.tot
Pizza       1.470970 1.638779      II
Wii         1.587289 1.295175      IV
Spaceships  1.612023 1.516649     III
Lightsabers 1.482453 1.421828       I
Minions     1.475032 1.509676      II
```

The default type of linkage is `direct`. In the following example we use `total` to be consistent with the `linkages` example. The critical value is also set[6] to 1.5 for illustrative purposes. The linkages are calculated for both regions, however, the \$ operator is used to only call upon a specific region. Notice that additional information beyond the key sector indicator is printed. This is to allow the user to evaluate how much room is available when putting the regions into these specified bins.

[6]In general it is recommended to use a critical value of 1. Make sure you understand how the interpretation of the key sector indicator changes (with respect to linkages) when the critical value changes.

Table 3: `key.sector()` Indicators

	FL $< crit$	FL $> crit$
BL $< crit$	I	II
BL $> crit$	IV	III

Here we can see how the indicators identify how the region-sector combinations are connected to one another. Notice that these numbers are region-sector aggregates. Therefore, there may exist firms within these sectors that could be independent of both supply and demand, but the industry as a whole could be dependent on both. Specifically, we have:

I - Generally independent

II - Generally dependent on interindustry demand, Generally independent on interindustry supply

III - Generally dependent

IV - Generally independent on interindustry demand, Generally dependent on interindustry supply.

From the above example, lets consider Narnia's Wii sector. The backward linkages of 1.59 is above the critical value of 1.50. This tells us the sector has a dependency on the inputs it uses to produce its products. However, the forward linkage of 1.30 is smaller than the chosen critical value. That is, the sector is less reliant on other industries to purchase its product as an input. Collectively, this industry relies on its inputs to produce but does not rely on other idustries to use its product as an input.

2.1.4 inverse.important()

If you want to examine how a productivity shock to a specific region-sector changes the dynamics in an input-output environment, then `inverse.important()` is the function for you (Blair & Miller, 2009). This function produces the change in the Leontief inverse matrix due to this productivity shock. This is calculated as follows:

$$\Delta L = \frac{\Delta a_{ij}}{1 - l_{ji}\Delta a_{ij}} F_1(i,j)$$

where $F_1(i,j)$ is the first order field of influence and Δa_{ij} is the % size of change.

2.1.5 upstream()

The goal of this function is to indicate how close the region-sector combination is to the final use on average. This follows the methodology of Antràs et al. (2012) to calculate upstream coefficients. The coefficients are weakly bounded below by one. A value close to one indicates that the product is near its final use, conversely, a value relatively larger than one indicates that the product is primarily used as an intermediate good.

`upstream()` essentially creates a new weighted Leontief inverse matrix by adjusting the matrix of technical input coefficients. It does so by multiplying the ratio of total production to total production net of net exports. `upstream()` calls upon the export.total() and import.total() to calculate the upstream coefficients. The calculation proceeds as follows:

$$d_{i,j} = a_{i,j}\frac{x_i}{x_i + e_{i,j} - m_{i,j}}$$
$$U = (I - D)^{-1}$$
$$u_i = \sum_{j=1}^{n} u_{i,j}$$

where capital letters are matrices and lower case letters are scalars following the notation of this package. The end result is a list over regions of the sector vectors of u_i. If you do not wish to have all regions and sectors, you can specify combinations of regions and sectors, or you can use an `EasySelect` object.

```
> # upstream
> data(toy.IO); class(toy.IO)
[1] "InputOutput"
> u1 <- upstream(toy.IO)
> u1$Hogwarts
            Hogwarts
Pizza       1.613984
Wii         1.622987
Spaceships  1.528488
Lightsabers 1.548542
Minions     1.523174
```

2.1.6 vs()

The vertical specialization is an indicator of the magnitude of using imported inputs to producing exported goods as shown in Hummels et al. (2001). To make these values comparable across region-sector combinations, the vertical specialization is divided by total exports as suggested. This is to avoid larger region-sectors that are relatively less vertically specialized dominating region-sectors that are small but very vertically specialized. Consequently, these values fall between zero and one, where a one would indicate fully vertically specialized.

Firstly, there are a few words of caution. Running this function only makes sense if your `InputOuput` object is a multiregional system. For each region used in analysis, a new Leontief inverse matrix is calculated. If the system is large, this can take quite some time. If you do not need all regions, you can either specify regions in advance using either exclusively numbers or matching region names verbatim, or you can use an `EasySelect` object.

For the calculation itself, `vs()` calls upon the import iteration of export.coef(). The vertical specialization is calculated as follows:

$$\frac{vs_r}{X_r^{total}} = \frac{1}{X_r^{total}} A_r^M L_r X_r$$

where X_r^{total} is the total exports for region r, A_r^M is the matrix of technical import coefficients, L_r is the domestic Leontief inverse calculated from the domestic matrix of technical coefficients (i.e. A_{rr} not the full A matrix), and X_r is the vector of total exports.

Let us take at how to use this function and possible error codes.

```
> vs1 <- vs(toy.IO, regions = "all")
Warning messages:
1: In import.coef(io, region = r) :
WARNING: io$M exists. This means the import coefficient matrix may be biased

 Coefficient matrix is still calculated.
2: In import.coef(io, region = r) :
WARNING: io$M exists. This means the import coefficient matrix may be biased

 Coefficient matrix is still calculated.
> vs1$Hogwarts
                    vs
Pizza       0.04189803
Wii         0.03642840
Spaceships  0.03693951
Lightsabers 0.03389838
Minions     0.03616019
```

Notice firstly the warning messages. Because the function calls upon import.coef() to generate the A^M matrix and this `InputOutput` object has an import matrix, it is possible there is double counting occuring. This is because `import.coef()` uses the elements within the matrix of technical coefficients from other regions to calculate the imports.

To get the exact vertical specialization for the region (or any other combination of regions sectors), a little work on the user side needs to occur. For illustrative purposes, let e_{rs}^t be the total exports for region-sector rs, and equivalently define vs_{rs} as the raw vertical specialization. Then the last command is equivalent to $\sum_{s\in S} \frac{vs_{rs}}{e_{rs}^t}$. This needs to be adjusted to $\frac{\sum_{s\in S} vs_{rs}}{\sum_{s\in S} e_{rs}^t}$. The exact change is calculated as follows:

```
i <- which(toy.IO$RS_label[, 1] == "Hogwarts")
xp <- export.total(toy.IO)
vsHog <- sum( (vs1$Hogwarts * xp[i, ]) * (1/sum(xp[i, ])) )
```

2.2 Feedback Loops

2.2.1 feedback.loop()

`feedback.loop()` solves the Linear Programming Assignment (LPA) problem for the intermediate transaction matrix. It does so in a hierarchical fashion, such that it calculates the first loop that generates that largest flow of intermediate transactions, then removes those matches from consideration to calculate the next feedback loop, and so on.

Each feedback loop produces an assignment following a sudoku-like constraint such that each row only has one column selected and each column only has one row selected (Sonis et al., 1995). It is common to have subloops within each loop.

The LPA problem solves the following optimization problem:

$$\begin{aligned} \max_S \; vec(Z)'vec(S) \quad & \text{such that:} \\ 1) \quad & A_{row}vec(S) = \mathbb{1} \\ 2) \quad & A_{col}vec(S) = \mathbb{1} \\ 3) \quad & 0 \leq vec(S) \leq \mathbb{1} \end{aligned}$$

where $vec(Z)$ is the vectorized intermediate transaction matrix, $vec(S)$ is the vectorized selector matrix of ones and zeros (in the sudoku-like pattern), A_{row} ensures the sum of the rows in S is one, A_{col} ensures the columns sum to one, and constraint (3) ensures each element is bounded between zero and one.

The matrices in constraints (1) and (2) are defined as follows:

$$A_{row} = \begin{bmatrix} 1 & 0_{1\times(n-1)} & & \dots & 1 & 0_{1\times(n-1)} & \\ 0 & 1 & 0_{1\times(n-2)} & \dots & 0 & 1 & 0_{1\times(n-2)} \\ & \vdots & & \ddots & & \vdots & \\ & 0_{1\times(n-1)} & 1 & \dots & & 0_{1\times(n-1)} & 1 \end{bmatrix}_{n\times(n^2)}$$

$$A_{col} = \begin{bmatrix} 1_{1\times n} & 0_{1\times n} & \dots & 0_{1\times n} \\ 0_{1\times n} & 1_{1\times n} & \dots & 0_{1\times n} \\ \vdots & \vdots & \ddots & \vdots \\ 0_{1\times n} & 0_{1\times n} & \dots & 1_{1\times n} \end{bmatrix}_{n\times(n^2)}$$

where n is the number or region-sector pairs.

For simplicity, suppose we have a 3×3 system that yields the first feed back loop selection of

$$S = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notice that S meets all of the criteria above. We can now trace out the flow of the loop. Starting at an arbitrary row, say the first row, we get to the second column. Moving the second row, we get to the first row. This is the first subloop of the feedback loop $1 \rightarrow 2 \rightarrow 1$.

The second subloop is the trivial $3 \rightarrow 3$. The function `feedback.loop()` identifies these subloops automatically for you.

Since some larger systems may take a while to compute, there is an option to compute the first n feedback loops. It also calculates the total value of intermediate transactions calculated for each loop. The output is a `FeedbackLoop` object which is a `list` over the vector of values and each loop. To see this observe the codebox below.

```
data(toy.IO)
class(toy.IO)

fbl = feedback.loop(toy.IO)
fbl$value

[1] 873 848 771 686 611 533 459 335 238 139
```

The subloops can be called by:

```
fbl$loop_1$subloop_2

  index                   RS
1     2         Hogwarts, Wii
2     7           Narnia, Wii
3     9   Narnia, Lightsabers
4     6         Narnia, Pizza
5     2         Hogwarts, Wii
```

Notice from `?feedback.loop` there is an option to aggregate sectors or regions.

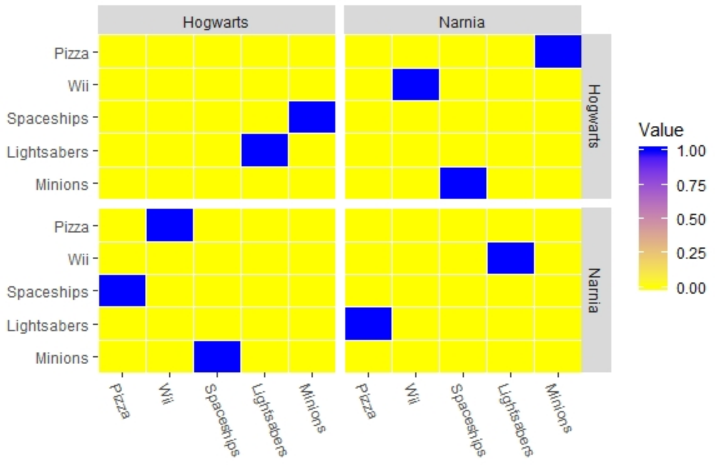
2.2.2 feedback.loop.matrix()

Input-output tables can be massive. For example, the World Input-Output Table from WIOD is over 2,400×2,400. This makes it infeasible to store 2,400 selector matrices each of which are are square matrices of the same dimension. To overcome this limitation, the subloops are stored instead. In order to obtain a specific loops selector matrix, you simply need to input your `FeedbackLoop` object into `feedback.loop.matrix()` along with the loop that you want, as demonstrated below.

```
data(toy.IO)
class(toy.IO)

fbl = feedback.loop(toy.IO)
fl_3 = feedback.loop.matrix(fbl, 3)
```

Figure 3: Visualizing Matrices



2.2.3 Feedback Loop Visualization

For either smaller input-output systems, or systems that have had either their regions or sectors aggregated by `feedback.loop()`, a heatmap via `heatmap.io()` can be produced to visualize which region-sector pairs are selected. This is demonstrated in Figure 3.

```
data(toy.IO)
class(toy.IO)

fbl = feedback.loop(toy.IO)
fl_1 = feedback.loop.matrix(fbl, 1)
heatmap.io(fl_1, RS_label = toy.IO$RS_label)
```

Furthermore, the total value of intermediate transactions identified by each loop can be plotted as follows:

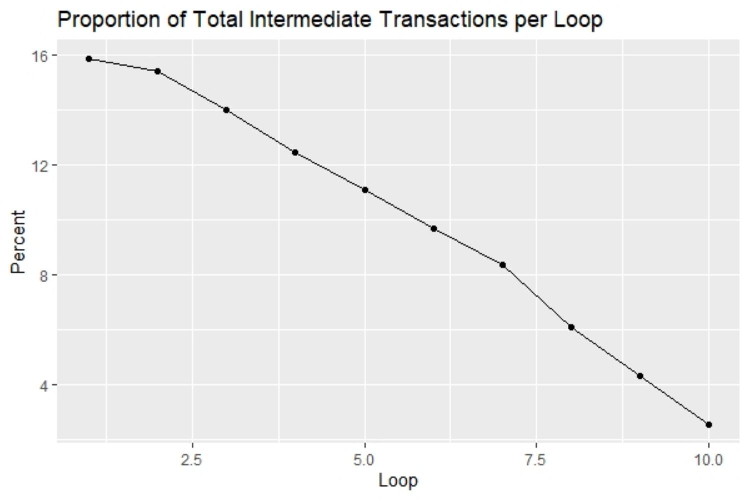
```
data(toy.IO)
class(toy.IO)

fbl = feedback.loop(toy.IO)
fbl$per = fbl$value / sum(fbl$value) * 100

obj = data.frame(x = 1:length(fbl$per), y = fbl$per)

ggplot(obj, aes(x = x, y = y)) +
  geom_line() + geom_point() +
  labs(x = 'Loop', y = 'Percent', title = 'Proportion of Total Intermediate
       Transactions per Loop')
```

Figure 4: VISUALIZING MATRICES



The output is shown in Figure 4. Note that figures generated with real data will more likely be convex than concave.

Lastly, the subloops can be plotted using network or graph theory via packages such as `ggnet2`, however, this package is not included with `ioanalysis`. The primary reason is that the diversity of input-output systems and specifications the user may want makes generalization unrealistic. As usual with educational materials, this is left as an exercise for the reader.

2.3 Helper Functions

The functions in this section are primarily intended to be used as helper functions used in the the primary functions. However, instead of being made hidden, they have been left available for the user to access them for other analysis. Because they are secondary functions, they do have less bells and whistles.

2.3.1 f.influence()

The field of influence is a recurisve function created by Sonis & Hewings (1992) that can be calculated up to the $n-1$ order where n is the number of region-sector combinations in the `InputOutput` object. This matrix is interpreted as the incremental to change to a specific region-sector combination. Typically, only the first order is used. Each recursion stores a new copy of the argument inputs on memory.

The first-order field of influence is calculated as follows:

$$F_1[i,j] = L_{.j}L_{i.}$$

and the k^{th} order field of influence is calculated by:

$$F_k[(i_1, ..., i_k), (j_1, ..., j_k)] = \frac{1}{k-1} \sum_{s=1}^{k} \sum_{r=1}^{k} (-1)^{s+r+1} l_{i_s, j_r} F_{k-1}[i_{-s}, j_{-r}]$$

where F is the field of influence, k is order of influence, l_{ij} is the i^{th} row and j^{th} column element of the Leontief Inverse and (with a slight abuse of notation) $-s$ indicates the s^{th} element has been removed.

2.3.2 f.influence.total()

This function uses the output from `f.influence()` to generate the total field of influence for the entire input-output system. In matrix notation, this is calculated as:

$$F^{total} = \sum_i \sum_j F_1[i, j]$$

where $F_1[i, j]$ is define in `f.influence()`.

2.3.3 export.coef()

`export.coef()` produces the export equivalent of the matrix of tehcnical coeficients (A). Additionally, there is another function called `import.coef()` that has the same idea, but for imports. This version is intended for use in the vertical specialization function. It requires that each region has the same sectors. This can be verified using `check.RS()`.

For `export.coef()`, the intution of what occurs in the background is as follows. If there are 5 sectors, then then end result is a 5×5 matrix of export coefficients. If each region in the A matrix constitutes a 5×5 block matrix, then it sums across the rows of these 5×5 block matrices for the region of interest, excluding the diagonal 5×5 block. The same intuition holds true for `import.coef()`, but instead sums across the columns.

2.3.4 export.total()

`export.total` (and `import.total()`) is simply the total value of exports for each region-sector combination, thus producing a vector of these values. It pulls these vales from the following matrices when applicable: matrix of intermediate transactions (Z), the export matrix (E) (or the import matrix (M)), and final demand (f) (or value added (V)). Specifically for the exports of a multi-region system, this function sums along the rows of sales to other regions within the intermediate transaction matrix (Z), plus the sales to other regions in the final demand matrix (f), and the exports (E).

3 Visualization

It can be hard to look at the numbers in a massive matrix and identify the important characteristics and trends. This section is aimed to ease this approach via visualization.

Visualization is useful for the researcher to identify key trends in an input-output system, to communicate these trends to an audience in presentations, and to show in publications to readers.

3.1 heatmap.io()

`heatmap.io()` can be used on any input-output matrix that correspond to an `RS_label`. The `RS_label` argument must be provided in order to correctly label and trim the plot. The plot is made utilizing the package `ggplot2`. It creates a heat map for sector interactions with on a grid of regions (see below). The default colors are chosen to be friendly to black and white printing, where darker indicates larger values. Additionally, there is an option to transform the elements in the matrix such as logarithm transformations.

Some input-output systems can be quite large, easily exceeding 100+ region-sector combinations. Naturally, this is too much to be seen visually in a heat map. `heatmap.io()` takes this into consideration with multiple ways to trim the image along the x-axis and the y-axis. One option is to specify the regions and/or sectors (numerically or by name) that the user wishes to display. This could look like row matrix of heat maps that region 1 sells to.

```
data(toy.IO)
RS_label = toy.IO$RS_label

fit = f.influence.total(toy.IO)
heatmap.io(fit, RS_label)

data(toy.ES)
ES2 = matrix(c(1,5,6,8,9))        # a way to
class(ES2) = 'EasySelect'         # trick the package
heatmap.io(fit, RS_label, ES_x = toy.ES, ES_y = ES2,
           low = '#00fcef', high = 'blueviolet')
```

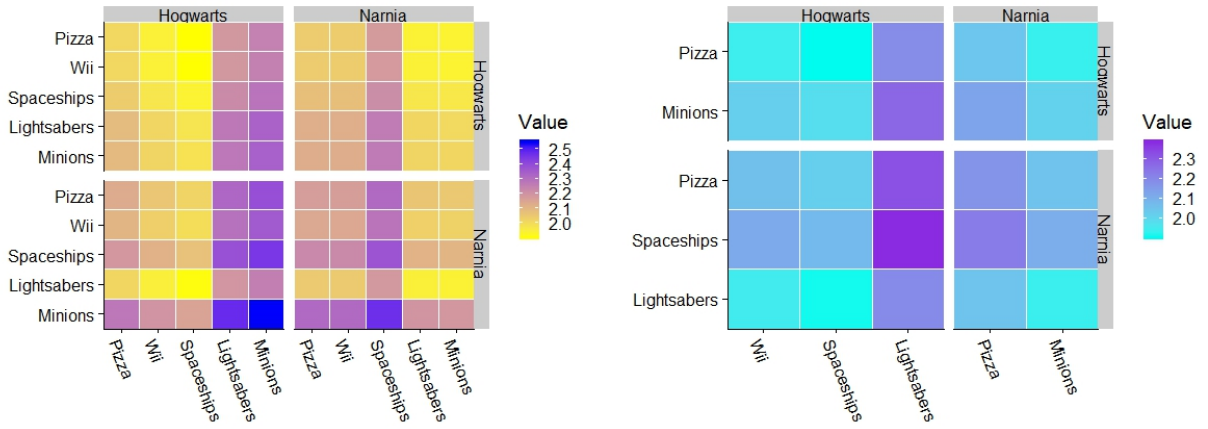
3.2 hist3d.io()

By popular demand, the package has an option to plot three dimensional histograms of `InputOutput` objects. These images are particularly useful when you want to communicate with your audience with three dimensional representations.

Since the taller bars of a 3D histogram can prevent viewing the smaller bars in the background, there are two options to adjust the representation of the histogram that can be used in combination. The first of which is rotating that image left-to-right and down-to-up. The second is to adjust the transparency of the bars.

Because of the ability to rotate the image, producing labels of regions and sectors is infeasible, particularly with indeterminate lengths of labels. Consequently, there is not an option to add labels. It is therefore suggested to use `hist3d.io` in conjunction with `heatmap.io`. Below is the code used to generate the image in Figure 6.

Figure 5: VISUALIZING MATRICES



```
data(toy.IO)
obj = toy.IO$Z[1:5, 1:5]

hist3d.io(obj, alpha = 0.7)
```

4 Impact Analysis

The following functions are primarily oriented around understanding how changes to the input-output system permeate. These changes can be due to changes over time, changes over regions, or hypothetical changes.

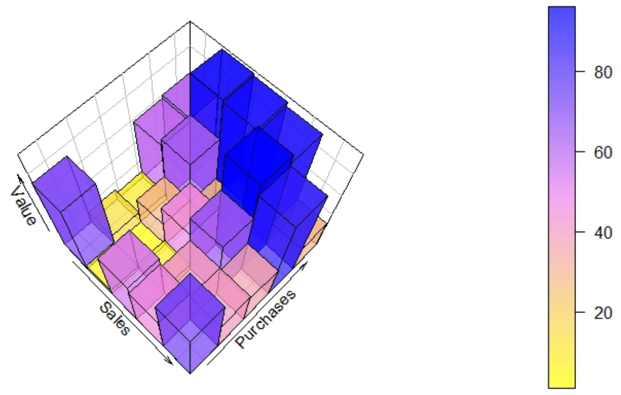
4.1 extraction()

The general idea behind hypothetical extraction is to determine what is the change to system's total production when a specific region-sector is removed (Dietzenbacher et al., 1993). This function has the option to evaluate these changes due to forward or backward `linkages`. Either you can use the raw value, which is equivalent to removing the demand of a region-sectors, or you can use the total extraction, which is removing the row and column corresponding to a specific region-sector. The output is a list over regions of a list over type of extraction.

The exact functions are calculated as follows:

1) `backward` uses the following where A_c is the matrix of technical input coefficients (A) with the j^{th} column replaced by zeros, X is the total production, and f is the final

Figure 6: 3D Visualizing Matrices



demand matrix aggregated across columns.

$$E^b = X - (I - A_c)^{-1} f$$

2) Similarly, forward is calculated where B_r is the matrix of technical output coefficients with the j^{th} row replaced by zeros, and V is the value added aggregated across rows.

$$E^f = X - \left(V(I - B_r)^{-1}\right)'$$

3) Then backward.total is calculated with A_{cr} as the matrix of technical input coefficients where the j^{th} row and column are replaced by zeros:

$$E^{b.tot} = X - (I - A_{cr})^{-1} f$$

4) Lastly, forward.total follows as:

$$E^{f.tot} = X - \left(V(I - B_{cr})^{-1}\right)'$$

Note that each hypothetical extraction requires the calculation of a variant of the Leontief inverse. Therefore, the same precautions about the computation duration apply to this function. This function can also be used with an EasySelect object. The default output is to produce the impact on each region-sector combination, however, there is an option to aggregate these values instead.

```
> E1 <- extraction(toy.IO, toy.ES)
> E1$Hogwarts
                          Wii  Spaceships  Lightsabers
Hogwarts.Pizza        299.3046   255.70141     312.4244
Hogwarts.Wii          792.3690   341.63601     321.4353
Hogwarts.Spaceships   899.5128  1241.01148     920.7511
Hogwarts.Lightsabers  462.7352   473.46025     837.1841
Hogwarts.Minions      121.5834   168.15738     155.3266
Narnia.Pizza          545.4602   501.61884     555.3375
Narnia.Wii            140.3173    99.75491     158.9805
Narnia.Spaceships     452.1373   460.11319     496.0949
Narnia.Lightsabers    549.6435   564.13726     539.2707
Narnia.Minions        250.7228   304.11064     238.5334

> E2 <- extraction(toy.IO, regions = c(1,2), sectors = c("Wii", "Minions"),
+                  type = c("backward", "backward.total"),
                   aggregate = TRUE)
> E2$Hogwarts$backward.total
          Wii   Minions
[1,] 4513.786 4359.234
```

4.2 output.decomposition()

This function is used when you have two `InputOutput` objects from different time periods (not necessarily consecutive time periods). `output.decomposition()` shows you how total output changes decomposed into two dimensions. The first is where the origin of the change is from ("`total`", "`internal`", or "`external`"). The second is the cause of the change ("`total`", "`finaldemand`", or "`leontief`").

There are several decomposition that can be made (Sonis et al., 1996). The following is first distinguished by cause, then by origin. A superscript of f indicates changes due to final demand, l indicates changes due to the Leontief inverse, and no superscript indicates total. A subscript of s indicates changes in output originating internally of the sectors, n indicates externally, and no subscript indicates total. L is the Leontief inverse and f is aggregated final demand. Analysis is over changes from period 1 to period 2. The values are calculated as follows:

1) Origin: `"total"`

$$\Delta X^f = L_1 \Delta f \qquad \Delta X^l = \Delta L f_1 \qquad \Delta X = \Delta L \Delta f$$

2) Origin: `"internal"`

$$\Delta X_s^f = diag(L_1)\Delta f \qquad \Delta X_s^l = diag(\Delta L) f_1 \qquad \Delta_s = diag(\Delta L)\Delta f$$

3) Origin: "external"

$$\Delta X_n^f = \Delta X^f - \Delta X_s^f \qquad \Delta X_n^l = \Delta X^l - \Delta X_s^l \qquad \Delta X_n = \Delta X - \Delta x_s$$

When calling upon `output.decomposition`, the default is to analyze of all types of decomposition. However, you can perform any combination of decompositions. The final result is a list over regions displaying the specific sectors decomposition. You can also explore the total effect by using the `sum()` function on a specific vector.

```
> output.decomposition(toy.IO, toy.IO2, origin = "external",
+                      cause = c("finaldemand","leontief"))
$Hogwarts
            ext.delta.X.f ext.delta.X.L
Pizza            12.81201    -204.54916
Wii              39.11599     -47.71853
Spaceships       35.97847     -32.44754
Lightsabers      37.56062    -123.05212
Minions          83.56893      62.89755

$Narnia
            ext.delta.X.f ext.delta.X.L
Pizza            59.55833   105.1533622
Wii              11.92300     3.6964222
Spaceships       41.31191    52.0130416
Lightsabers      53.03289    -0.6726956
Minions          37.08126   -36.3298116
```

4.3 mpm()

The multiplier product matrix is used to understand the structure of the input-output system (Nazara et al., 2003). This can either be used to compare two different systems, or it could be used to compare the same system across time. `mpm()` provides a visual interpretation in relation to backward and forward linkages as illustrated below. The mpm is calculated as:

$$M = \frac{1}{V}(L\mathbb{1})(\mathbb{1}'L) \qquad V = \mathbb{1}'L\mathbb{1}$$

Where L is the Leontief inverse, and $\mathbb{1}$ is an appropriately sized $n \times 1$ vector of ones. To use the function, simply enter `mpm(toy.IO)`

5 Updating

Updating is a technique–typically a recursive technique–to create an estimate for a new input-output matrix. A prominent feature of updating is that it will balance your matrix; that is, that the sum of estimated intermediate transactions (Z) plus the sum of final demand (f) is equal the total output (X). This is useful for when you want to forecast a future period's input-output matrix, if you want to rebalance a matrix using new estimates, or if you want to disaggregate a matrix.

The functions in this section are either iterative functions that calculate the new input-output matrix for you, or creates coefficients that are useful in the intermediate stages or for analysis themself.

5.1 ras()

The RAS algorithm is iterative, calculating a new matrix of technical input coefficeints (A) (Blair & Miller, 2009). To use this algorithm, you need estimates of the row and column sums of the matrix of intermediate transactions (Z). This can be obtained in one of two ways. The first way is to obtain values (estimates or actual) for the (Z) matrix through formal forecasting or surveys. Alternatively, if you have values for the total output vector (X), the final demand (f), and the value added, then you can back out the estimates for the row sums and column sums of (Z).

To illustrate this second approach, let Z be a $n \times n$ matrix, f be a $n \times m$ matrix, and V be a $p \times n$ matrix. Then we have

$$\hat{Z}_{row} = X' - \mathbb{1}_{1\times p}V$$
$$\hat{Z}_{col} = X - f\mathbb{1}_{m\times 1}$$

This matrix multiplication creates the correct dimensions and values to run the algorithm.

The language in the following is for the case of creating a forecasted input-output matrix, however, the technique is the same for other updating/balancing applications.

Define the following: A_t as the current period matrix of technical coefficients, A_{t+1} as the next periods matrix of technical coefficients, A^n as the nth step in of the A matrix's convergence, u_{t+1} the forecasted values for the ROW sums of Z, and v_{t+1} as the forecasted values for the COLUMN sums of Z. Then we also define $u^0 = A_t X_t$ as the starting estimate for the row sums, $R^n = diag(u_{t+1}/u^{n-1})$, $v^0 = X'R^1A_t$, and $S^n = diag(v_{t+1}/v^{n-1})$. Then the iterative process proceeds as follows:

$$\hat{A}^1_{t+1} = R^1 A_t \tag{Step 1}$$
$$\hat{A}^2_{t+1} = \hat{A}^1_{t+1}S^1 = R^1 A_t S^1 \tag{Step 2}$$
$$\hat{A}^3_{t+1} = R^2 R^1 A_t S^1 \tag{Step 3}$$
$$\hat{A}^4_{t+1} = R^2 R^1 A_t S^1 S^2 \tag{Step 4}$$
$$\lim_{n\to\infty} \hat{A}^{2n}_{t+1} = \lim_{n\to\infty} \left[R^n ... R^1\right] A_t [S^1 ... S^n] = A_{t+1} \tag{Step ∞}$$

Typically, this process should computationally converge within 50 iterations.

The following example assumes there is growth across all region-sectors in the input-output system. Note that this is a small system, so the computational speeds are fast. If your system is much larger, there as an option `verbose = TRUE` that prints the iteration it is on. There is also an option `type` that determines the norm used to evaluate convergence from the `norm()` function. Here the tolerance for convergence is left at the default value of 0.001.

```
set.seed(117)
growth <- 1 + 0.1 * runif(10)
sort(growth)

X <- toy.IO$X
X1 <- X * growth
U <- rowSums(toy.IO$Z)
U1 <- U * growth
V <- colSums(toy.IO$Z)
V1 <- V * growth

> Ahat <- ras(toy.IO, X1, U1, V1, maxiter = 10, verbose = TRUE)
   Iteration: 1    norm: 0.0695049251775855
   Iteration: 2    norm: 0.00173269421875299
   Iteration: 3    norm: 0.000120598694459476
```

5.2 lq()

The location quotient produces a forecast for the matrix of technical input coefficients (Blair & Miller, 2009). It uses the simple location quotient formula defined as

$$\begin{aligned}\hat{A}_{t+1} &= A_t LQ \\ LQ &= diag(lq_i) \\ lq_i &= \frac{X_i^r/X^r}{X_i^n/X^n}\end{aligned} \tag{4.2.1}$$

where X^n is the total production, X^r is the total production for region r, X_i^r is the production for region r sector i, and X_i^n is the total production for the i^{th} sector.

From equation (4.2.1) we can decompose calculation to build the interpretation. First notice that lq_i is the location quotient for a specific sector in a specific region. The numerator is the ratio of output in a specific sector in a region, over the total output for the specific region. Notice this number must be between zero and one. Similarly, the denominator is the ratio of output of that same sector across regions over the total output of the whole system resulting in a number between zero and one. If $lq_i > 1$ then we know that in this region, this sector produces relatively more compared to the systems average. However, in the function itself if the $lq_i \geq 1$, then it is set to $lq_i = 1$. This is to ensure every value in the matrix of technical input coefficients is less than 1; that is, preventing predictions that region-sectors produce more than total output.

5.3 rsp()

The regional supply percentage (RSP) works similarly to the location quotient updating, but focuses on imports and exports (Nazara et al., 2003). The RSP is calculated as follows:

$$\begin{aligned}\hat{A}_{t+1} &= \hat{p}A_t \\ \hat{p} &= diag(p_i) \\ p_i &= \frac{X_i - E_i}{X_i - E_i + M_i}\end{aligned}$$

where X_i is total output for region-sector i, E_i is exports, and M_i is imports. Notice that $p_i \in [0, 1] \ \forall \ i$.

6 Disaggregating Input-Ouput Systems

6.1 disaggregate()

While this function will not create a perfect disaggregated input-output table (seen in the details below), it will provide an estimate of a disaggregated system. The function follows the methodology outlined in (Blair & Miller, 2009). It also only disaggregates national tables. See `agg.region()` if your system has more than one region.

The function only requires three pieces of information to create the new system. The first of which is a disaggregated total production vector for all of the new regions. The final two pieces of information are for the `RAS` procedure: the estimate row sums and columns sums of the disaggregated intermediate transaction matrix `Z`. These two elements can be obtained in multiple ways, such as population weighted distribution of the original aggregated `Z` values.

The first step of the procedure is calculate the following matrix for each new region (r)

$$\begin{bmatrix} A^{rr} & A^{r\tilde{r}} \\ A^{\tilde{r}r} & A^{\tilde{r}\tilde{r}} \end{bmatrix}$$

such that

$$a_{ij}^{rr} = \begin{cases} LQ_i^r a_{ij}^n & \text{if } LQ_i^r < 1 \\ a_{ij}^n & \text{if } LQ_i^r \geq 1 \end{cases} \qquad a_{ij}^{\tilde{r}\tilde{r}} = \begin{cases} LQ_i^{\tilde{r}} a_{ij}^n & \text{if } LQ_i^{\tilde{r}} < 1 \\ a_{ij}^n & \text{if } LQ_i^{\tilde{r}} \geq 1 \end{cases}$$

$$A^{\tilde{r}r} = A^N - A^{rr} \qquad A^{r\tilde{r}} = A^N - A^{\tilde{r}\tilde{r}}$$

where r is the region of interest, $\tilde{r}$ is "the rest of the economy", and A^N is the original matrix of technical input coefficients. All location quotients (LQ) are calculated for the user as outlined in `lq()`.

Next the algorithm calculates

$$\begin{bmatrix} Z^{rr} & Z^{r\tilde{r}} \\ Z^{\tilde{r}r} & Z^{\tilde{r}\tilde{r}} \end{bmatrix} = \begin{bmatrix} A^{rr} & A^{r\tilde{r}} \\ A^{\tilde{r}r} & A^{\tilde{r}\tilde{r}} \end{bmatrix} \begin{bmatrix} \hat{X}^{r} & 0 \\ 0 & \hat{X}^{\tilde{r}} \end{bmatrix} = \begin{bmatrix} A^{rr}\hat{X}^{r} & A^{r\tilde{r}}\hat{X}^{\tilde{r}} \\ A^{\tilde{r}r}\hat{X}^{r} & A^{\tilde{r}\tilde{r}}\hat{X}^{\tilde{r}} \end{bmatrix}$$

which allows for creation of the block matrix

$$\begin{bmatrix} Z^{11} & \blacksquare & \cdots & \blacksquare & Z^{1\tilde{1}} \\ \blacksquare & Z^{22} & \cdots & \blacksquare & Z^{2\tilde{2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \blacksquare & \blacksquare & \cdots & Z^{RR} & Z^{R\tilde{R}} \\ Z^{\tilde{1}1} & Z^{\tilde{2}2} & \cdots & Z^{\tilde{R}R} & \blacksquare \end{bmatrix}_{[s(R+1)]\times[s(R+1)]}$$

where s is the number of sectors. Finally, each column of $\blacksquare$ matrices is filled by distributing $\frac{1}{2(R-1)}[Z^{\tilde{r}r} + Z^{r\tilde{r}}]$ appropriately. Note that Blair & Miller (2009) recommends instead to fill with $\frac{1}{R-1}[Z^{\tilde{r}r}]$, however, simulations with the former method has proved to be more reliable.

Next the following matrix is created to then be put into the `RAS` algorithm

$$\begin{bmatrix} \tilde{A}^{11} & \cdots & \tilde{A}^{1R} \\ \vdots & \ddots & \vdots \\ \tilde{A}^{R1} & \cdots & \tilde{A}^{RR} \end{bmatrix} = \begin{bmatrix} Z^{11} & \cdots & \frac{1}{2(R-1)}[Z^{\tilde{1}1} + Z^{R\tilde{R}}] \\ \vdots & \ddots & \vdots \\ \frac{1}{2(R-1)}[Z^{\tilde{R}R} + Z^{1\tilde{1}}] & \cdots & Z^{RR} \end{bmatrix} \hat{X}^{-1}$$

To actually implement the algorithm in the package, see the following code. It is important to ensure that the elements of `X`, `U`, and `V` are in a list format. In the example below, one method of calculating weights is created for the user.

```
data(toy.IO)
X = list(toy.IO$X[1:5], toy.IO$X[6:10])

io = agg.region(toy.IO, regions = c(1,2), newname = 'Magic')

V <- U <- vector('list', 2)
w = list(sum(X[[1]]), sum(X[[2]])) # weights
for(i in 1:2){
  U[[i]] = rowSums(io$Z) * w[[i]]/sum(unlist(w))
  V[[i]] = colSums(io$Z) * w[[i]]/sum(unlist(w))
}

A.new = disaggregate(io, X, U, V,
                     new.regions = unique(toy.IO$RS_label[,1]))

Z.new = A.new %*% diag(c(unlist(X)))

io.new = as.inputoutput(Z = Z.new, X = unlist(X),
                        RS_label = toy.IO$RS_label)
```

References

Antràs, P., Chor, D., Fally, T., & Hillberry, R. (2012). Measuring the upstreamness of production and trade flows. *American Economic Review*, *102*(3), 412–16.

Blair, P. D., & Miller, R. E. (2009). *Input output analysis: Foundations and extensions.* New York: Cambridge University Press.

Dietzenbacher, E., van der Linden, J. A., & Steenage, A. E. (1993). The regional extraction method: Ec input-output comparisons. *Economic Systems Research*, *5*(2).

Hummels, D., Ishii, J., & Yi, K.-M. (2001). The nature and growth of vertical specialization in world trade. *Journal of International Economics*, *54*(1), 75–96.

Nazara, S., Guo, D., Hewings, G. J. D., & Dridi, C. (2003). Pyio: Input-output analysis with python. *REAL Discussion Paper 03-t-23. University of Illinois at Urbana-Champaign.*

Sonis, M., Hewings, G. J., & Gazel, R. (1995). The structure of multi-regional trade flows: hierarchy, feedbacks and spatial linkages. *The Annals of Regional Science*, *29*(4), 409–430.

Sonis, M., & Hewings, G. J. D. (1992). Coefficient change in input-output models: Theory and applications. *Economic Systems Research*, *4*(2), 143-158.

Sonis, M., Hewings, G. J. D., & Guo, J. (1996). Sources of structural change in input-output systems: A field of influence approach. *Economic Systems Research*, *8*(1), 15-32.

A Input Output Tables

The package is based around input-output tables that are based in a monetary units. However, if you have a system that has matrices or columns of matrices that are in non-monetary units (such as kg), then the package still performs the operations in the function; the package cannot tell the difference. Albeit, it is up to the user to correctly interpret the output from the functions. This package is based around an input-output system as follows:

INPUT-OUTPUT TABLE

			Purchases				Final Demand				Total
	Region	Sector	R_1 S_1	R_1 S_2	R_2 S_1	R_2 S_2	R_1 Gov	R_1 Inv	R_2 Gov	R_2 Inv	
Sales	R_1	S_1	$Z_{n\times n}$				$f_{n\times m}$				$X_{n\times 1}$
	R_1	S_2									
	R_2	S_1									
	R_2	S_2									
VA	Wages tax		$V_{p\times n}$				$fV_{p\times m}$				
	Total		$X'_{1\times n}$								

To see an example within the package that corresponds to `toy.IO`, enter the following commands:

```
data(toy.FullIOTable)
View(toy.FullIOTable)
```

There are several matrices that are part of an `InputOutput` object. The first matrices discussed are those that are standard in an input-output system, then the tables that require calculation are discussed.

Z The intermediate transaction matrix is the raw value of goods and services sold across the region-sectors. The package only accepts a square Z matrix. Consider the aluminum industry in America (AA). If looking at the row labeled AA, we see across the columns the region-sectors that purchase aluminum from America. This could be, for example, automotive firms in America or Canadian aerospace industries. Then if we look at the column labeled AA, we see what AA purchases as inputs. This could be everything from aluminum ore to electricity from various regions.

f The final demand matrix is a disaggregated version of the national accounting equation. If you input-output table has a column for net exports (i.e. there is no distinction between imports and exports) it is recommended to leave the net exports column in the f matrix. The interpretation of the elements proceeds in the same manor as the Z matrix.

X The total output matrix is simply the total value of goods produced by each region sector combination. This is not the same thing as the contribution to GDP by each region sector combination, because it includes intermediate goods.

V The value added matrix is the additional expenses that were used to produce the goods for that region-sector. Common features are employee compensation and taxes paid. Notice that there is no distinction between the regions in the value added matrix. If you want to make this distinction for say labor, it is recommended to close the system to employment by adding columns and rows for labor by each region.

fV The value added of final demand is interpreted similarly to V, but it is instead the value added for final demand. This matrix is oftentimes not included in an input-output system.

A The technical input coefficients matrix is calculated using the intermediate transactions matrix (Z) and the total output matrix (X) by

$$a_{ij} = \frac{z_{ij}}{x_j}$$

or in matrix notation

$$A = Z \times diag\left(\frac{1}{X}\right)$$

Here the i^{th} row j^{th} column element a_{ij} is interpreted as the ratio of *input* i used to produce j to the total value of i produced.

B The technical output coefficient matrix is similar to A, but is the calculated as

$$b_{ij} = \frac{z_{ij}}{x_i}$$

or in matrix notaion

$$B = diag\left(\frac{1}{X}\right) \times Z$$

L The Leontief inverse is calculated as $L = (I - A)^{-1}$ from the following derivation:

$$\begin{aligned} X &= Z + f \\ IX &= AX + f \\ (I - A)X &= f \\ X &= (I - A)^{-1} f \\ &= Lf \end{aligned}$$

Notice that we have $l_{ij} = \frac{\partial x_i}{\partial f_j}$; that is, l_{ij} is the change in total output given a change in final demand.

G The Ghoshian inverse is define as $G = (I - B)^{-1}$ derived in the same manner as L, but recall $Z = XB$.

E, M The export and import matrices are to distinguish these values for the input-output. If you system makes this distinction, then the exports (E) are typically stored in the f matrix and imports (M) are typically stored in the V matrix. Separating these values is particularly useful, for example, in the `upstream()` function.

B Version Changes and Updates

Below is a list of changes made to the package `ioanalysis` throughout the lifetime package of the package. The first digit is reserved for major or significant package changes, the second digit is reserved for adding functions to the package, and the third digit is reserved for fixing bugs within each function.

If you experience any issues, bugs, or errors, then please contact the package maintainer found in the `DESCRIPTION` file located by using the command `help(package = 'ioanalysis')`.

B.1 Version 0.1.1

Minor bug fixes to issues that prevented use of unbalanced region-sector input-output systems.

B.2 Version 0.1.2

Adjusted the input for `f.influence()`.

B.3 Version 0.1.3

Corrected functional forms within a few functions.

B.4 Version 0.2.1

By popular request, the total field of influence was added to the package. Additionally, a versatile visualization tool has been added. This function allows for heat maps of any matrix in the input-output system that can be trimmed by the user. The intent is to be used in a presentations and papers (with default colors that are black and white friendly).

B.5 Version 0.2.2

The new version of R (4.0) allows for `matrix` class objects to have multiple class types such as `matrix` and `array`. Additionally, since the main object of the package `InputOutput` is at its heart a `list` object it can also have multiple class types.

The checks within each function assumed there was only one class type per object, which produces an warning of multiple inputs for each `if` statement since there is the potential for multiple `TRUE` or `FALSE` inputs. An example of the lines of code that were changed is changing statements from `if{class(io) != 'InputOutput'}` to `if{!'InputOutput' %in% class(io)}`.

B.6 Version 0.3.0

Added hierarchical feedback loop analysis. A feedback loop traces out a loop of region-sector combinations following the Linear Programming Assignment problem. The output produces a selector matrix with a sudoku-like constraint, such that each row only has one column selected, and each column only has one row selected.

B.7 Version 0.3.1

`feedback.loop()` used to store all feedback loop matrices, which created issues for large input-output systems. `feedback.loop.matrix()` was created to reduce the demand on RAM by a factor of n.

B.8 Version 0.3.2

`feedback.loop()` output was altered to identify subloops within each loop. Accordingly, `feedback.loop.matrix()` was altered to produce selector matrices from the new subloop output.