# Input and Output Data Analysis for System Dynamics Modeling using the tidyverse Libraries of R

Jim Duggan,
School of Computer Science,
National University of Ireland Galway.

## Introduction

From a number of perspectives, system dynamics is a data intensive activity. First, each modelling challenge addresses behaviour over time, where historical time series data informs the model building process, and techniques such as calibration and optimisation (Rahmandad, Oliva, & Osgood, 2015) are deployed to estimate parameters, and enhance user confidence in model outputs. Second, the simulation of higher order models (Forrester, 1987) typically yield many time-based observations across a significant number of variables. These results must be interpreted and analysed as part of the model building and policy analysis process. Third, simulation methods such as sensitivity analysis (Hekimoğlu & Barlas, 2016; Jadun, Immerstedt, Bush, Inman, & Peterson, 2017; Walrave, 2016) generate large data sets that need to be processed for further analysis, for example, techniques such as statistical screening (Ford & Flynn, 2005; Taylor, David N. Ford, & Ford, 2010; Yasaman & Ford, 2016). Therefore, in the context of these data intensive modelling processes, there are opportunities for system dynamics modellers to leverage complementary data exploration technologies such as R (Duggan, 2016b).

The R programming language provides a flexible framework for supporting system dynamics modeling. In particular, R now contains a suite of libraries, collectively known as the `tidyverse`[1], that are specifically designed to process *rectangular data*, which is highly structured data with rows as individual observations, and columns containing variables. In a system dynamics context, a row represents the simulation output at a unique point in time, and columns contain model variables (i.e. stocks, flows and auxiliaries). Given this perspective, the output from a system dynamics model with *n* time steps and *m* variables can be viewed as a single rectangular data set with dimensions (*n x m).* Many of the `tidyverse` libraries provide quick and efficient ways to process rectangular data, including: importing data from external sources (e.g. comma-separated files), summarising and aggregating observations (frequency counts, summary functions), and visualising large data sets. Before describing these libraries, an overview of R is provided.

[1] https://www.tidyverse.org

## Overview of R

R is a dialect of the S computer language, which was developed at Bell Laboratories. In 1998 its innovative design was acknowledged by the Association of Computing Machinery Software (ACM) with a citation that R "will forever alter the way people analyse, visualize, and manipulate data". R's inventor, Dr. John Chambers, proposed that "our mission, as users and creators of software for data analysis, is to enable the best and most thorough exploration of data possible." This means that users of the software must be able to ask meaningful questions about their applications, quickly and flexibly (Chambers, 2008). The ubiquity of R as an open-source data analysis tool is confirmed through the growth of available R packages that reside on the *Comprehensive R Archive Network* - CRAN[2]. R can be used with a minimum amount of programming knowledge, and the following ideas (Chambers, 2008) are helpful to understand computations in R (Wickham, 2014):

- *Everything that exists is an object.* Objects are central to storing data in R, and can represent different data structures. Examples of objects include vectors (data of the same type), lists (data of mixed-types), matrices, data frames, and, more recently, tibbles. The *tibble* is used throughout this paper, and is important for analysing output from system dynamics models.

- *Everything that happens is a function call.* Transforming data and generating results are achieved using R functions, from base mathematical functions such as **`exp()`**, **`sin()`** and **`abs()`**, to the range of functions provided by the `tidyverse` libraries.

An implementation of a simple dynamic model in R is now shown[3]. This models of growth for a population P, where the growth model is defined by the derivative shown in equation (1), and $g$ is the fractional growth rate.

$$\frac{dP}{dt} = g\ P \tag{1}$$

The analytical solution to (1) can be derived using the rules of integration, and the solution is shown in equation (2), where $P_0$ is the initial population.

$$P_t = P_0\ e^{gt} \tag{2}$$

Equation (2) is now implemented in R, given an initial population value, the growth rate, and the time horizon for the model (see appendix two for details on how to install R and RStudio). First, a vector object to represent time is created (**`t`**) which contains a sequence of numbers starting at 0 and ending at 10. Note that the symbol **`<-`** is the assignment operator in R.

[2] https://cran.r-project.org

[3] All code examples in this paper are accessible at https://github.com/JimDuggan/SDMR

```
> t <- seq(from = 0,to = 10)
>
> t
 [1]  0  1  2  3  4  5  6  7  8  9 10
```

Elements of a vector can be individual accessed and manipulated. For example, consider the following commands to access the vector values.

```
> # access the first element of t
> t[1]
[1] 0
> # access the first three elements of t
> t[1:3]
[1] 0 1 2
```

*Vectorization* is a process whereby the same operation (or function) is applied to each vector element, and the output is a new vector of the same size, containing the results. This is a powerful feature of R, which allows one single function call to be applied to all individual elements of a vector. For example, we could multiply the vector **t** by a constant and obtain the following result.

```
> t * 10
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

Continuing with the population growth model, two additional variables are created, **g** which represents an arbitrary fractional growth rate, and **init_pop** that contains the initial population value.

```
> g <- 0.02
> g
[1] 0.02

> init_pop <- 1000

> init_pop
[1] 1000
```

All the variables are in place to model population growth through the implementation of equation (2). This multiplies the initial population by the calculation of the exponential function **exp(g*t),** and stores the result in the new variable **pop1**. The operation once again demonstrates vectorization, where the resulting output length (11 elements) is the same as the input length of the input time vector, and therefore the function calculation is made using every element of the input vector **t**.

```
> pop1 <- init_pop * exp(g*t)
>
> pop1
 [1] 1000.000 1020.201 1040.811 1061.837 1083.287 1105.171 1127.497
 [8] 1150.274 1173.511 1197.217
```

Given that in R *everything that happens is a function call*, system dynamics modellers can also define their own purpose-built functions using the *function*

keyword. A function usually has a name, a set of parameters (inputs) and returns a result. The last evaluated expression is automatically returned from a function. The code from the previous example is encapsulated in a function, named **f1**. It takes three parameters: the initial population, a time vector and the growth rate. Simple functions can be defined like this in one line of R code.

```
> f1 <- function(init, t, g) {init * exp(g*t)}
```

The function is then called, and the result copied to the vector **pop2**.

```
> pop2 <- f1(1000, seq(from = 0,to = 10),0.02)
```

Notice that the values in **pop2** are the same as those in **pop1**, the only difference in calculation is that a function was used to generate results for **pop2**.

```
> pop2
 [1] 1000.000 1020.201 1040.811 1061.837 1083.287 1105.171 1127.497
 [8] 1150.274 1173.511 1197.217
```

As this paper will now show, using R with system dynamics does not require a user to write their own purpose-built functions, although that option is always available, and recommended as a way of maximising the benefit of using R.

Furthermore, recent developments in R to support data science have provided an important set of pre-built functions, data and documentation that can be used to process data. The specific `tidyverse` libraries used in this paper, and relevant to system dynamics modellers, are shown in table 1.

Table 1: Summary of useful `tidyverse` libraries for system dynamics

| R Library[4] | Purpose |
|---|---|
| **readr** | Provides functions that support importing rectangular data into R. |
| **tibble** | A data structure that supports storing rectangular data in R. |
| **tidyr** | Provides functions to create tidy data structures that facilitate ease of processing and visualization |
| **dplyr** | Provides functions to process data, and create summaries |
| **ggplot2** | Provides functions for data visualisation, including time series and scatter-plots. |

The first of two examples showing how R can support the system dynamics modelling process is now presented, and focuses on importing and manipulating rectangular time-series data.

[4] Libraries in R can be installed using the function **install.packages(<package name>)**

**Importing, processing and visualising data**

This example focuses on a common modelling task, whereby exploratory analysis is performed on a data set, as part of the model building process. Here, the data explored is based on United Kingdom incidence data from the 1957 influenza outbreak (Vynnycky & Edmunds, 2008), which shows (see table 2) the number of reported cases per week, and by different age cohorts. We will demonstrate how to create a script to import this data, and perform a number of useful data manipulation operations.

Table 2: Time Series Influenza Data from the 1957 Pandemic (UK Data)

| Week | Young | Child | Adult | Elderly |
|---|---|---|---|---|
| **1** | 0 | 0 | 1 | 1 |
| **2** | 0 | 2 | 6 | 1 |
| **3** | 0 | 2 | 4 | 2 |
| **4** | 23 | 73 | 63 | 11 |
| **5** | 63 | 208 | 173 | 41 |
| **6** | 73 | 207 | 171 | 27 |
| **7** | 66 | 150 | 143 | 7 |
| **8** | 26 | 40 | 87 | 29 |
| **9** | 17 | 18 | 33 | 12 |
| **10** | 3 | 4 | 13 | 6 |
| **11** | 2 | 6 | 16 | 5 |
| **12** | 1 | 6 | 11 | 3 |
| **13** | 0 | 1 | 6 | 5 |
| **14** | 0 | 2 | 2 | 2 |
| **15** | 0 | 1 | 3 | 0 |
| **16** | 0 | 1 | 4 | 6 |
| **17** | 0 | 1 | 3 | 0 |
| **18** | 2 | 1 | 7 | 1 |
| **19** | 1 | 1 | 6 | 2 |

The first step is to load the relevant libraries that are needed for the data analysis task, and these are the five libraries referenced earlier in table 1.

```
library(readr)
library(tibble)
library(tidyr)
library(dplyr)
library(ggplot2)
```

Next, the function **`read_csv()`** (from the library *readr*) is used to read in the rectangular data from the comma-separated file, and copy this data into the tibble data structure.

```
> inc <- read_csv("papers/SD tidyverse/data/Incidence.csv")

Parsed with column specification:
cols(
  Week = col_integer(),
  Young = col_integer(),
  Child = col_integer(),
```

```
  Adult = col_integer(),
  Elderly = col_integer()
)
```

The new tibble maps onto the source file structure, with five columns, where R also selects the appropriate data type for each column. Given that there may be many rows in a dataset, the function **`slice()`** – contained in `dplyr` – can be used to view a subset of the data. Slice takes two parameters: (1) the name of the tibble variable, and (2) the rows to be viewed. Below, we can view the first six rows of the tibble, and these match the data previously shown in table 2.

```
> slice(inc,1:6)
# A tibble: 6 x 5
   Week Young Child Adult Elderly
  <int> <int> <int> <int>   <int>
1     1     0     0     1       1
2     2     0     2     6       1
3     3     0     2     4       2
4     4    23    73    63      11
5     5    63   208   173      41
6     6    73   207   171      27
```

While this data represents what is contained in the file, it is not in a good structure to use with the `tidyverse`. This is because for `tidyverse` operations in R, it is important to have data in *tidy format*. There are three rules that must be followed in order to make a tidy dataset (Wickham & Grolemund, 2016): (1) each variable must have its own column; (2) each observation must have its own row; and, (3) each value must have its own cell. In this case, four of the columns shown in table 2 (Young, Child, Adult and Elderly) are not variables in their own right, rather they represent data belonging to a more general variable (e.g. cohort). To facilitate ease of processing, the data shown in table 2 needs to be transformed into what is represented in table 3.

Table 3: Time series influenza data in tidy data format

| Week | Cohort | Incidence |
|---|---|---|
| **1** | Young | 0 |
| **1** | Child | 0 |
| **1** | Adult | 1 |
| **1** | Elderly | 1 |
| **2** | Young | 0 |
| **2** | Child | 2 |
| **2** | Adult | 6 |
| **2** | Elderly | 1 |

In table 3 *every row is an observation, and every column a variable*. Therefore, for each week we have 4 observations, one for each cohort. The library `tidyr` contains a function called **`gather()`** which transforms *untidy* data, such as that shown in table 2, into its tidy data equivalent. The **`gather()`** function takes the following parameters:

- The tibble containing the data to be processed.

- The name for the new variable that will be created. In this case, the variable is called *Cohort*.
- The name for the column containing the values. Here, this column will be named *Incidence.*
- Finally, the range of columns that needed to be "tidied" is provided, and in this case, it is columns 2-4, which can be conveniently referenced by their names, separated by a colon.

The command to create the new tibble is given by:

```
> t_inc <- gather(inc, Cohort, Incidence, Young:Elderly)
```

When this new tibble variable is examined in R, it has 76 rows (19 x 4) and 3 columns, and the data has been narrowed by reducing the number of columns and increasing the number of rows.

```
> t_inc
# A tibble: 76 x 3
    Week Cohort Incidence
   <int>  <chr>     <int>
 1     1  Young         0
 2     2  Young         0
 3     3  Young         0
 4     4  Young        23
 5     5  Young        63
 6     6  Young        73
 7     7  Young        66
 8     8  Young        26
 9     9  Young        17
10    10  Young         3
# ... with 66 more rows
```

The library `dplyr` (Wickham & Grolemund, 2016) provides five verbs (functions) to process tidy data, summarised in table 4.

Table 4: dplyr functions for processing data in tibbles

| R Library | Purpose |
|---|---|
| **`filter`** | Pick observations by their values |
| **`arrange`** | Reorder the rows |
| **`select`** | Pick variables by their names |
| **`mutate`** | Create new variables with functions of existing variables |
| **`summarise`** | Collapse many values down to a single summary |

All five functions work similarly, where:

- The first argument is a tibble.
- The subsequent arguments decide what to do with the tibble.
- The result is a tibble.

Based on the two tibbles (**`t_inc`** and **`inc`**) examples of how to use these functions is now presented. The **`filter()`** function allows the user to specify a filtering condition on a tibble and therefore the result is a new tibble whereby the rows are subsetted. For example, to create a new tibble that only contains

results from the younger cohort, the following call can be made, using the equality operator ==.

```
> filter(t_inc,Cohort=="Young")
# A tibble: 19 x 3
    Week Cohort Incidence
   <int>  <chr>     <int>
 1     1  Young         0
 2     2  Young         0
 3     3  Young         0
 4     4  Young        23
 5     5  Young        63
 6     6  Young        73
```

More complex conditions can be modelled, for example, to extract the first five weeks data for the young cohort, the "and" operator (`&`) can be used as part of the condition.

```
> filter(t_inc,Cohort=="Young" & Week <=5)
# A tibble: 5 x 3
   Week Cohort Incidence
  <int>  <chr>     <int>
1     1  Young         0
2     2  Young         0
3     3  Young         0
4     4  Young        23
5     5  Young        63
```

The second `dplyr` function is **`arrange(),`** which sorts the data in ascending order. The function **`desc()`** can also be used to re-arrange in descending order.

```
> f <- filter(t_inc,Cohort=="Young" & Week <=5)
>
> arrange(f,Incidence)
# A tibble: 5 x 3
   Week Cohort Incidence
  <int>  <chr>     <int>
1     1  Young         0
2     2  Young         0
3     3  Young         0
4     4  Young        23
5     5  Young        63


> arrange(f,desc(Incidence))
# A tibble: 5 x 3
   Week Cohort Incidence
  <int>  <chr>     <int>
1     5  Young        63
2     4  Young        23
3     1  Young         0
4     2  Young         0
5     3  Young         0
```

The function **`select()`** is useful in situations where the data set has many variables and the modeller needs to focus on a reduced subset. In contrast to the **`filter()`** function which reduces the number of rows, the **`select()`** function reduces the number of columns. Returning to the tibble `inc`, the columns for the young cohort could be extracted as follows:

```
> s_data <- select(inc,Week:Young)
>
> slice(s_data,1:6)
# A tibble: 6 x 2
   Week Young
  <int> <int>
1     1     0
2     2     0
3     3     0
4     4    23
5     5    63
6     6    73
```

The function **`mutate()`** facilitates the creation of new tibble variables based on the values of tibble variables. For example, if the total number of infected per week needed to be calculated from the original data set (by summing the four incidence columns), the **`mutate()`** function can be used to add this new column to the tibble (the first 10 rows are shown below).

```
> tot_inc <- mutate(inc,Total=Young+Child+Adult+Elderly)

> slice(tot_inc,1:10)
# A tibble: 10 x 6
    Week Young Child Adult Elderly Total
   <int> <int> <int> <int>   <int> <int>
 1     1     0     0     1       1     2
 2     2     0     2     6       1     9
 3     3     0     2     4       2     8
 4     4    23    73    63      11   170
 5     5    63   208   173      41   485
 6     6    73   207   171      27   478
 7     7    66   150   143       7   366
 8     8    26    40    87      29   182
 9     9    17    18    33      12    80
10    10     3     4    13       6    26
```

Finally, the function **`summarise()`** provides a convenient way to collapse many row values down to a single summary. There are two additional functions that can be used with summarise to enhance its operation:

- The function **`group_by()`** which takes an existing tibble and converts it into a grouped tibble so that subsequent summary operations can be performed by group.

- The pipe operator (%>%), which is a powerful tool for clearly expressing a sequence of operations (Wickham & Grolemund, 2016). The output from one operation can be sent directly into the input for a subsequent operation, and therefore it eliminates the need for temporary variables.

As an example, consider the tidy data in the tibble **`t_inc`**. The following command will create a new tibble that contains the sum of cohort incidence

values from each week. In effect, this collapses 4 weekly row observations (Young, Child, Adult and Eldery) into one single aggregated value.

```
> wk_tot <- t_inc %>% group_by(Week) %>%
        summarise(Incidence=sum(Incidence))
```

The sequence of steps in this command are:

- The tibble **t_inc** is the input data set. This is then "piped" into the function **group_by().**
- The function **group_by()** groups the data by the column *Week*, and this resulting tibble is "piped" into the function **summarise().**
- The function **summarise()** creates an output tibble that contains two columns, and aggregates observations into summarised values. The first is *Week*, the second is *Incidence*, which contains the sum of the incidence values for each week, where the values from four rows are summed into one row.

The resulting aggregate tibble (first six rows only) is shown below.

```
> slice(wk_tot,1:6)
# A tibble: 6 x 2
   Week Incidence
  <int>     <int>
1     1         2
2     2         9
3     3         8
4     4       170
5     5       485
6     6       478
```

Additional analysis can also be performed using the summarise function. As a reminder, the variable **t_inc** contains (in tidy data format – with just one observation per row) all of the weekly incidence values for each cohort.

```
> t_inc
# A tibble: 76 x 3
    Week Cohort Incidence
   <int>  <chr>     <int>
 1     1  Young         0
 2     2  Young         0
 3     3  Young         0
 4     4  Young        23
 5     5  Young        63
 6     6  Young        73
 7     7  Young        66
 8     8  Young        26
 9     9  Young        17
10    10  Young         3
# ... with 66 more rows
```

Because the data is in this format, further analysis can be performed using the **summarise()** function, where the data is grouped by each cohort type. For example, for each group we can calculate the: total number of infected, peak value, peak week, average, standard deviation, minimum and maximum values. Note that the **which()** function in R provides the row index of a value, and

therefore can be used to extract other variables that are contained in the same row of data. In this case, once we know the row number for the maximum incidence value, the week number can then be identified. The functions **mean()** and **sd()** are R functions to calculate the mean and standard deviation.

```
t_coh <- t_inc %>%
           group_by(Cohort) %>%
           summarise(TotalInfected=sum(Incidence),
                     PeakValue=max(Incidence),
                     PeakWeek=Week[which(Incidence==max(Incidence))],
                     AvrValue=mean(Incidence),
                     SD=sd(Incidence))
```

The output is shown below. This demonstrates the power of `dplyr`, in that useful information can be easily aggregated, and so provide insight to support the modeling process.

```
> t_coh
# A tibble: 4 x 6
  Cohort  TotalInfected PeakValue PeakWeek AvrValue    SD
  <chr>           <int>     <dbl>    <int>    <dbl> <dbl>
1 Adult             752      173.        5    39.6   59.3
2 Child             724      208.        5    38.1   70.1
3 Elderly           161       41.        5     8.47  11.4
4 Young             277       73.        6    14.6   24.9
```

In addition to data processing, visualization is an important activity for data exploration. Once data is in a tidy format, the R library `ggplot2` can be used to visualize data (Wickham, 2016). This is essentially a mini-language specifically tailored for producing graphics, and it does so in an intuitive and layered manner, where additional transformations can be added to a chart by using the addition (+) operator. For example, consider the data stored in the variable **tot_inc** which stores an additional column (Total) that contains a summary of weekly values. To display this on a chart (with time on the x-axis), the following command is used.

```
ggplot(data=tot_inc,mapping = aes(x=Week,y=Total)) +
      geom_line() + geom_point()
```

The first call is to **ggplot(),** and the data source is identified (the tibble tot_inc), and the mapping between the data values for the x and y axis are specified in a called to the function **aes()** – which is an abbreviation for the word aesthetic. Once the data and the mappings are defined, subsequent function calls can be made to (1) draw a line and (2) show each data point on the line. The overall graph generated by this code is shown in figure 1.

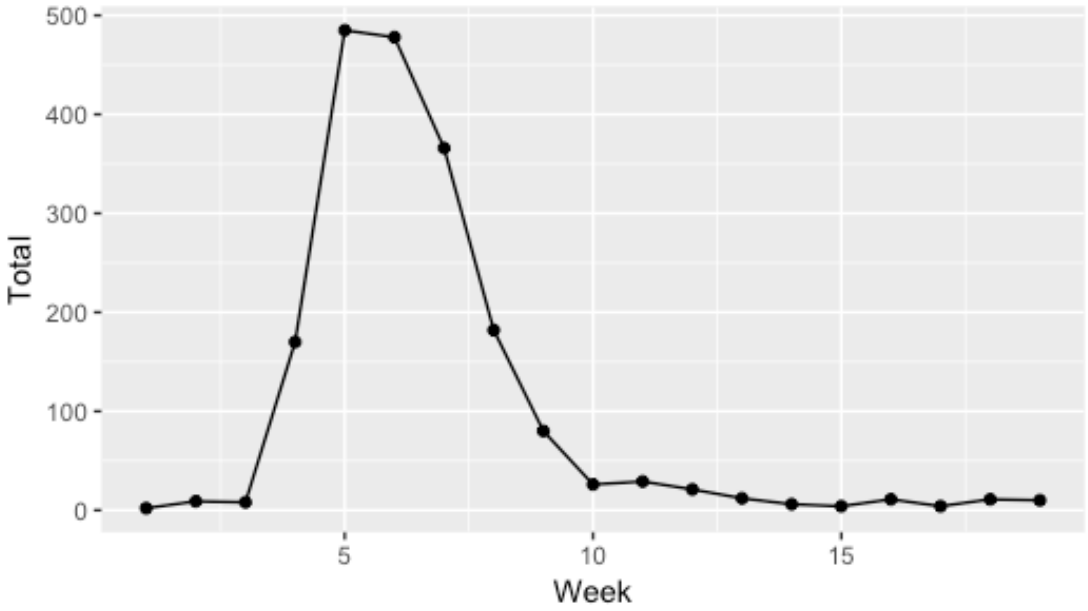

Figure 1: View of the total incidence values

The `ggplot2` library offers additional features for generating user-friendly visualisations. For example, the function **`aes()`** accepts additional parameters which can be used to distinguish categorical data (e.g. cohorts) in the plot. In the example below (shown in figure 2),, the parameter *colour* is set to the variable *Cohort*, and the `ggplot2` system will automatically generate four separate subplots, colour these differently, and provide a supporting legend.

```
ggplot(t_inc,aes(x=Week,y=Incidence,color=Cohort)) +
       geom_line() + geom_point()
```

There are a wide range of visualization options for further producing graphics using ggplot2, and the interested reader is referred to (Wickham, 2016).

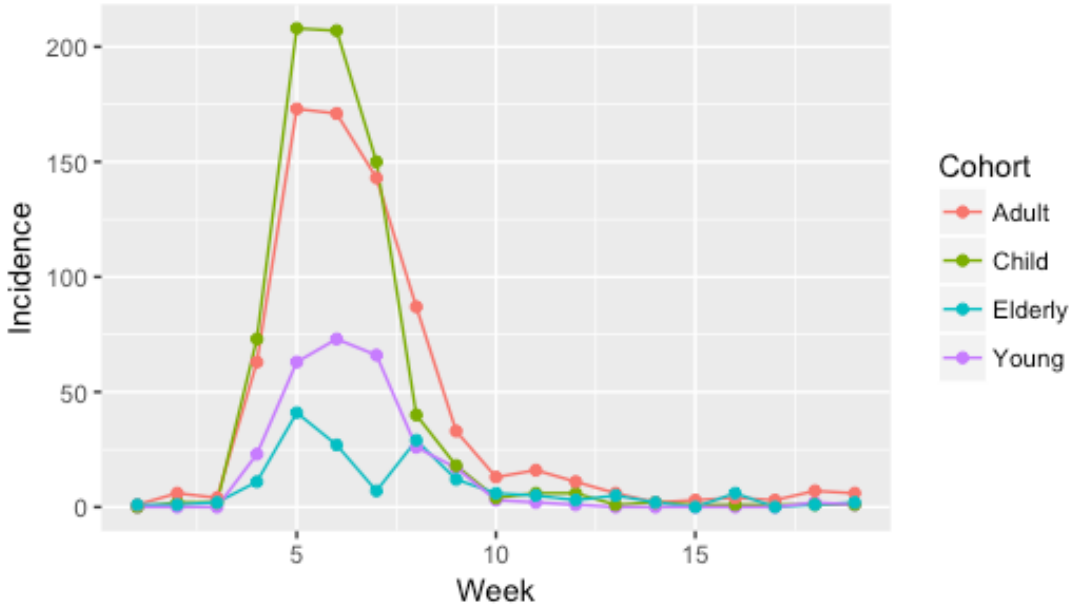


Figure 2: View of the total incidence values

In summary, this section has shown how to use the `tidyverse` libraries to:

- Read rectangular data into an R program, and explore data using five functions in `dplyr`.
- Convert wide data to narrow format, so that every row is an observation, and each column is a variable.
- Use the pipe operator along with **`group_by()`** and **summarise()** to generate aggregate summaries of data.
- Visualise tibble data using `ggplot2`.

For system dynamics, a key objective is also to process and visualize simulation output data, and the following section shows how the `tidyverse` functions can be used to process simulation output, where the output is generated from a system dynamics modeling tool.

**Analysis of Simulation Model Output**

In this example, R is used in a *post-processing manner*, where the output, generated in Vensim, is from a disaggregate *Susceptible – Infected – Recovered* model is processed using the `tidyverse`. The stock and flow model is shown in figure 3 (Duggan, 2016a), and it provides a useful structure to explore infectious diseases transmission dynamics between three population cohorts (see appendix one for the complete equation listing). These three cohorts are young (Y), adult (A) and elderly (E). A simulation run is executed in Vensim, and all the simulation results are exported to a file.

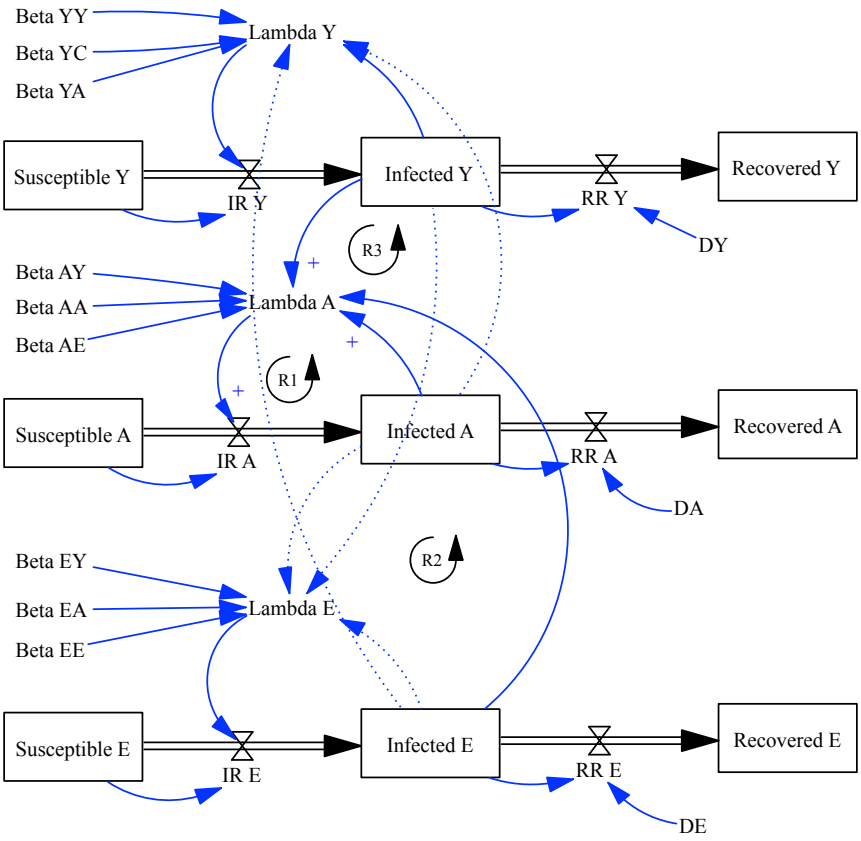


Figure 3: The SIR Model for three cohorts

The idea here is that every model variable from the simulation run is contained in the output file, and this file is then imported into R using the **`read_csv()`** function.

```
res <- read_csv("tidyverse/data/SimulationOutput.csv")
```

The variable **`res`** is a tibble data structure with 47 variables (columns) and 161 observations (rows) for each time step of the simulation. While 47 variables may seem unwieldy, the key point is that the *five functions* from `dplyr` can be used to process the data. The R function **`colnames()`** can be used to display all the model variables.

```
> colnames(res)
 [1] "Time"              "Beta AA"           "Beta AE"
 [4] "Beta AY"           "Beta EA"           "Beta EE"
 [7] "Beta EY"           "Beta YA"           "Beta YE"
[10] "Beta YY"           "CE AA"             "CE AE"
[13] "CE AY"             "CE EA"             "CE EE"
[16] "CE EY"             "CE YA"             "CE YE"
[19] "CE YY"             "DA"                "DE"
[22] "DY"                "Infected A"        "Infected E"
[25] "Infected Y"        "IR A"              "IR E"
[28] "IR Y"              "Lambda A"          "Lambda E"
[31] "Lambda Y"          "Pop A"             "Pop E"
[34] "Pop Y"             "Prop A Infected"   "Prop E Infected"
[37] "Prop Y Infected"   "Recovered A"       "Recovered E"
[40] "Recovered Y"       "RR A"              "RR E"
```

```
[43] "RR Y"              "Susceptible A"    "Susceptible E"
[46] "Susceptible Y"     "Total Population"
```

Given that the model has a disaggregate structure, an initial task is to select all the stock variables for each cohort. This can be done using the **select()** function, along with a function called **starts_with(),** whereby variables starting with a common root can be selected. In this example, all model variable names that start with "Susceptible", "Infected" and "Recovered" are selected from the **res** tibble, and "piped" to the **out** tibble.

```
out <- res %>%
         select(Time,starts_with("Susceptible"),
                     starts_with("Infected"),
                     starts_with("Recovered"))
```

The **out** variable shows a significant reduction in variables (from 47 to 10), and has the same number of rows.

```
> out
# A tibble: 161 x 10
    Time `Susceptible A` `Susceptible E` `Susceptible Y` `Infected A`
   <dbl>           <dbl>           <dbl>           <dbl>        <dbl>
 1 0.000           50000           25000           25000      0.00000
 2 0.125           50000           25000           25000      0.00000
 3 0.250           50000           25000           25000      0.01562
 4 0.375           50000           25000           25000      0.05469
 5 0.500           50000           25000           25000      0.12850
 6 0.625           50000           25000           25000      0.25320
 7 0.750           50000           25000           24990      0.45220
 8 0.875           50000           25000           24990      0.75850
 9 1.000           50000           25000           24990      1.22000
10 1.125           50000           24990           24980      1.90200
# ... with 151 more rows, and 5 more variables: `Infected E` <dbl>,
`Infected Y` <dbl>, `Recovered A` <dbl>, `Recovered E` <dbl>,
`Recovered Y` <dbl>
```

In order to support further analysis, the tibble **out** can be *tidied* so that each row contains one observation. Similar to the earlier transformation of the data in Table 2, the **gather()** function is used to simplify the data structure from 10 columns to 3, and the number of rows are increased to 1,449 (which is 161 observations x 9 different stock variables). The **gather()** function tidies the data in all the columns, from `*Susceptible A*` to `*Recovered Y*`.

```
out_td <- out %>% gather(key=Variable,value = Amount,
                         `Susceptible A`:`Recovered Y`)
> out_td
# A tibble: 1,449 x 3
    Time      Variable Amount
   <dbl>         <chr>  <dbl>
 1 0.000 Susceptible A  50000
 2 0.125 Susceptible A  50000
 3 0.250 Susceptible A  50000
 4 0.375 Susceptible A  50000
 5 0.500 Susceptible A  50000
 6 0.625 Susceptible A  50000
 7 0.750 Susceptible A  50000
 8 0.875 Susceptible A  50000
 9 1.000 Susceptible A  50000
```

```
10 1.125 Susceptible A  50000
# ... with 1,439 more rows
```

With this tidy format, informative categorical information can now be added to each observation. In particular, the cohort (young, adult or elderly) and class (susceptible, infected or recovered) for each observation can be identified using the following R features:

- Two new columns are added (Cohort and Class) using the **`mutate()`** function.
- The function **`case_when()`** can be used to perform an if-else style operation on the data. If the condition is true (left hand side), the variable is set to the corresponding value on the right hand side.
- Within the **`case_when()`** function, the string matching function **`grepl()`** is used. A string pattern is specified as the first parameter, using *regular expression* rules.

For example, the regular expression pattern **"A$"** will match any term that ends in "A", as the character **"$"** is an anchor that represents the end of a string. The regular expression pattern "^S" will match any term that starts with "S", given that the character **"^"** is an anchor that represents the start of a string.

```
new_td <- out_td %>%
           mutate(Cohort=case_when(
                   grepl("A$",Variable) ~ "Adult",
                   grepl("E$",Variable) ~ "Elderly",
                   grepl("Y$",Variable) ~ "Young"),
                  Class=case_when(
                   grepl("^S",Variable) ~ "Susceptible",
                   grepl("^I",Variable) ~ "Infected",
                   grepl("^R",Variable) ~ "Recovered"))
```

The flexibility offered by regular expressions can support the creation of new variable types based on string matching, and this is confirmed by examining the the tibble **`new_td`**.

```
> slice(new_td,1:6)
# A tibble: 6 x 5
   Time Variable      Amount Cohort Class
  <dbl> <chr>          <dbl> <chr>  <chr>
1 0.    Susceptible A 50000. Adult  Susceptible
2 0.125 Susceptible A 50000. Adult  Susceptible
3 0.250 Susceptible A 50000. Adult  Susceptible
4 0.375 Susceptible A 50000. Adult  Susceptible
5 0.500 Susceptible A 50000. Adult  Susceptible
6 0.625 Susceptible A 50000. Adult  Susceptible
```

There are a number of advantages to adding categorical variables to the data structure. The first is that it allows the data to be aggregated by disease

compartment, in that the total sum of susceptible, infected and recovered can be calculated for each time step. This is shown with the new variable **agg** which groups the observations by time and class (susceptible, infected and recovered), and then sums the amount.

```
> agg <- new_td %>% group_by(Time,Class) %>%
+           summarise(Total=sum(Amount))

> agg
# A tibble: 483 x 3
# Groups:   Time [?]
     Time Class              Total
    <dbl> <chr>              <dbl>
 1 0.000  Infected           1.00
 2 0.000  Recovered          0.00
 3 0.000  Susceptible   100000.00
 4 0.125  Infected           1.44
 5 0.125  Recovered          0.0625
 6 0.125  Susceptible    100000.00
 7 0.250  Infected           2.04
 8 0.250  Recovered          0.152
 9 0.250  Susceptible    100000.00
10 0.375  Infected           2.88
# ... with 473 more rows
```

The second benefit of adding categorical variables to data sets is that the simulation results can be visualised over each cohort. For example, this new aggregate data can be plotted using the variable *Class* as a colour, and generate the graph shown in figure 4.

```
ggplot(agg,aes(x=Time,y=Total,colour=Class)) + geom_point()
+geom_line()
```

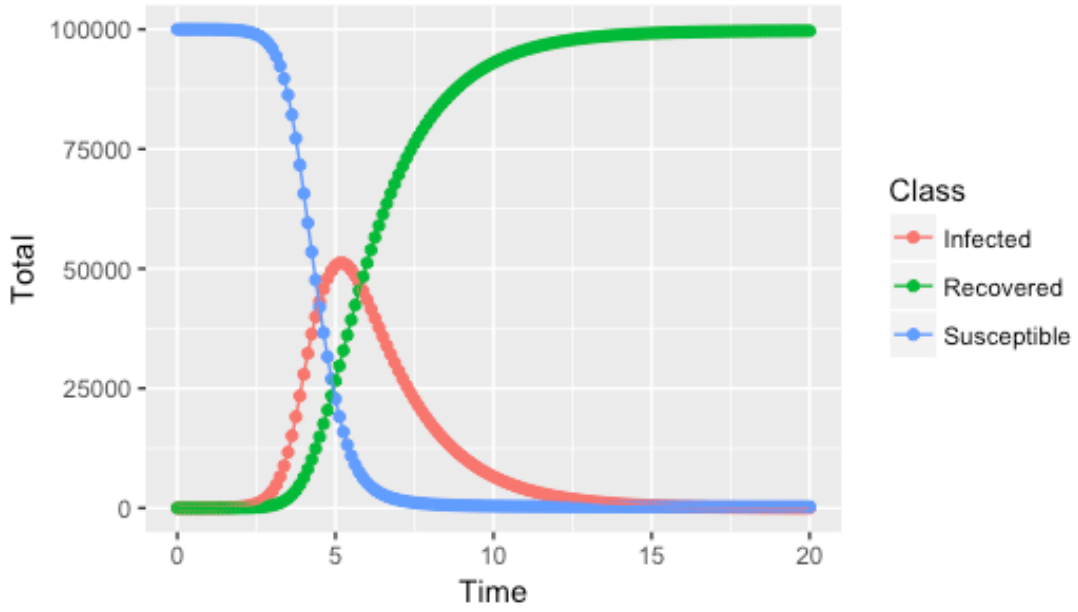


Figure 4: Aggregate stock data calculated by R

## Conclusion

While R is primarily viewed as a toolset to support data scientists, innovative new libraries such as the `tidyverse` can be leveraged to support the system dynamics model building process. This paper has shown how time series data can be accessed and manipulated, and how the entire model output from a simulation run can be processed for informative summaries, and for data visualisation. A further application of the `tidyverse` is to support the process of analysing large data sets produced through sensitivity analysis of system dynamics models.

## References


Chambers, J. M. (2008). *Software for Data Analysis: Programming with R*: Springer Publishing Company, Incorporated.

Duggan, J. (2016a). Diffusion Models *System Dynamics Modeling with R* (pp. 97-121). Cham: Springer International Publishing.

Duggan, J. (2016b). An Introduction to R *System Dynamics Modeling with R* (pp. 25-47). Cham: Springer International Publishing.

Ford, A., & Flynn, H. (2005). Statistical screening of system dynamics models. *System Dynamics Review, 21*(4), 273-303.

Forrester, J. W. (1987). Lessons from system dynamics modeling. *System Dynamics Review, 3*(2), 136-149. doi:10.1002/sdr.4260030205

Hekimoğlu, M., & Barlas, Y. (2016). Sensitivity analysis for models with multiple behavior modes: a method based on behavior pattern measures. *System Dynamics Review, 32*(3-4), 332-362. doi:doi:10.1002/sdr.1568

Jadun, P., Immerstedt, L. J., Bush, B., Inman, D., & Peterson, S. (2017). Application of a variance-based sensitivity analysis method to the Biomass Scenario Learning Model. *System Dynamics Review, 33*(3-4), 311-335. doi:doi:10.1002/sdr.1594

Rahmandad, H., Oliva, R., & Osgood, N. (2015). *Analytical methods for dynamic modelers*: MIT Press.

Taylor, T. R. B., David N. Ford, & Ford, A. (2010). Improving model understanding using statistical screening *System Dynamics Review, 26*(1), 73-87.

Vynnycky, E., & Edmunds, W. J. (2008). Analyses of the 1957 (Asian) influenza pandemic in the United Kingdom and the impact of school closures. . *Epidemiology and infection, 136*(02), 166-179.

Walrave, B. (2016). Determining intervention thresholds that change output behavior patterns. *System Dynamics Review, 32*(3-4), 261-278. doi:doi:10.1002/sdr.1564

Wickham, H. (2014). *Advanced R*: CRC Press.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*: Springer.

Wickham, H., & Grolemund, G. (2016). *R for Data Science*: O'Reilly.

Yasaman, J., & Ford, D. N. (2016). Quantifying the impacts of rework, schedule pressure, and ripple effect loops on project schedule performance. *System Dynamics Review, 32*(1), 82-96. doi:doi:10.1002/sdr.1551

## Appendix One: SIR Cohort Model Equations

```
(01)  Beta AA = CE AA / Pop A

(02)  Beta AE = CE AE / Pop A

(03)  Beta AY = CE AY / Pop A

(04)  Beta EA = CE EA / Pop E

(05)  Beta EE = CE EE / Pop E

(06)  Beta EY = CE EY / Pop E

(07)  Beta YA = CE YA / Pop Y

(08)  Beta YE = CE YE / Pop Y

(09)  Beta YY = CE YY / Pop Y

(10)  CE AA = 2

(11)  CE AE = 1

(12)  CE AY = 0

(13)  CE EA = 1

(14)  CE EE = 0.5

(15)  CE EY = 1

(16)  CE YA = 2

(17)  CE YE = 1

(18)  CE YY = 3

(19)  DA = 2

(20)  DE = 2

(21)  DY = 2

(22)  Infected A = INTEG( IR A - RR A , 0)

(23)  Infected E = INTEG( IR E - RR E , 0)

(24)  Infected Y = INTEG( IR Y - RR Y , 1)

(25)  IR A = Lambda A * Susceptible A

(26)  IR E = Susceptible E * Lambda E

(27)  IR Y = Lambda Y * Susceptible Y

(28)  Lambda A = Beta AY * Infected Y + Beta AA * Infected A + Beta
AE * Infected E
```

```
(29)  Lambda E = Beta EE * Infected E + Beta EA * Infected A + Beta
EY * Infected Y

(30)  Lambda Y = Beta YY * Infected Y + Beta YA * Infected A + Beta
YE * Infected E

(31)  Pop A = Susceptible A + Infected A + Recovered A

(32)  Pop E = Susceptible E + Infected E + Recovered E

(33)  Pop Y = Susceptible Y + Infected Y + Recovered Y

(34)  Prop A Infected = 100 * Recovered A / Pop A

(35)  Prop E Infected = 100 * Recovered E / Pop E

(36)  Prop Y Infected = 100 * Recovered Y / Pop Y

(37)  Recovered A = INTEG( RR A , 0)

(38)  Recovered E = INTEG( RR E , 0)

(39)  Recovered Y = INTEG( RR Y , 0)

(40)  RR A = Infected A / DA

(41)  RR E = Infected E / DE

(42)  RR Y = Infected Y / DY

(43)  Susceptible A = INTEG( - IR A , 50000)

(44)  Susceptible E = INTEG( - IR E , 25000)

(45)  Susceptible Y = INTEG( - IR Y , 24999)

(46)  Total Population = Pop E + Pop A + Pop Y


(47)  FINAL TIME = 20
      The final time for the simulation.

(48)  INITIAL TIME = 0
      The initial time for the simulation.

(49)  SAVEPER = TIME STEP  [0,?]
      The frequency with which output is stored.

(50)  TIME STEP = 0.125 [0,?]
      The time step for the simulation.
```

## Appendix Two: Installing R Studio

R is a free software environment for statistical computing and graphics, and it compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To download R, access the web page https://www.r-project.org and follow the instructions.

Once R is installed, it is recommended to install R Studio.

RStudio is available on https://www.rstudio.com is an integrated development environment (IDE) for R, and is free and open-source. It provides an interactive workbench for creating, testing and running R scripts. This includes separate windows (see figure 5) for:

- R scripts, containing the model equations and data processing scripts.
- An interactive console, for running, testing and debugging commands.
- The global environment, containing information on all variables stored in R's workspace.
- Access to the file system and graphical plots.

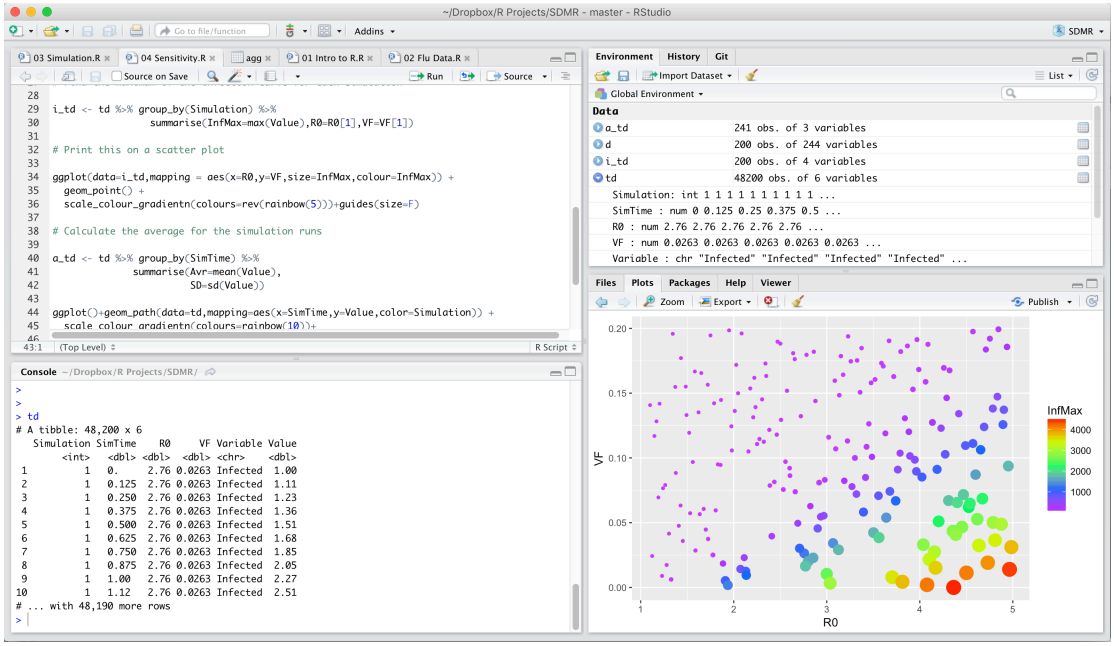


Figure 5 The R Studio IDE

Finally, all the examples are available at https://github.com/JimDuggan/SDMR .