faoswsAupus: A package replicating the logic of the AUPUS statistical working system

Joshua M. Browning, Michael. C. J. Kao
Food and Agriculture Organization
of the United Nations

Abstract

This vignette provides a detailed description of the usage of functions in the **faoswsAupus** package.

Keywords: AUPUS, Standardization.

1. Introduction

First, let us give an overview of the procedure for creating the food balance sheets. The data we collect for food balance sheets is usually at the primary level (e.g. wheat, milk, etc.). However, there are several exceptions: first, some countries may report production of non-primary products (flour, beer, etc.) but this is not commonly the case. Additionally, trade information is provided in great detail: we may know how much bread went from country A to country B. We must account for these trade imbalances within the commodity balances (for example, we must consider the trade imbalance of bread in the wheat commodity tree). One approach would be to simply roll-up all trade imbalances into the top level equivalent (i.e. wheat, in this example) but there are further complications. Processing of wheat into flour creates bran and germ, and these products are almost entirely put into feed utilization.

Thus, we take the following approach in creating the food balance sheets.

1. We start with the "primary" commodities: wheat, barley, milk, etc.
   (a) If some elements of the balance are missing (i.e. our modules have failed to impute them), then impute these values using the old AUPUS methodology (as a ratio of production, total supply, etc.).
   (b) Now, we must balance at this level. Balancing is done by maximum likelihood or something similar, and so measurement error is allocated to all products (with more error allocated to products with larger estimated variances).
2. Now, the amount allocated to food for the primary commodity is to be converted into production at the first processed level (flour, butter, etc.) if the values at that level are missing (otherwise the official data is used). Note: if official data is available at this level, this should inform the shares or the food from the primary commodity but should not be used to compute the extraction rate. **How does this drive what we do on the primary level? If this is the only child, then we just update the food for the primary. But, if there are multiple children (some without official data) then does this knowledge inform an adjustment to the food or the shares or both?**
   (a) First, some of the primary commodity may be eaten as such, and hence the percent

going to processing should be determined and a quantity then removed from food going to processing.

(b) Next, the shares (which specifies how the processed commodity should be allocated to all of it's available children) is applied to allocate the amount processed into it's various children.

(c) Lastly, the extraction rate (which specifies a sort of conversion rate from a parent commodity to it's child) should be applied to convert the processed parent commodity into quantity of the child.

For example, suppose we want to create flour production from wheat. We may have 90% of wheat being processed (and 10% left as such), 95% of processed wheat that is allocated to flour and 5% allocated to beer, and an 80% extraction rate of wheat to flour. Then, if we had 100 thousand tons of wheat, we would convert this to 100(90%)(95%)(80%) = 68.4 thousand tons of flour.

3. In the above step, we must also be careful to create all appropriate elements as specified by the tree structure. For example, bran and germ are also created when wheat is processed into flour. These are not elements with separate shares but should be thought of as by-products in the creation of flour.

4. We continue steps 1-3 and process commodities down the commodity tree and balance at each step until we reach a point where no further processing is required. For example, wheat must be processed down into flour so we can create bran and germ (as these elements often get allocated to feed). However, it does not need to be processed into bread, as no important information is gained in that process. Another interesting example is the barley tree: barley must be processed into barley malt and then barley beer, and this processing must occur because alcoholic products are placed in a different group in the standardization (i.e. aggregation) of the commodity tree. Note that as we process down, the production of the processed commodities must be fixed in order to ensure the parent commodities remain balanced. When balancing a "terminal node" (i.e. a node whose food value will not go into a processed commodity) we must standardize all lower commodities back to this node. This ensures the entire tree is balanced down to the terminal node. As an example, consider the wheat tree. First, we balance wheat. Then, food from the wheat balance becomes production of flour (after conversion by shares, extraction rates, and processing rate). We also standardize all traded children commodities (bread, biscuits, etc.) into flour and then balance flour.

5. We can now compute nutrient information (calories, proteins, fats) from quantities.

6. Now, we must standardize everything back up to it's primary equivalent. We start at the lowest nodes and divide by extraction rates to compute parent quantites. Calories, on the other hand, can be added directly in the standardization process. **Is this right???** However, there are several special cases/important notes:

   - With "by-products" (for example, wheat bran and germ) we do not standardize quantities as they are already accounted for in the main product standardization. However, standardization of calories/fats/proteins is performed for all products by adding the calorie/fat/protein values.
   - Some products (oils, juices, beers, etc.) can be created from multiple parents. In this case, the products must be rolled up into various parents, and the appropriate allocation to parents is not clear. We may use shares to determine this allocation, but we could have problems if a country has a large trade deficit in a child and little availability in a parent (or even larger problems if default shares are used and

a country does not actually produce or trade a parent). Thus, allocation should be generally done based on availability. However, in some cases we need to be able to specify that preference be given to certain parents. An example of this could be beer where preference should be given to barley over, say, bananas, wheat, etc.

- Separate trees may be used for processing vs. standardizing, as some commodities do not roll up into their respective parents (e.g. beer is not in cereals and butter is not in milk).
- Production should **not** be standardized. This is because production of children commodities come directly from food of parent commodities, and so essentially they are already accounted for. All other elements should be standardized, though. Alternatively, we could standardize production if we deduct input from processing values, but that seems to add complication to the standardization procedure and it is not clear if this will improve our estimates.

2. Example

Consider a very simple example of the wheat tree. In this example, we would need distributions to perform the balances, but that is ignored for this example and balances are simply done arbitrarily to avoid complication. Also, we assume there is only production, imports, exports, food and waste. Dashes indicate unavailable data.

	Prod.	Imp.	Exp.	Food	Waste
Wheat	90	20	10	100	5
Flour	-	30	5	-	0
Biscuits	-	0	10	-	0
Bread	-	0	10	-	0

- Balance Wheat:

	Prod.	Imp.	Exp.	Food	Waste
Wheat	90	20	10	**98**	**2**
Flour	-	30	5	-	0
Biscuits	-	0	10	-	0
Bread	-	0	10	-	0

- Process to flour:

	Prod.	Imp.	Exp.	Food	Waste
Wheat	90	20	10	98	2
Flour	**82**	30	5	-	0
Biscuits	-	0	10	-	0
Bread	-	0	10	-	0

- Create by-products (skipped for simplicity)
- Standardize bread and biscuits (using extraction rates, not shown here):

	Prod.	Imp.	Exp.	Food	Waste
Wheat	90	20	10	98	2
Flour	82	30	**35**	-	0
Biscuits	**10**	0	10	**0**	0
Bread	**10**	0	10	**0**	0

- Balance flour:

	Prod.	Imp.	Exp.	Food	Waste
Wheat	90	20	10	98	2
Flour	82	30	35	**77**	0
Biscuits	10	0	10	0	0
Bread	10	0	10	0	0

- Standardize to wheat:

	Prod.	Imp.	Exp.	Food	Waste
Wheat	90	**56**	**52**	**~~9892~~**	2
Flour	82	30	35	77	0
Biscuits	10	0	10	0	0
Bread	10	0	10	0	0

  Note that in this case, we have to overwrite the food for wheat because it's not really food (in the sense that it's not actually consumed here). The food value here is more of an input to food processing, and so maybe we should add that element. Then, when we standardize, we'll remove the input to food processing elements and only keep the (standardized) food values.

3. Data Initialization

First, we need to load the relevant packages:

```
library(faoswsAupus)
library(faosws)
library(igraph)
library(data.table)
```

This package comes with many functions for replicating the AUPUS and standardization procedures as well as several useful datasets. The first we will use is called 'US', and it contains an example of the data that is used within this package. There is also a variable called usAupusParam, and that variable contains information about the US dataset.

```
is(US)

## [1] "list"   "vector"

names(US)

## [1] "aupusData"          "inputData"          "ratioData"
## [4] "shareData"          "balanceElementData" "itemInfoData"
## [7] "populationData"     "extractionRateData"

sapply(US, class)

##      aupusData    inputData    ratioData    shareData    balanceElementData
## [1,] "data.table" "data.table" "data.table" "data.table" "data.table"
## [2,] "data.frame" "data.frame" "data.frame" "data.frame" "data.frame"
##      itemInfoData populationData extractionRateData
## [1,] "data.table" "data.table"   "data.table"
## [2,] "data.frame" "data.frame"   "data.frame"
```

```
sapply(US, dim)

##      aupusData inputData ratioData shareData balanceElementData itemInfoData
## [1,]         8         7         8         2                   8            8
## [2,]        75         6        39         5                   4            3
##      populationData extractionRateData
## [1,]              1                  8
## [2,]              6                  5

is(usAupusParam)

## [1] "list"   "vector"

names(usAupusParam)

## [1] "areaCode"    "itemCode"    "elementCode" "year"        "keyNames"
```

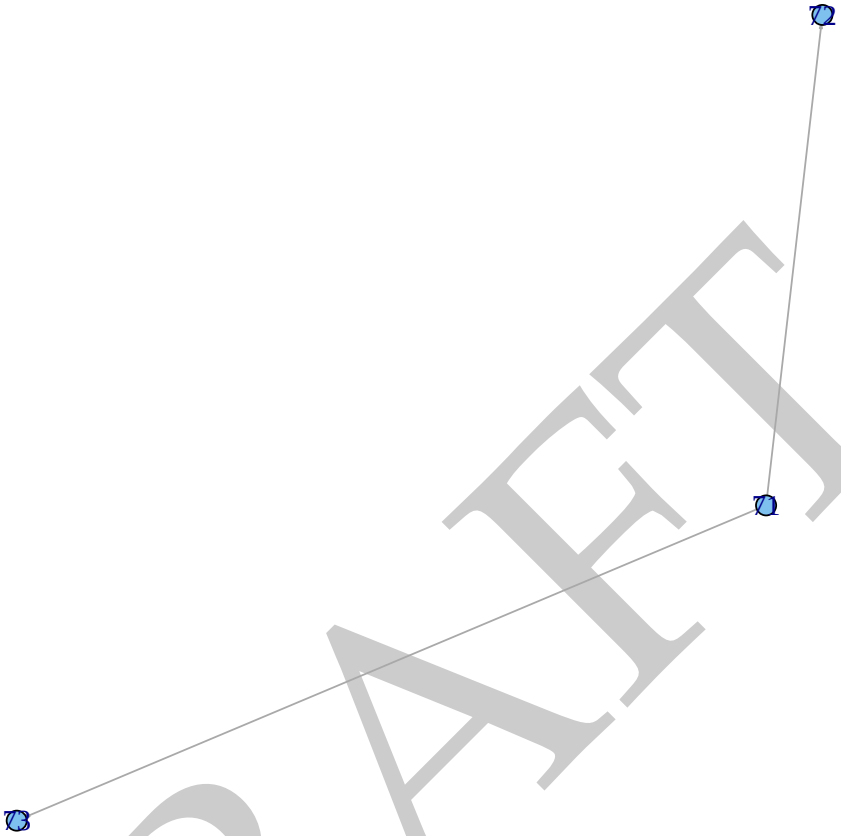
Note: usually these values would be generated in the following manner:

```
GetTestEnvironment(
    baseUrl = "https://hqlprswsas1.hq.un.fao.org:8181/sws",
    token = "34c18199-4f48-4ce4-a8d5-16ce62e8e23d")
usAupusParam = getAupusParameter(areaCode = "231", assignGlobal = FALSE,
                                 yearsToUse = 2009:2013)
US = getAupusDataset(aupusParam = usAupusParam)
```

The GetTestEnvironment function from the faosws package sets up the right variables in R for querying the SWS database, and the specific token I used provides the necessary information for the AUPUS dataset. The parameters and data are then pulled from the SWS. However, for this vignette, we'll just use the data that already exists in this package.

We see that the US dataset contains 8 data.tables. Let's look at a subset of this data to more easily understand what we're working with. First, let's ensure we grab a meaningful subset of data. The plotCommodityTree function takes the shareData dataset and generates a plot for how the commodities are related to one another.

```
plotCommodityTree(US$shareData)
```



For simplicity, let's look at only commodities 71, 72, and 73.

```
US = subsetAupus(aupusData = US, parentKeys = c(71, 72, 73),
                 aupusParam = usAupusParam)
```

Now, we wish to work with this data in a network framework, and the package contains a few functions to generate that framework:

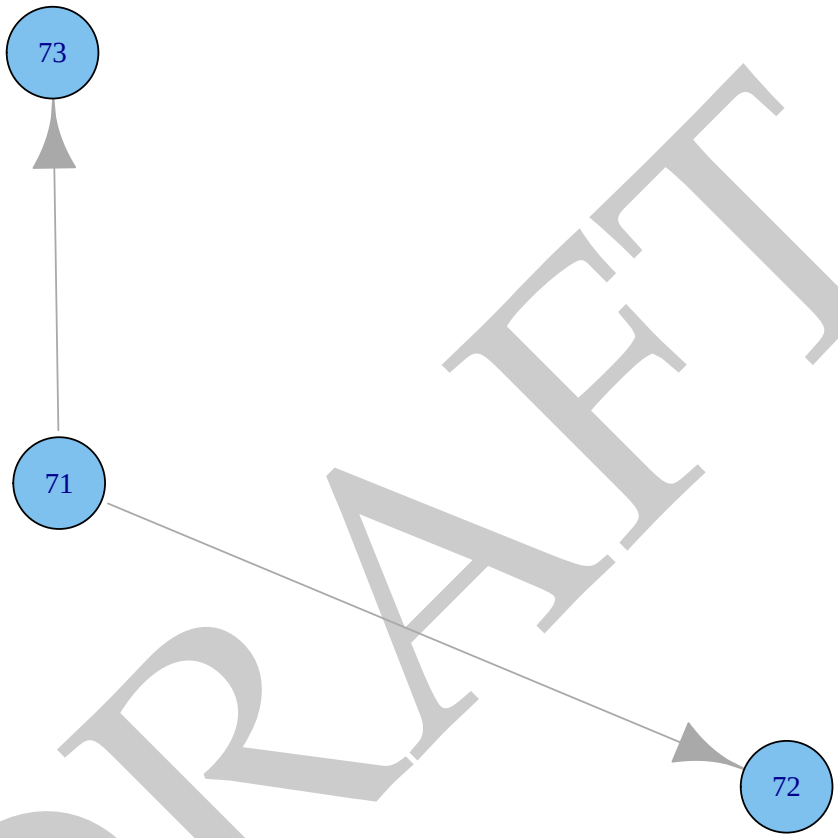
```
aupusNetwork = suaToNetworkRepresentation(dataList = US,
                                          aupusParam = usAupusParam)
names(aupusNetwork)

## [1] "nodes" "edges"

sapply(aupusNetwork, class)

##       nodes         edges
```

```
## [1,] "data.table" "data.table"
## [2,] "data.frame" "data.frame"

sapply(aupusNetwork, dim)

##      nodes edges
## [1,]     8     7
## [2,]   116     9
```

We see that now the data has been condensed down into two objects: nodes and edges. The nodes dataset has 15 rows (5 years times 3 commodities). There are only 10 rows in the edges dataset, as there are only 2 edges times 5 years. The nodes dataset is essentially the merged aupusData, itemInfoData, ratioData, balanceElementData, and populationData from US, while the edges dataset contains the datasets shareData, extractionRateData, and inputData from US. Here's our reduced network visualization:

```
plotCommodityTree(US$shareData, edge.arrow.size = 2, vertex.size = 25)
aupusNetwork$nodes[, .(geographicAreaFS, timePointYearsSP, measuredItemFS)]

##    geographicAreaFS timePointYearsSP measuredItemFS
## 1:              231             2009             41
## 2:              231             2009             71
## 3:              231             2009             72
## 4:              231             2009             73
## 5:              231             2009            113
## 6:              231             2009            632
## 7:              231             2009            634
## 8:              231             2009            654

colnames(aupusNetwork$nodes)[1:10]

##  [1] "geographicAreaFS"                "timePointYearsSP"
##  [3] "measuredItemFS"                  "Value_measuredElementFS_11"
##  [5] "flagFaostat_measuredElementFS_11" "Value_measuredElementFS_21"
##  [7] "flagFaostat_measuredElementFS_21" "Value_measuredElementFS_31"
##  [9] "flagFaostat_measuredElementFS_31" "Value_measuredElementFS_41"

aupusNetwork$edges[, .(geographicAreaFS, timePointYearsSP,
                       measuredItemParentFS, measuredItemChildFS)]

##    geographicAreaFS timePointYearsSP measuredItemParentFS measuredItemChildFS
## 1:              231             2009                   71                  41
## 2:              231             2009                   71                  72
## 3:              231             2009                   71                  73
## 4:              231             2009                   71                 113
## 5:              231             2009                   71                 632
## 6:              231             2009                   71                 634
## 7:              231             2009                   71                 654

colnames(aupusNetwork$edges)
```

```
## [1] "measuredItemParentFS"   "measuredItemChildFS"    "timePointYearsSP"
## [4] "geographicAreaFS"       "Value_input"            "flagFaostat_input"
## [7] "Value_extraction"       "flagFaostat_extraction" "Value_share"
```



4. AUPUS

The entire AUPUS procedure is encapsulated in one function: Aupus. However, let's look at some of the functions called within this main function to understand how the process works. First, we have to set up some of the things as done by the Aupus function:

```
nodes = aupusNetwork$nodes
edges = aupusNetwork$edges
nodes = coerceColumnTypes(aupusParam = usAupusParam, data = nodes)
edges = coerceColumnTypes(aupusParam = usAupusParam, data = edges)
from = usAupusParam$keyNames$itemParentName
to = usAupusParam$keyNames$itemChildName
```

```
processingLevelData = edges[, findProcessingLevel(.SD, from = from,
    to = to, aupusParam = usAupusParam),
    by = c(usAupusParam$keyNames$areaName, usAupusParam$keyNames$yearName)]
setkeyv(processingLevelData, key(nodes))
invisible(nodes[processingLevelData, `:=`(processingLevel, i.processingLevel)])
invisible(nodes[is.na(processingLevel), processingLevel := 0])
nodes[, c(key(nodes), "processingLevel"), with = FALSE]

##    geographicAreaFS measuredItemFS timePointYearsSP processingLevel
## 1:              231             41             2009               1
## 2:              231             71             2009               0
## 3:              231             72             2009               1
## 4:              231             73             2009               1
## 5:              231            113             2009               1
## 6:              231            632             2009               1
## 7:              231            634             2009               1
## 8:              231            654             2009               1
```

The processing level function above uses functions from the **igraph** package to determine the "processing level." This value indicates the order in which a particular node is processed: nodes at level 0 have no inputs/dependencies/children, and thus they can be processed immediately. Once level 0 nodes have been processed, we have the data necessary for processing their parents, and these are nodes with processing level 1. Aggregation continues until all processing levels have been processed.

Our example is simple: 72 and 73 are children of 71, and so 71 must first be processed before we process 72 and 73. Thus, 71 is processingLevel = 0 and 72 and 73 have processingLevel = 1.

At each level, there are three main processes that are performed:

1. The main AUPUS module is ran on each node. This module computes each individual element following the logic of the old system.
2. The edges of the graph are updated.
3. The inputs from processing are updated.

4.1. Main AUPUS Module

The main function here is calculateAupusElements. This function calls all of the individual element calculation functions. Each element function has it's own calculation, and is documented within it's help page. However, we'll show an example for a few of the functions. First, though, note that we must subset the AUPUS data: we only want to process the commodities at the lowest processing level right now:

```
toProcess = nodes[processingLevel == 0, ]
```

Ok, now let's compute element 11:

```
toProcess[, Value_measuredElementFS_11]

## [1] 12000
```

```
toProcess[, Value_measuredElementFS_161]

## [1] NA

toProcess[, flagFaostat_measuredElementFS_11]

## [1] ""

calculateEle11(data = toProcess, aupusParam = usAupusParam)

## integer(0)

toProcess[, Value_measuredElementFS_11]

## [1] 12000

toProcess[, flagFaostat_measuredElementFS_11]

## [1] ""
```

Element 11 is called "Initial Existence" and thus it is set to "Final Existence" (element 161) from the previous year, if that value exists and if element 11 is currently missing. In this case, all of element 161's values are missing and thus no updating occurs. Let's continue processing some elements:

```
calculateEle21(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle41(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle51(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle314151(data = toProcess, aupusParam = usAupusParam)

## [[1]]
## integer(0)
##
## [[2]]
## integer(0)

calculateEle63(data = toProcess, aupusParam = usAupusParam)

## [[1]]
## [1] 1
##
## [[2]]
## integer(0)
```

```
calculateEle71(data = toProcess, aupusParam = usAupusParam)

## [[1]]
## integer(0)
##
## [[2]]
## integer(0)

calculateEle93(data = toProcess, aupusParam = usAupusParam)

## [[1]]
## [1] 1
##
## [[2]]
## integer(0)

calculateTotalSupply(data = toProcess, aupusParam = usAupusParam)
tail(colnames(toProcess))

## [1] "Value_balanceElement" "Value_population_11"  "Value_population_21"
## [4] "processingLevel"      "newSummation"         "TOTAL_SUPPLY"

toProcess$TOTAL_SUPPLY

## [1] 297711
```

Each of the calculateEleXX functions above calculates updated values for each element and returns a vector of list of vectors with indices. These indices represent the rows that were updated in the computation of this element. The last function, calculateTotalSupply, returns nothing but adds an additional column to the toProcess data.table. This column, TOTAL_SUPPLY, represents the total supply for this commodity. Now, we can proceed with computing other elements. Some elements, such as 101, 111, 121, 131, and 141 use total supply to fill in their values:

```
calculateEle101(stotal = "TOTAL_SUPPLY", data = toProcess,
                aupusParam = usAupusParam)

## integer(0)

calculateEle111(stotal = "TOTAL_SUPPLY", data = toProcess,
                aupusParam = usAupusParam)

## [[1]]
## integer(0)
##
## [[2]]
## integer(0)

calculateEle121(stotal = "TOTAL_SUPPLY", data = toProcess,
                aupusParam = usAupusParam)
```

```
## integer(0)

calculateEle131(stotal = "TOTAL_SUPPLY", data = toProcess,
                aupusParam = usAupusParam)

## integer(0)

calculateEle141(stotal = "TOTAL_SUPPLY", data = toProcess,
                aupusParam = usAupusParam)

## integer(0)

calculateEle144(population11Num = "Value_population_11",
                data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle151(stotal = "TOTAL_SUPPLY", data = toProcess,
                aupusParam = usAupusParam)

## integer(0)

calculateEle161(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle171(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle174(population11Num = "Value_population_11",
                                  data = toProcess,
                                  aupusParam = usAupusParam)

## integer(0)
```

Once we've computed element 141, we can compute nutritive values: calories, proteins, and fats. These are based on ratios provided in the database (to convert quantity into these values).

```
calculateTotalNutritive(ratioNum = "Ratio_measuredElementFS_261",
                            elementNum = 261, data = toProcess,
                            aupusParam = usAupusParam)

## integer(0)

calculateDailyNutritive(population11Num = "Value_population_11",
                        population21Num = "Value_population_21",
                        dailyElement = 264, totalElement = 261,
                        data = toProcess, aupusParam = usAupusParam)
```

```
## integer(0)

calculateTotalNutritive(ratioNum = "Ratio_measuredElementFS_271",
                            elementNum = 271, data = toProcess,
                            aupusParam = usAupusParam)

## integer(0)

calculateDailyNutritive(population11Num = "Value_population_11",
                        population21Num = "Value_population_21",
                        dailyElement = 274, totalElement = 271,
                        data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateTotalNutritive(ratioNum = "Ratio_measuredElementFS_281",
                            elementNum = 281, data = toProcess,
                            aupusParam = usAupusParam)

## integer(0)

calculateDailyNutritive(population11Num = "Value_population_11",
                        population21Num = "Value_population_21",
                        dailyElement = 284, totalElement = 281,
                        data = toProcess, aupusParam = usAupusParam)

## integer(0)
```

We now need to calculate two remaining elements (541 and 546, containing final and total demand) and then we can balance.

```
calculateEle541(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateEle546(data = toProcess, aupusParam = usAupusParam)

## integer(0)

calculateTotalUtilization(data = toProcess, aupusParam = usAupusParam)
calculateBalance(supply = "TOTAL_SUPPLY", utilization = "TOTAL_UTILIZATION",
                 data = toProcess, aupusParam = usAupusParam)

## integer(0)

tail(colnames(toProcess))

## [1] "Value_measuredElementFS_542" "Value_measuredElementFS_543"
## [3] "Value_measuredElementFS_544" "Value_measuredElementFS_545"
## [5] "TOTAL_UTILIZATION"           "BALANCE"
```

Now, we have the TOTAL_SUPPLY, TOTAL_UTILIZATION, and BALANCE values.

4.2. Update Edges

The second step in the AUPUS procedure is to update the edges of the commodity network. In this step, we're updating the extraction rates and the input from processing values on the edges data.table with the values from the nodes table. Note: the 131 element represents the input from processing value, and the 41 element has extraction rates.

```
toProcess[, c("timePointYearsSP", "Value_measuredElementFS_131",
              "Value_measuredElementFS_41"), with = F]

##    timePointYearsSP Value_measuredElementFS_131 Value_measuredElementFS_41
## 1:             2009                       83800                   17418.12

edges[, c("timePointYearsSP", "measuredItemChildFS", "Value_share",
          "Value_input", "Value_extraction"), with = FALSE]

##    timePointYearsSP measuredItemChildFS Value_share Value_input
## 1:             2009                  41           0           0
## 2:             2009                  72         100       83800
## 3:             2009                  73         100       83800
## 4:             2009                 113           0           0
## 5:             2009                 632           0           0
## 6:             2009                 634           0           0
## 7:             2009                 654           0           0
##    Value_extraction
## 1:         6600.000
## 2:         8000.000
## 3:         1600.000
## 4:         9500.000
## 5:         2500.000
## 6:         3418.423
## 7:         4000.000

updateEdges(nodes = toProcess, edges = edges, aupusParam = usAupusParam)
edges[, c("timePointYearsSP", "measuredItemChildFS", "Value_share",
          "Value_input", "Value_extraction"), with = FALSE]

##    timePointYearsSP measuredItemChildFS Value_share Value_input
## 1:             2009                  41           0           0
## 2:             2009                  72         100       83800
## 3:             2009                  73         100       83800
## 4:             2009                 113           0           0
## 5:             2009                 632           0           0
## 6:             2009                 634           0           0
## 7:             2009                 654           0           0
##    Value_extraction
## 1:         17418.12
## 2:         17418.12
## 3:         17418.12
```

```
## 4:         17418.12
## 5:         17418.12
## 6:         17418.12
## 7:         17418.12
```

The Value_input column of edges is updated (in theory) to reflect the amount of the parent commodity flowing to the child. In this case, the original values for inputs from processing were already correct, and so nothing is changed with them. However, the extraction rates are different, and those values are changed on the edges table (updated to reflect the nodes table).

4.3. Update Inputs from Processing

In this step, the data from the edges data.table is passed to the nodes data.table at the next processing level.

```
nodesNextLevel = nodes[processingLevel == 1, ]
nodesNextLevel[, c("timePointYearsSP", "measuredItemFS",
                   "Value_measuredElementFS_31"), with = FALSE]

##    timePointYearsSP measuredItemFS Value_measuredElementFS_31
## 1:             2009             41                      245000
## 2:             2009             72                       83800
## 3:             2009             73                       83800
## 4:             2009            113                      237000
## 5:             2009            632                           0
## 6:             2009            634                     4387988
## 7:             2009            654                     3858388

updateInputFromProcessing(nodes = nodesNextLevel,
                          edges = edges,
                          aupusParam = usAupusParam)
nodesNextLevel[, c("timePointYearsSP", "measuredItemFS",
                   "Value_measuredElementFS_31"), with = FALSE]

##    timePointYearsSP measuredItemFS Value_measuredElementFS_31
## 1:             2009             41                           0
## 2:             2009             72                       83800
## 3:             2009             73                       83800
## 4:             2009            113                           0
## 5:             2009            632                           0
## 6:             2009            634                           0
## 7:             2009            654                           0
```

This entire section (i.e. the whole AUPUS procedure) can be run by calling the function Aupus. This function will also compute the processing levels and iterate through each of them.

5. Standardization

Standardization refers to the process of aggregating multiple commodities up to one representative commodity. For example, wheat can be reported directly, or derivatives of wheat, such as wheat flour, can also be reported. To get a simpler view of the food balance sheet, we

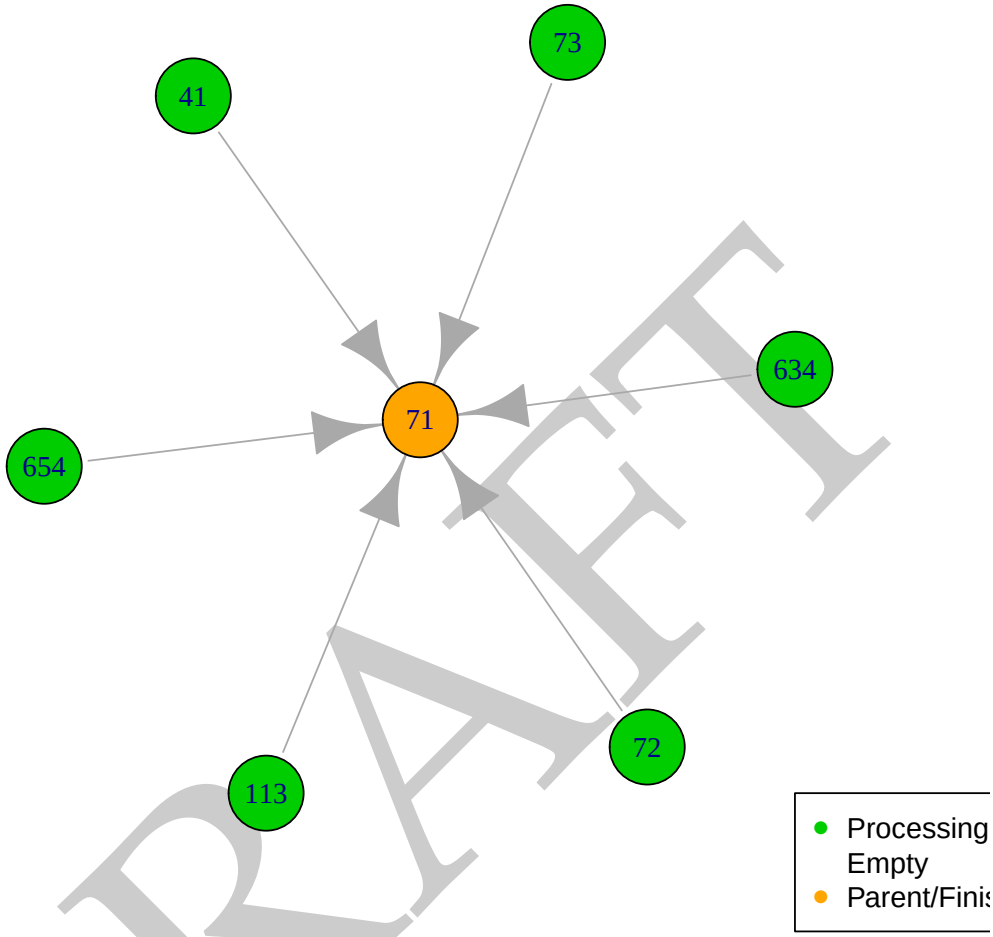
can "standardize" commodities up to their parent commodities. This allows us to reduce the size of the food balance sheet and make it easier to understand/analyze, but still retains all of the food information available.

Now, from the previous sections, we have an object called "updatedAupusNetwork" which contains a data.table called "nodes" and a data.table called "edges." We can convert this object into a graph object using the constructStandardizationGraph function. We also pass a character vector containing the names of the FBS element variables we're interested in.

```
FBSelements =
    c("Value_measuredElementFS_51", "Value_measuredElementFS_61",
      "Value_measuredElementFS_91", "Value_measuredElementFS_101",
      "Value_measuredElementFS_111", "Value_measuredElementFS_121",
      "Value_measuredElementFS_141", "Value_measuredElementFS_151")
standardizationGraph =
    constructStandardizationGraph(aupusNetwork = aupusNetwork,
                                  standardizeElement = FBSelements,
                                  aupusParam = usAupusParam)
is(standardizationGraph)

## [1] "list"   "vector"

sapply(standardizationGraph, class)

##     2009
## "igraph"
```

Let's now begin the standardization procedure:

```
standardization(standardizationGraph[[1]], standardizeElement = FBSelements,
                plot = TRUE, aupusParam = usAupusParam,
                vertex.size = 20, edge.arrow.size = 2, vertex.label.cex = 1)

##    measuredItemFS Value_measuredElementFS_51 Value_measuredElementFS_61
## 1:             71                      177630                     1512481
##    Value_measuredElementFS_91 Value_measuredElementFS_101
## 1:                     3967710                    44497.73
##    Value_measuredElementFS_111 Value_measuredElementFS_121
## 1:                        76200                           0
##    Value_measuredElementFS_141 Value_measuredElementFS_151
## 1:                     1242490                    178120.8
```



To understand more clearly what happened, let's examine element 51 (which represents the quantity of produced goods).

```
vertex.attributes(standardizationGraph[[1]])[1:2]

## $name
## [1] "41"  "71"  "72"  "73"  "113" "632" "634" "654"
##
## $Value_measuredElementFS_51
## [1]  161700  177630   67040   13408  225150       0 1500000 1543355

edge.attributes(standardizationGraph[[1]])[c(3, 4)]

## $Value_input
## [1]     0 83800 83800     0     0     0     0
##
## $flagFaostat_input
## [1] "M" "/" "/" "M" "M" "M" "M"
```

```
E(standardizationGraph[[1]])

## Edge sequence:
##
## [1] 41  -> 71
## [2] 72  -> 71
## [3] 73  -> 71
## [4] 113 -> 71
## [5] 632 -> 71
## [6] 634 -> 71
## [7] 654 -> 71
```

So, items 72 and 73 will be standardized to item 71 (based on the edges). The current value for element 51, item 71 is 177,630. All 67,040 units from item 72 can be standardized/converted to item 71. This is because the Value_share is 100, implying 100% of item 72 came from 71 (and not other commodities as well). The extraction rate values presented are out of a base of 10,000, so for 71 to 72 the actual extraction rate is 8,000/10,000 = 80%. Thus, we can manually calculate the value for element 51, item 71:

```
177630 +                          # The initial value
    67040 * 10000/8000 * 100/100 + # Standardizing element 72
    13408 * 10000/1600 * 100/100   # Standardizing element 73

## [1] 345230
```

This value matches what we computed using the standardization function above. The standardization function, however, performs the above operations for each commodity and element type, and it performs the procedure multiple times (if necessary) to standardize multiple levels back to one main commodity. The fbsStandardization function performs this standardization for each graph and returns the result as a data.table.

```
fbsStandardization(graph = standardizationGraph,
                   standardizeElement = FBSelements,
                   plot = FALSE, aupusParam = usAupusParam)

##    measuredItemFS Value_measuredElementFS_51 Value_measuredElementFS_61
## 1:             71                     177630                     1512481
##    Value_measuredElementFS_91 Value_measuredElementFS_101
## 1:                     3967710                    44497.73
##    Value_measuredElementFS_111 Value_measuredElementFS_121
## 1:                        76200                           0
##    Value_measuredElementFS_141 Value_measuredElementFS_151 timePointYearsSP
## 1:                      1242490                    178120.8             2009
```

Affiliation:

Joshua M. Browning and Michael. C. J. Kao
Economics and Social Statistics Division (ESS)
Economic and Social Development Department (ES)

Food and Agriculture Organization of the United Nations (FAO)
Viale delle Terme di Caracalla 00153 Rome, Italy
E-mail: joshua.browning@fao.org, michael.kao@fao.org
URL: https://svn.fao.org/projects/SWS/RModules/faoswsAupus/