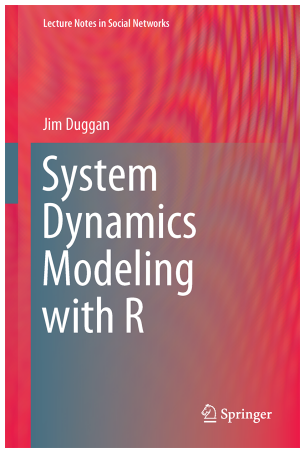
CT561: Systems Modelling & Simulation

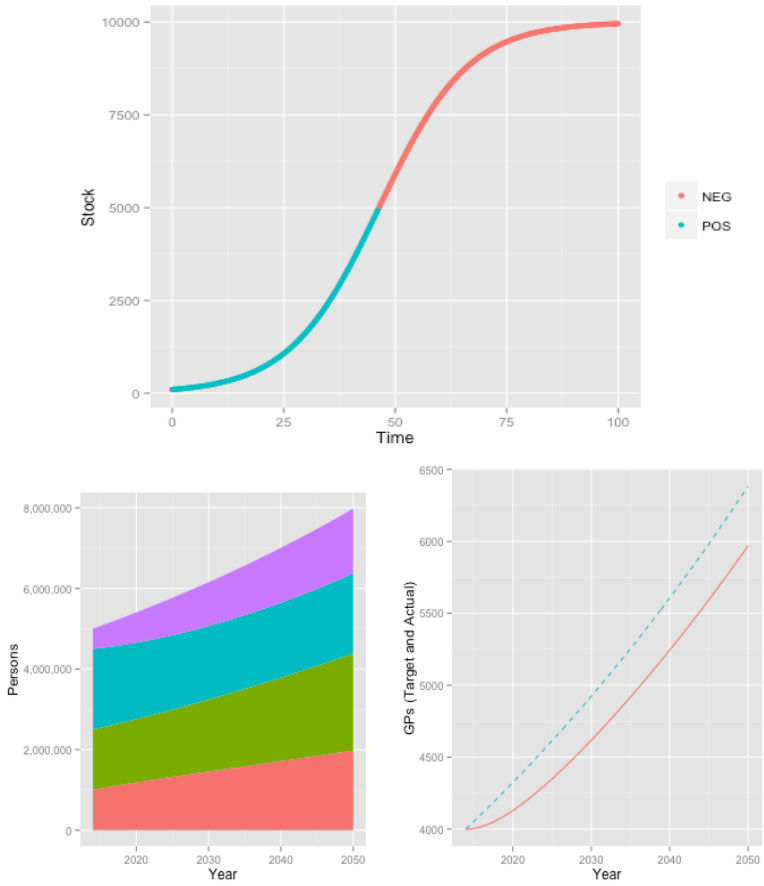
Lecture 7: Introduction to R and the deSolve package

Dr. Jim Duggan,
School of Engineering & Informatics
National University of Ireland Galway.
https://github.com/JimDuggan/SDMR
https://twitter.com/_jimduggan



What is R?

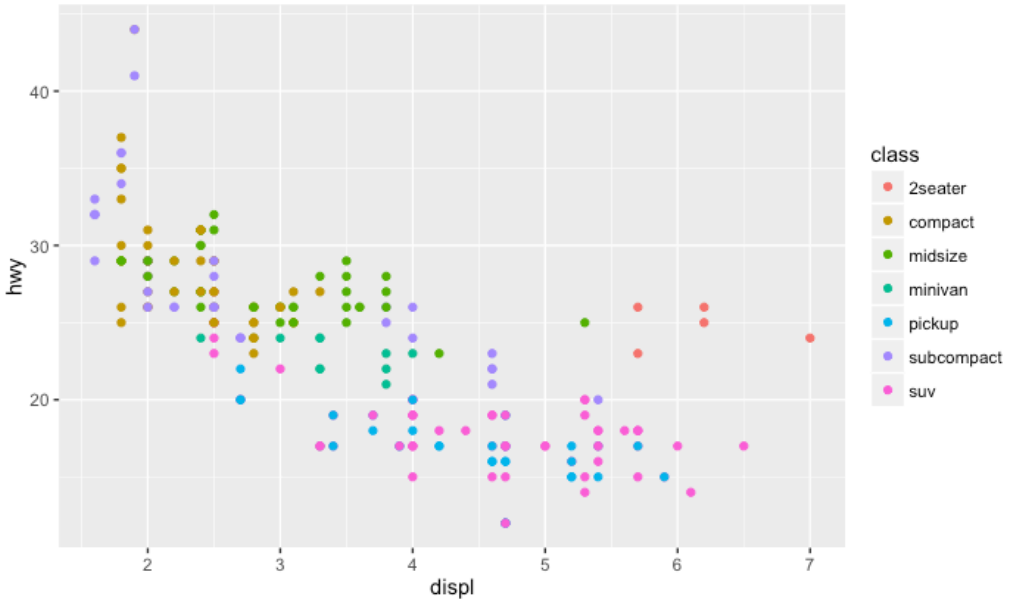
- It is a dialect of the S language, developed at Bell Laboratories
- Open source, functional & object-oriented language
- R's *mission* is to enable the best and most thorough exploration of data possible (Chambers 2008).



Lecture Overview

1. Atomic Vectors
2. Functions
3. Lists
4. Data Frames
5. Visualisation
6. deSolve package

	Homogenous	Heterogenous
1d	**Atomic Vector**	**List**
2d	Matrix	**Data Frame**
nd	Array	



Vectors

- The basic data structure in R is the Vector
- Vectors come in two flavours
  - Atomic vectors
  - Lists
- They have 3 common properties
  - Type, typeof(), what it is
  - Length, length(), how many elements
  - *Attributes, attributes, additional metadata*

(1) Atomic Vectors

- A sequence of elements that have the same data type (Matloff 2009)
- Four common types
  - logical
  - integer
  - double (or numeric)
  - character
- Usually created with c() – short for combine

```
> dbl_var <- c(2.2, 2.5, 2.9)
> str(dbl_var)
 num [1:3] 2.2 2.5 2.9
>
> int_var <- c(0L, 1L, 2L)
> str(int_var)
 int [1:3] 0 1 2
>
> log_var<- c(TRUE, TRUE, F, FALSE)
> str(log_var)
 logi [1:4] TRUE TRUE FALSE FALSE
>
> chr_var<- c("CT5102","CT561")
> str(chr_var)
 chr [1:2] "CT5102" "CT561"
```

Atomic vector types

```
> dbl_var
[1] 2.2 2.5 2.9
> typeof(dbl_var)
[1] "double"
>
> int_var
[1] 0 1 2
> typeof(int_var)
[1] "integer"
```

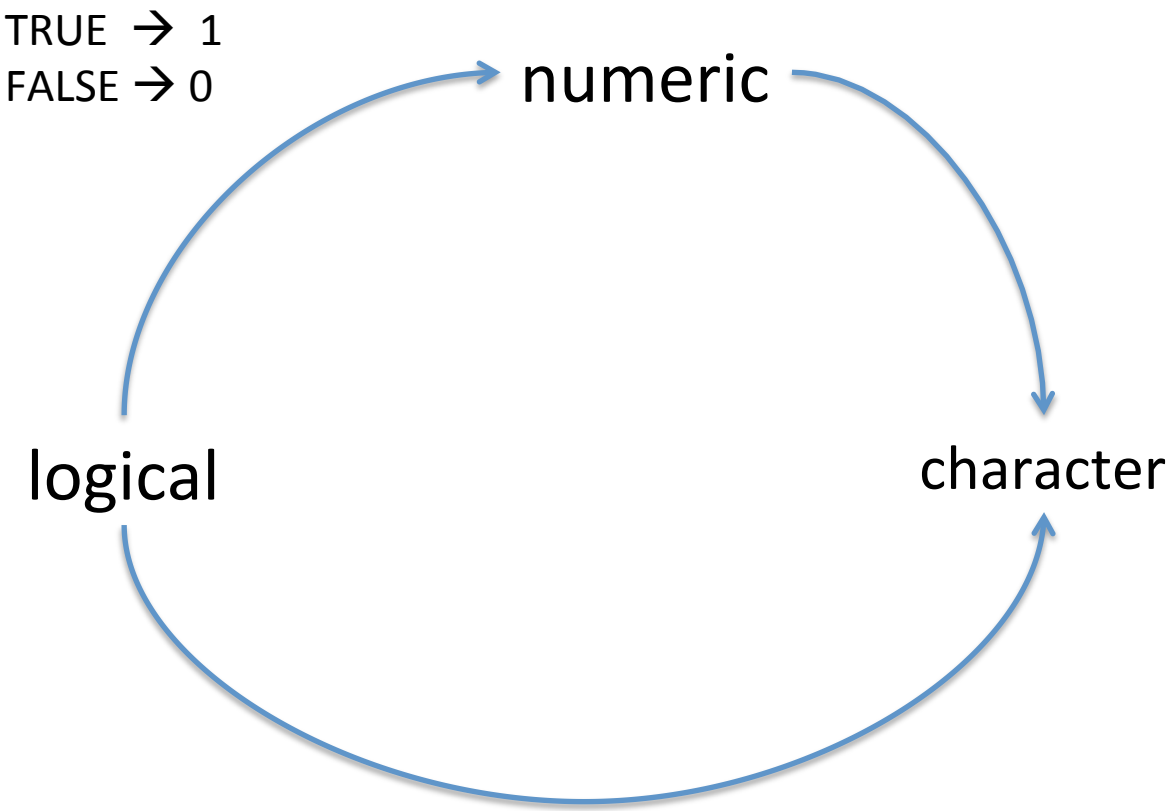
```
> log_var
[1]  TRUE  TRUE FALSE FALSE
> typeof(log_var)
[1] "logical"
>
> chr_var
[1] "CT5102" "CT561"
> typeof(chr_var)
[1] "character"
```

Coercion of atomic vectors

- All elements of an atomic vector MUST be of the same type
- When different type are combined, they will be coerced into the most flexible types
- What will be the type and values of
  - c(1L, T, F)
  - c(1,T,F)

Coercion Rules
Least to most flexible

- logical
- integer
- double
- character



Grolemund (2014) p 52

Challenge 7.1

- Determine the types for each of the following (coerced) vectors

```
v1<- c(1L, T, FALSE)

v2<- c(1L, T, FALSE, 2)

v3<- c(T, FALSE, 2, "FALSE")

v4<- c(2L, "FALSE")

v5<- c(0L, 1L, 2.11)
```

Creating atomic vectors using sequences

The colon operator (:) generates regular sequences (atomic vectors) within a specified range.

```
> v1<-1:10
> v1
 [1]  1  2  3  4  5  6  7  8  9 10

> v2<-3:13
> v2
 [1]  3  4  5  6  7  8  9 10 11 12 13
```

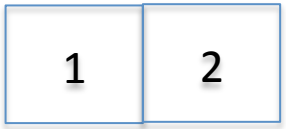
Subsetting Atomic Vectors

- R's subsetting operators are powerful and fast
- For atomic vectors, the operator [is used
- There are six ways to subset an atomic vector in R
- In R, the index for a vector starts at 1

```
> x<-c(2.1, 4.2, 3.3, 5.4)
>
> x
[1] 2.1 4.2 3.3 5.4
>
>
> x[1]
[1] 2.1
> x[c(1,3)]
[1] 2.1 3.3
```

(1) Positive integers

Positive integers return elements at the specified position




```
> x<-1:5
>
> x
[1] 1 2 3 4 5
>
> x[1:2]
[1] 1 2
>
> x[5]
[1] 5
>
> x[5:1]
[1] 5 4 3 2 1
```

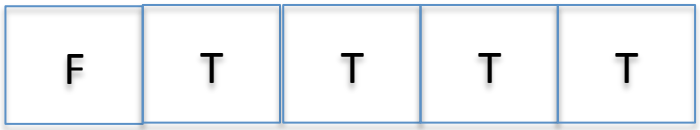
(2) Negative integers

Negative integers omit elements at specified positions

```
> x
[1] 1 2 3 4 5
>
> x[-1]
[1] 2 3 4 5
>
> x[-(3:4)]
[1] 1 2 5
```

(3) Logical Vectors

Select elements where the corresponding logical value is TRUE. This approach supports ***recycling***




```
> x
[1] 1 2 3 4 5
> x[c(F,T,T,T,T)]
[1] 2 3 4 5
```

```
> x
[1] 1 2 3 4 5
>
> x[c(T,F)]
[1] 1 3 5
```

Logical Vectors - Advantages

Expressions can be used to create a logical vector

```
> x
[1] 1 2 3 4 5
> b <- x < median(x)
>
> b
[1]  TRUE  TRUE FALSE FALSE FALSE
> x[b]
[1] 1 2
> x[x<median(x)]
[1] 1 2
```

Logical Expressions in R

Operators	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	not x
x | y	x OR y
x & y	x AND y

```
> x
[1] 1 2 3 4 5
>
> b<- x<median(x) | x > median(x)
>
> b
[1]  TRUE  TRUE FALSE  TRUE  TRUE
>
> x[b]
[1] 1 2 4 5
```

Challenge 7.2

- Create an R vector of squares of 1 to 10
- Find the minimum
- Find the maximum
- Find the average
- Subset all those values greater than the average

(2) Functions

- A function is a group of instructions that:
  - takes input,
  - uses the input to compute other value, and
  - returns a result (Matloff 2009).
- Functions are a fundamental building block of R (Wickham 2015)
- Users of R should adopt the habit of creating simple functions which will make their work more effective and also more trustworthy (Chambers 2008).
- Functions are declared:
  - using the **function** reserved word
  - are objects

Understanding Computations in R

"Everything that exists is an object.
Everything that happens is a function call."

John Chambers

General Form

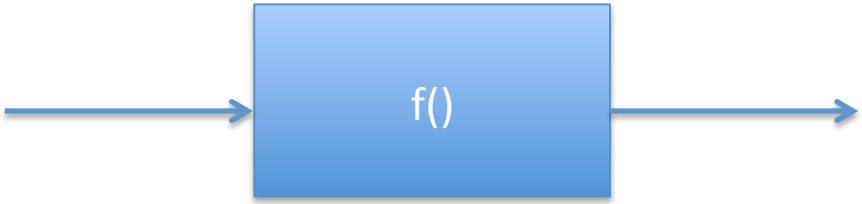
function(*arguments*)

expression

- *arguments* gives the arguments, separated by commas.
- *Expression* (body of the function) is any legal R expression, usually enclosed in { }
- **Last evaluation is returned**

Example

```
> f <- function(x){x^2}
>
> f(4)
[1] 16
```




```
> f(1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Challenge 2.1

- Write an R function that filters a vector to return all even numbers

Function Arguments

- It is useful to distinguish between *formal arguments* and the *actual arguments*
  - **Formal arguments** are the property of the function
  - **Actual arguments** can vary each time the function is called.
- When calling functions, arguments can be specified by
  - Complete name
  - Partial name
  - Position

```
f <- function(abcdef, bcde1, bcde2){
  c(a=abcdef, b1=bcde1, b2=bcde2)
}
```

```
f(1,2,3)

f(2,3,abcdef = 1)

f(2,3,a = 1)

f(2,3,b = 1)
```

```
> f(1,2,3)
 a b1 b2
 1  2  3
>
> f(2,3,abcdef = 1)
 a b1 b2
 1  2  3
>
> f(2,3,a = 1)
 a b1 b2
 1  2  3
>
> f(2,3,b = 1)
Error in f(2, 3, b = 1) : 
```

Return values

- The last expression evaluated in a function becomes the return value, the result of invoking the function
- Generally good style to reserve the use of an explicit **return()** when returning early

```
f<- function(x){
  if( x < 20){
    0
  }else {
    10
  }
}

g<- function(x){
  if( x < 20){
    return(0)
  }else {
    return(10)
  }
}
```

Challenge 7.3

Write a function that takes in a vector and returns a vector with no duplicates. Make use of the R function duplicated()

duplicated {base} R Documentation

Determine Duplicate Elements

Description

`duplicated()` determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.

`anyDuplicated(.)` is a "generalized" more efficient shortcut for `any(duplicated(.))`.

Usage

```
duplicated(x, incomparables = FALSE, ...)
```

(3) Lists in R

- Lists are different from atomic vectors because their elements can be of any type, including lists.
- list() creates a list, instead of c()

```
>
> x<- list(1:3, "a", c(T,F,T), c(2.3, 5.9))
>
> str(x)
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
```

Using c()

- c() will combine several lists into one
- c() will coerce atomic vectors to a list before combining them

```
> str(x)
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
>
> y <- c(1:3, 2:4)
>
> str(y)
 int [1:6] 1 2 3 2 3 4
```

```
> z<-c(x,y)
>
> str(z)
List of 10
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
 $ : int 1
 $ : int 2
 $ : int 3
 $ : int 2
 $ : int 3
 $ : int 4
```

Subsetting lists

- Works in the same way as subsetting an atomic vector
- Using [will always return a list
- [[and $ pull out the contents of a list
- To get the contents, you need [[

If list x is a train carrying objects, then x[[5]] is the object in car 5, x[4:6] is a train of cars 4-6" @RLangTip

Example

```
> x<- list(1:3, let="a", c(T,F,T))
>
> x
[[1]]
[1] 1 2 3

$let
[1] "a"

[[3]]
[1]  TRUE FALSE  TRUE
```

```
> x[1]
[[1]]
[1] 1 2 3

> str(x[1])
List of 1
 $ : int [1:3] 1 2 3

>
> x[[1]]
[1] 1 2 3
>
> str(x[[1]])
 int [1:3] 1 2 3
```

$ operator

- $ is a shorthand operator, where x$y is equivalent to **x[[“y”,exact=FALSE]]**
- Often used to access variables in a data frame
- $ does partial matching

```
> x
[[1]]
[1] 1 2 3

$let
[1] "a"

[[3]]
[1]  TRUE FALSE  TRUE

> x$let
[1] "a"
>
> x$l
[1] "a"
```

Challenge 7.4

- Write a function that takes in a vector of numbers and returns a list containing the:
  - Minimum
  - Maximum
  - Median
  - Standard deviation
  - Range

sapply(x,f)

- Another use of user-defined functions in R is as parameters to the *apply* family of functions.
- The general form of the **sapply(x,f,fargs)** function is as follows:
  - **x** is the target vector or list
  - **f is the function to be called and applied to each element**
  - **fargs** are the optional set of arguments that can be applied to the function f.
  - **sapply()** returns a vector, **lapply()** returns a list

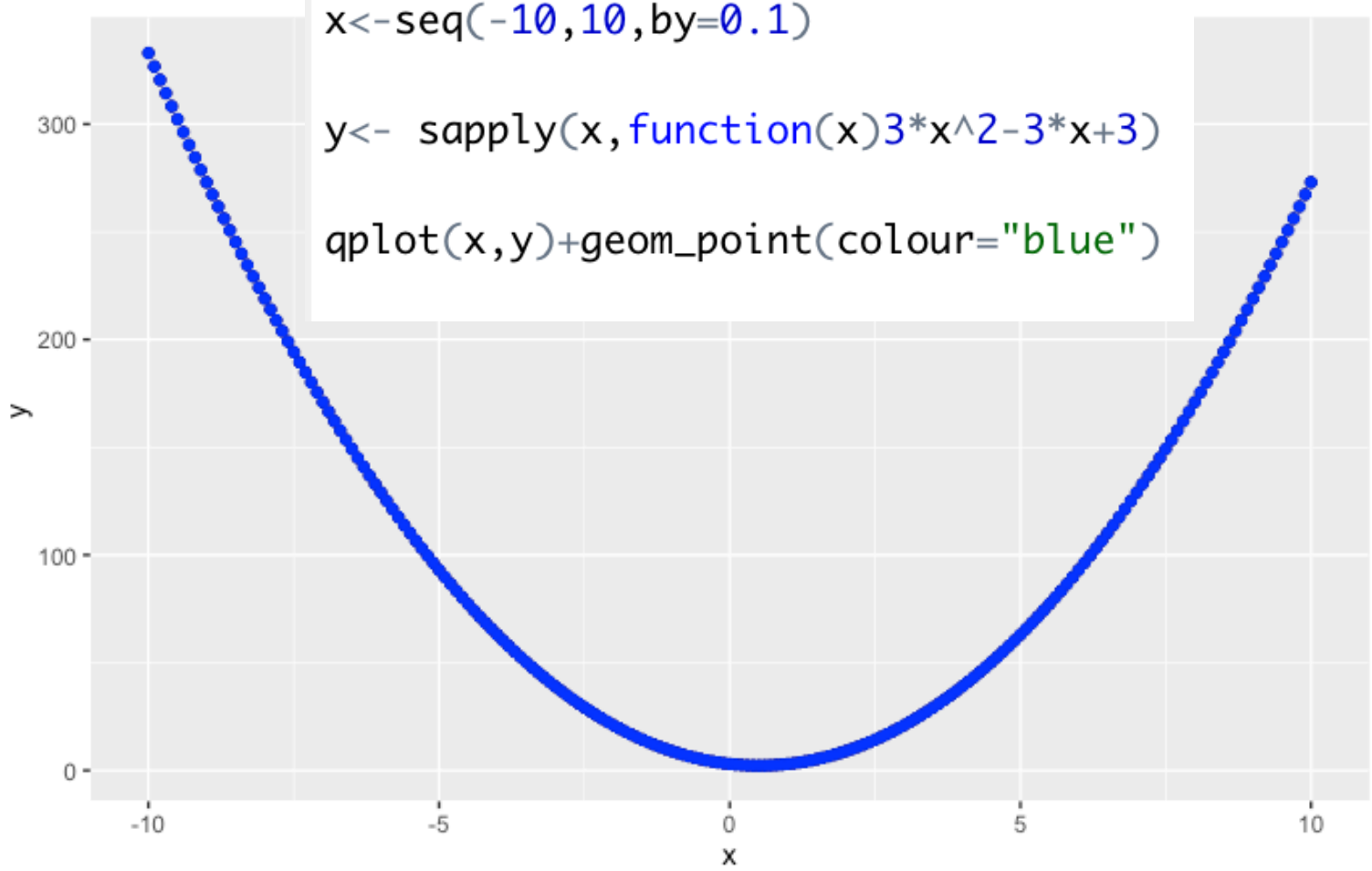
Example

```
library(ggplot2)

x<-seq(-10,10,by=0.1)

y<- sapply(x,function(x)3*x^2-3*x+3)

qplot(x,y)+geom_point(colour="blue")
```



(4) Data Frames

- The most common way of storing data in R
- Under the hood, a data frame is a list of equal-length vectors
- A two-dimensional structure, it shares properties of both a list and a matrix

	Homogenous	Heterogenous
1d	Atomic Vector	List
2d	Matrix	**Data Frame**
nd	Array	

Creating a data frame…

```
> df <- data.frame(x=1:5,y=LETTERS[1:5],stringsAsFactors=F)
>
> str(df)
'data.frame':   5 obs. of  2 variables:
 $ x: int  1 2 3 4 5
 $ y: chr  "A" "B" "C" "D" ...
```

```
> df
  x y
1 1 A
2 2 B
3 3 C
4 4 D
5 5 E
```

Logical subsetting

- Common technique for extracting rows out of a data frame

```
> str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Examples

```
> mtcars[mtcars$gear == 5,]
                mpg cyl  disp  hp drat    wt qsec vs am gear carb
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
>
> mtcars[mtcars$gear == 5 & mtcars$cyl == 8,]
                mpg cyl disp  hp drat   wt qsec vs am gear carb
Ford Pantera L 15.8   8  351 264 4.22 3.17 14.5  0  1    5    4
Maserati Bora  15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
```

Sampling from a data frame…

- Selecting n random observations from a data frame

```
> mtcars[sample(nrow(mtcars),2),]
                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
>
> mtcars[sample(nrow(mtcars),2),]
                mpg cyl  disp hp drat    wt qsec vs am gear carb
Merc 240D      24.4   4 146.7 62 3.69 3.190 20.0  1  0    4    2
Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.9  1  1    4    1
```

Adding new columns

```
> cities <- data.frame(Name=c("Dublin", "London", "Paris", "Madrid"),
+                      Population=c(553165,8673713,2244000,3141991))
>
> cities
    Name Population
1 Dublin     553165
2 London    8673713
3  Paris    2244000
4 Madrid    3141991
>
> cities$Type <- ifelse(cities$Population > 3000000,"LARGE","MEDIUM")
>
> cities
    Name Population   Type
1 Dublin     553165 MEDIUM
2 London    8673713  LARGE
3  Paris    2244000 MEDIUM
4 Madrid    3141991  LARGE
```

Reading from a Spreadsheet

Year	Leinster	Munster	Connacht	Ulster
1841	1973731	2396161	1418859	740048
1851	1672738	1857736	1010031	571052
1861	1457635	1513558	913135	517783
1871	1339451	1393485	846213	474038
1881	1278989	1331115	821657	438259
1891	1187760	1172402	724774	383758
1901	1152829	1076188	646932	345874
1911	1162044	1035495	610984	331165
1926	1149092	969902	552907	300091
1936	1220411	942272	525468	280269
1946	1281117	917306	492797	263887
1951	1336576	898870	471895	253252
1956	1338942	877238	446221	235863

```
library(gdata)

pop <- read.xls("R code/04 Matrices & DF/CensusData.xlsx")
```

pop – a data frame in R

```
> head(pop)
  Year Leinster Munster Connacht Ulster
1 1841  1973731 2396161  1418859 740048
2 1851  1672738 1857736  1010031 571052
3 1861  1457635 1513558   913135 517783
4 1871  1339451 1393485   846213 474038
5 1881  1278989 1331115   821657 438259
6 1891  1187760 1172402   724774 383758
```

Getting the total population

```
> pop$Total <- pop$Leinster+pop$Munster+pop$Connacht+pop$Ulster
>
> head(pop)
  Year Leinster Munster Connacht Ulster   Total
1 1841  1973731 2396161  1418859 740048 6528799
2 1851  1672738 1857736  1010031 571052 5111557
3 1861  1457635 1513558   913135 517783 4402111
4 1871  1339451 1393485   846213 474038 4053187
5 1881  1278989 1331115   821657 438259 3870020
6 1891  1187760 1172402   724774 383758 3468694
```

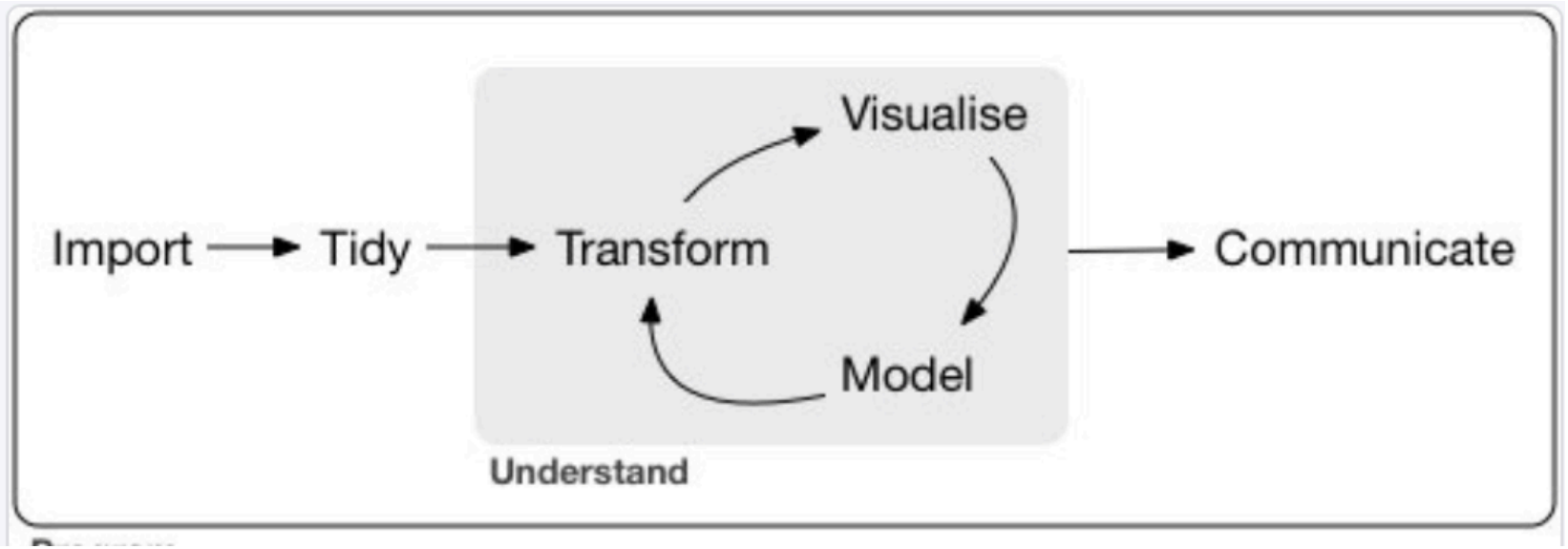
Challenge 7.5

- Given that a data frame can be manipulated using matrix notation, find another way to calculate the total population (*hint: use the apply function*)

```
> head(pop)
  Year Leinster Munster Connacht Ulster
1 1841  1973731 2396161  1418859 740048
2 1851  1672738 1857736  1010031 571052
3 1861  1457635 1513558   913135 517783
4 1871  1339451 1393485   846213 474038
5 1881  1278989 1331115   821657 438259
6 1891  1187760 1172402   724774 383758
```

(5) Visualisation (library ggplot2)

- Name based on Leland Wilkinson's *grammar of graphics,* which provides a formal, structured perspective on how to describe data graphics
- **ggplot2** package developed by Hadley Wickham



Plot components (Wickham 2016)

- Data
- A set of aesthetic mappings between variables in the data and visual properties
- At least one layer which describes how to render each observation. Layers are usually created with a **geom** function

Example (library ggplot2)

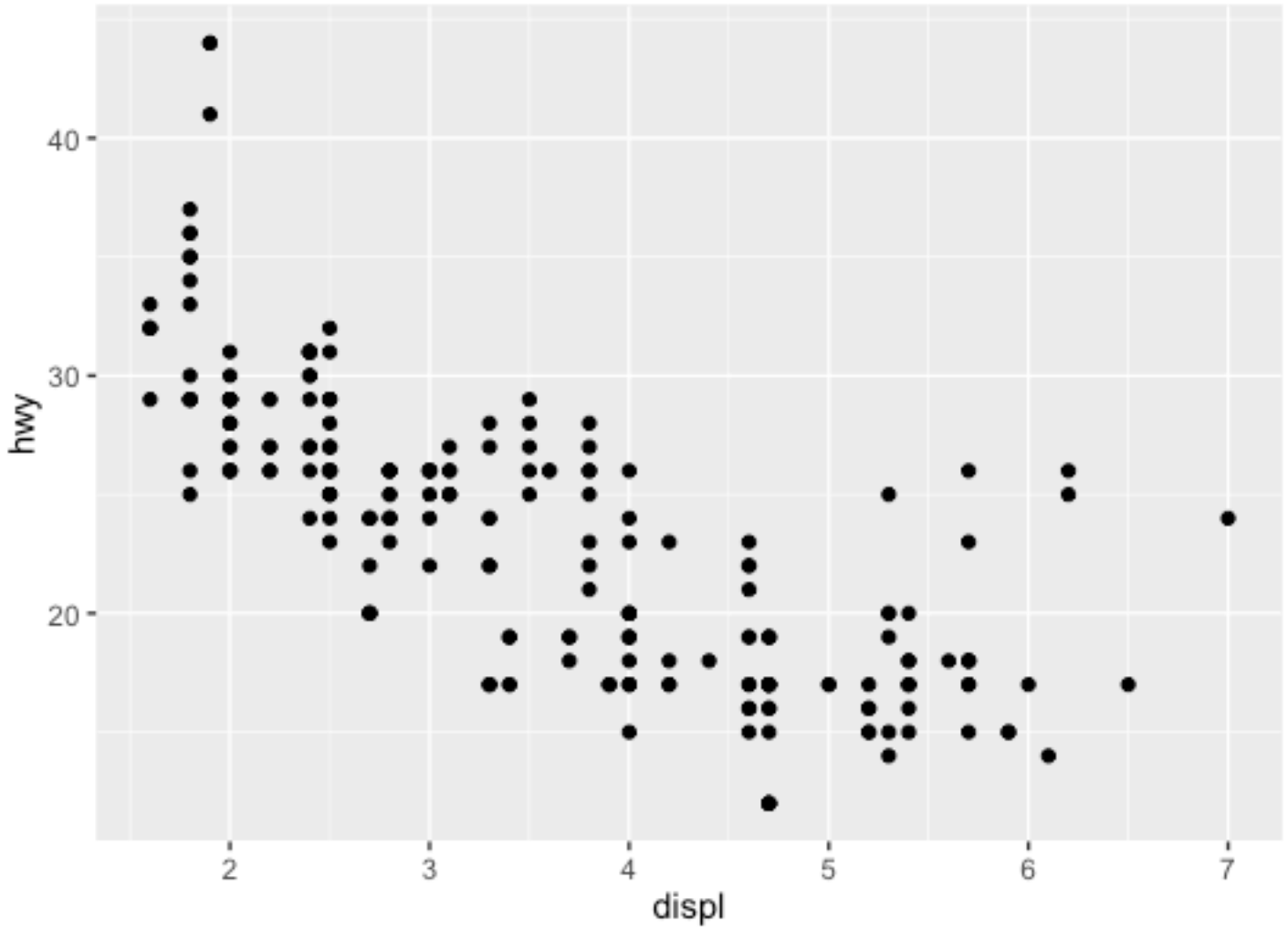
```
> mpg
# A tibble: 234 x 11
   manufacturer      model displ  year   cyl      trans   drv   cty   hwy    fl   class
          <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int> <chr>   <chr>
1          audi         a4   1.8  1999     4   auto(l5)     f    18    29     p compact
2          audi         a4   1.8  1999     4 manual(m5)     f    21    29     p compact
3          audi         a4   2.0  2008     4 manual(m6)     f    20    31     p compact
4          audi         a4   2.0  2008     4   auto(av)     f    21    30     p compact
5          audi         a4   2.8  1999     6   auto(l5)     f    16    26     p compact
6          audi         a4   2.8  1999     6 manual(m5)     f    18    26     p compact
7          audi         a4   3.1  2008     6   auto(av)     f    18    27     p compact
8          audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26     p compact
9          audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25     p compact
10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28     p compact
# ... with 224 more rows
```

DATA

AESTHETIC MAPPINGS

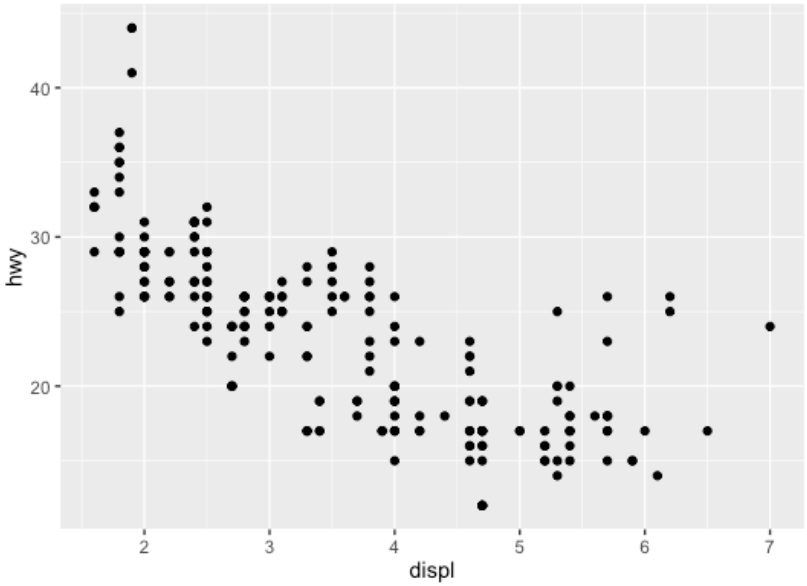
```
ggplot(mpg, aes(x=displ, y = hwy)) +
  geom_point()
```

GEOMETRY



Scatterplot structure

- Data: mpg
- Aesthetic Mapping
  - Engine size mapped to x position
  - Fuel economy to y position
- Layer: points
- Note
  - Data and aesthetics supplied in ggplot()
  - Layers added with +

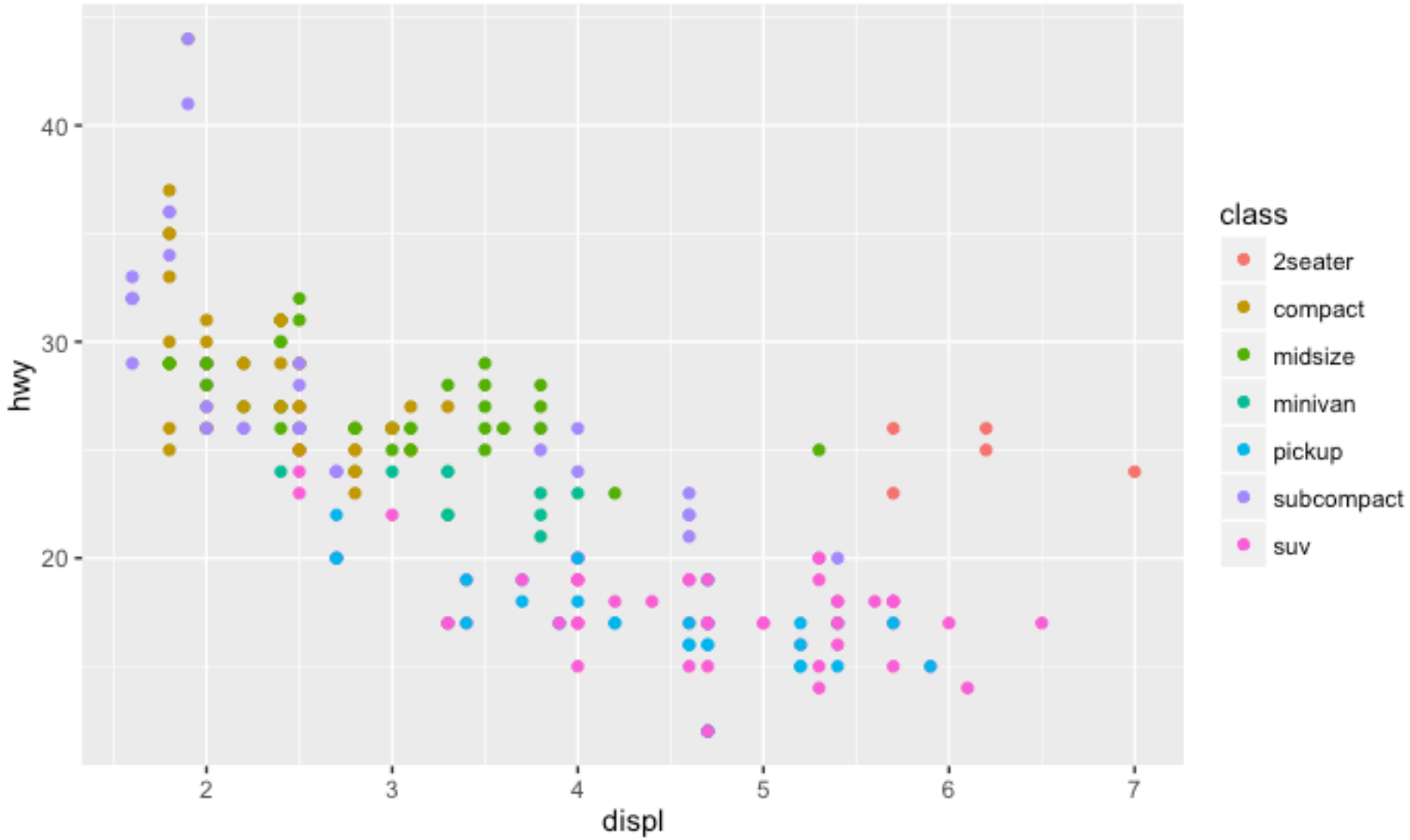



```
ggplot(mpg, aes(x=displ, y = hwy)) +
    geom_point()
```

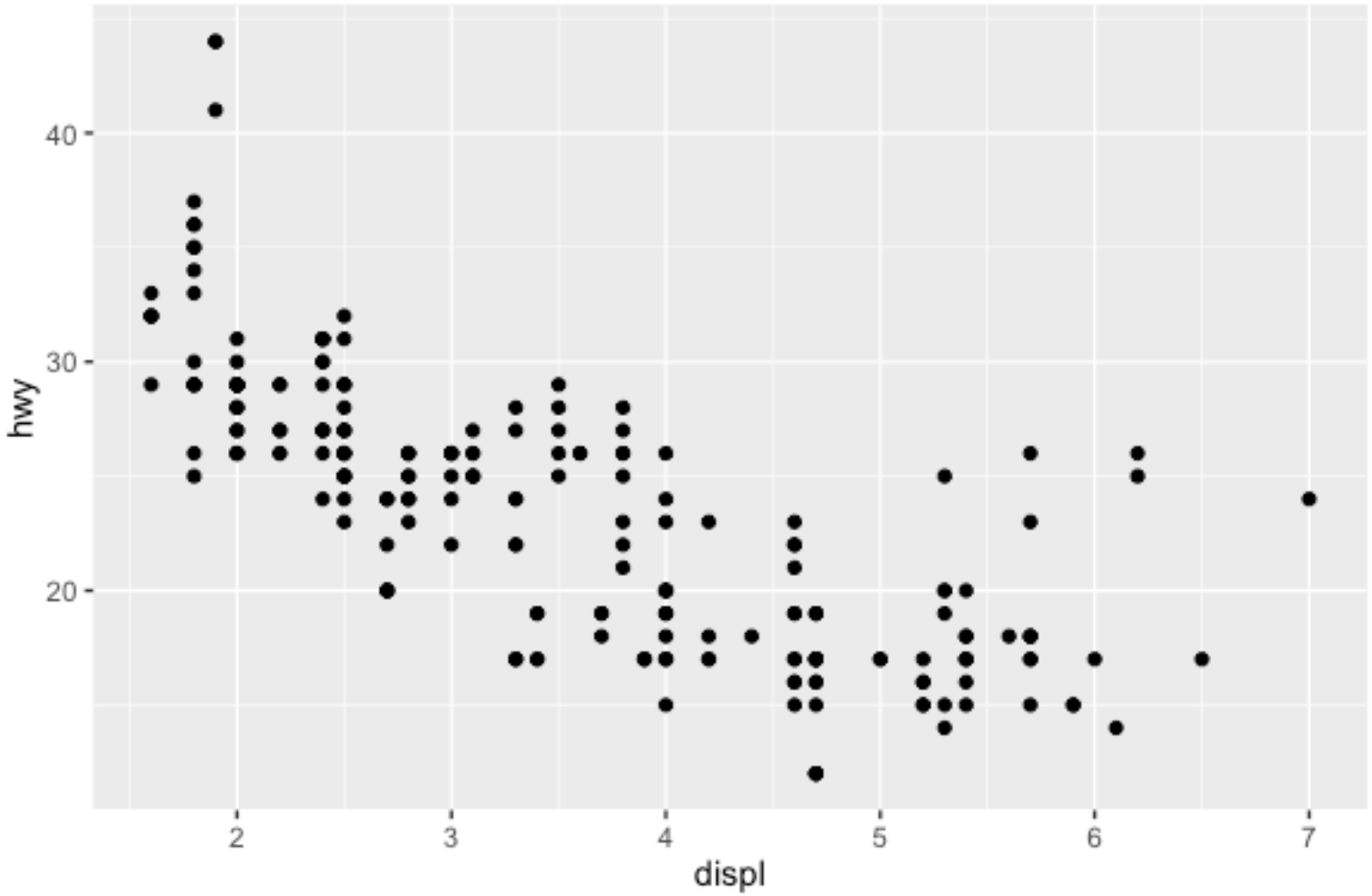
Colour, Size and Shape

- To add additional variables to a plot, other aesthetics can be used
  - Colour
  - Shape
  - Size
- Work the same way as x and y aesthetics, and are added into the call to aes

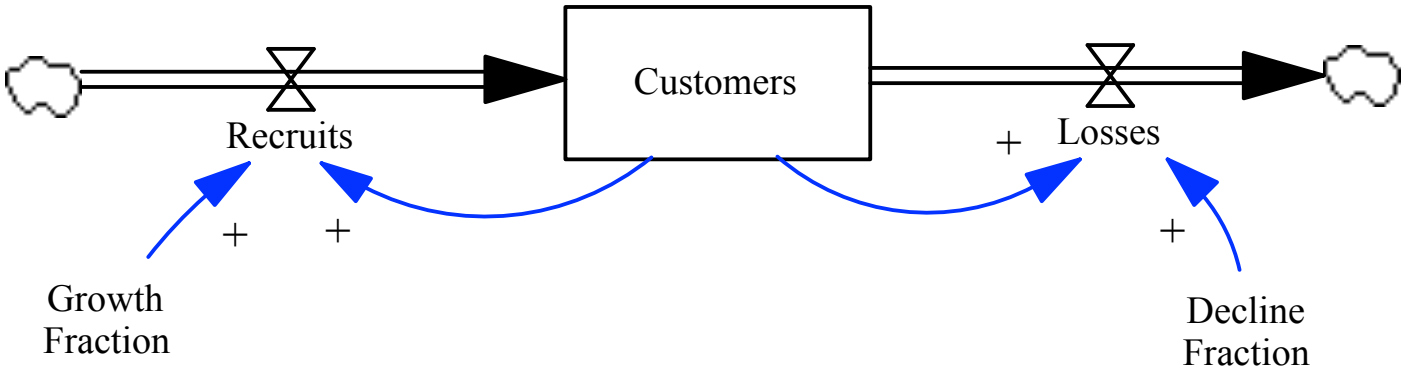
```
ggplot(mpg, aes(x=displ, y = hwy, colour=class)) +
  geom_point()
```



```
qplot(x=displ,y=hwy, data=mpg)
```



(6) Introduction to deSolve



$$Customers = INTEGRAL(Recruits - Losses, 10000)$$

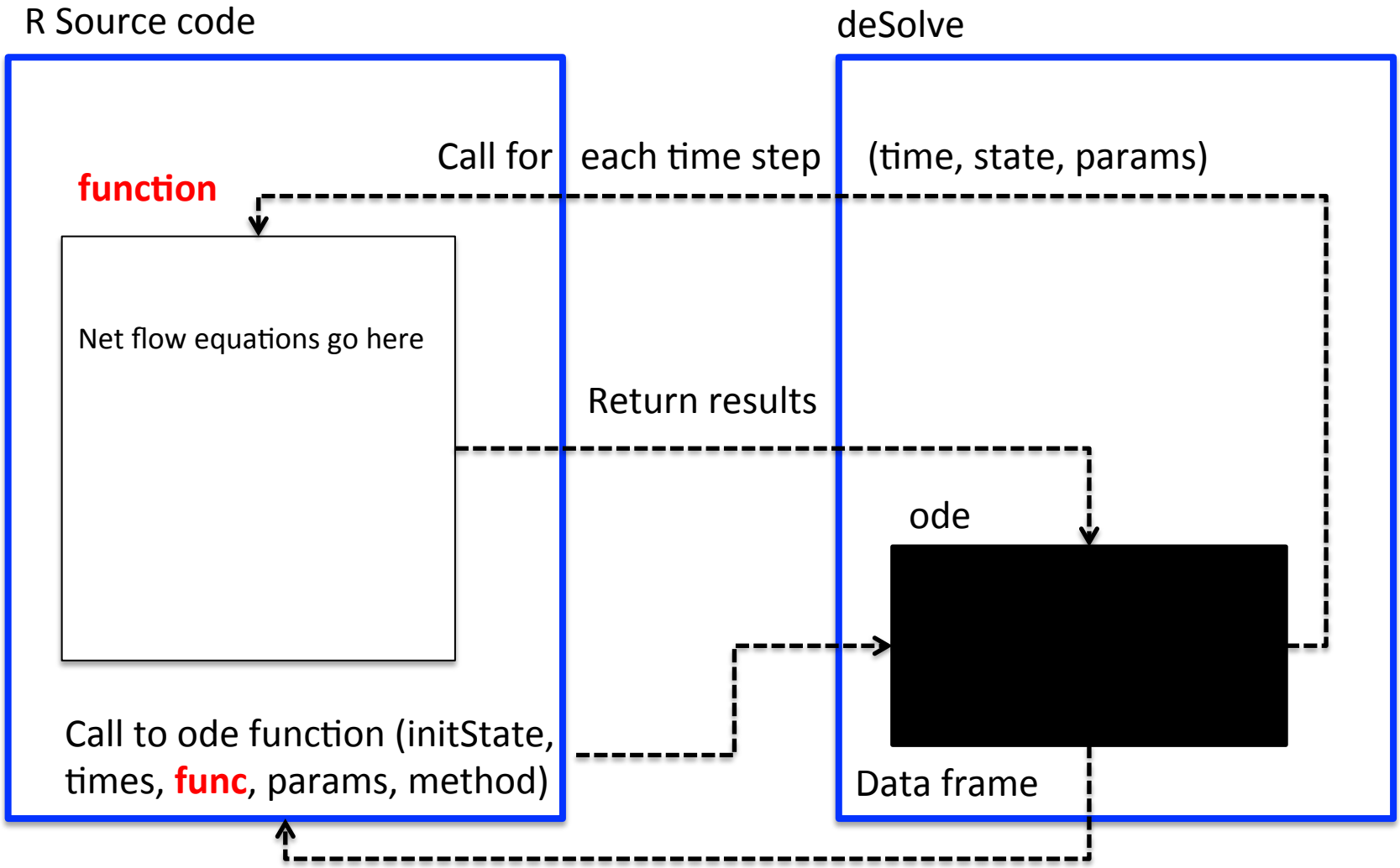
$$Recruits = Customers \times Growth\ Fraction$$

$$Losses = Customers \times Decline\ Fraction$$

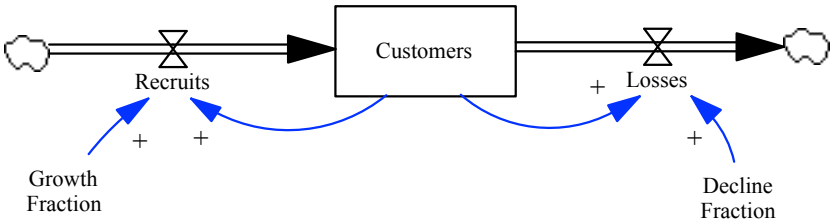
$$Growth\ Fraction = 0.07$$

$$Decline\ Fraction = 0.03$$

deSolve Approach (callbacks)



R Code (1/4)




```
library(deSolve)
library(ggplot2)

# Setup simulation times and time step
START<-2015; FINISH<-2030; STEP<-0.25

# Create time vector
simtime <- seq(START, FINISH, by=STEP)

# Create stock and auxs
stocks  <- c(sCustomers=10000)
auxs    <- c(aGrowthFraction=0.08, aDeclineFraction=0.03)
```

R Code (2/4)

```
# Model function
model <- function(time, stocks, auxs){
  with(as.list(c(stocks, auxs)),{

    fRecruits<-sCustomers*aGrowthFraction

    fLosses<-sCustomers*aDeclineFraction

    dC_dt <- fRecruits - fLosses

    return (list(c(dC_dt),
                 Recruits=fRecruits, Losses=fLosses,
                 GF=aGrowthFraction,DF=aDeclineFraction))
  })
}
```

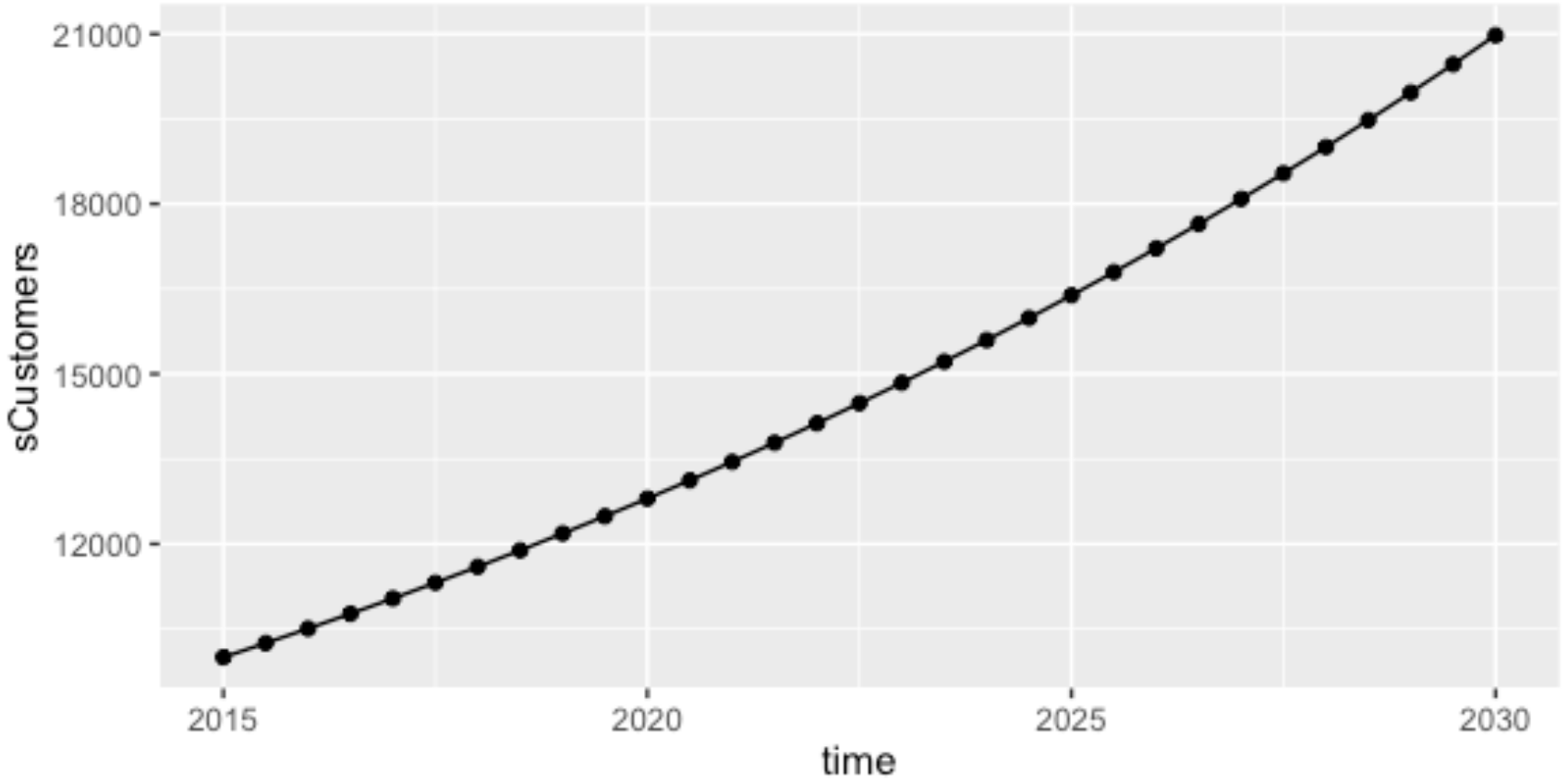
R Code (3/4)

```
# Run simulation
o<-data.frame(ode(y=stocks, times=simtime, func = model,
                  parms=auxs, method="euler"))
```

```
> o[1:10,]
    time      sCustomers Recruits  Losses     NetFlow   GF   DF
1  2015.00   10000.00    800.0000 300.0000   500.0000  0.08 0.03
2  2015.25   10125.00    810.0000 303.7500   506.2500  0.08 0.03
3  2015.50   10251.56    820.1250 307.5469   512.5781  0.08 0.03
4  2015.75   10379.71    830.3766 311.3912   518.9854  0.08 0.03
5  2016.00   10509.45    840.7563 315.2836   525.4727  0.08 0.03
6  2016.25   10640.82    851.2657 319.2246   532.0411  0.08 0.03
7  2016.50   10773.83    861.9065 323.2150   538.6916  0.08 0.03
8  2016.75   10908.50    872.6804 327.2551   545.4252  0.08 0.03
9  2017.00   11044.86    883.5889 331.3458   552.2431  0.08 0.03
10 2017.25   11182.92    894.6337 335.4877   559.1461  0.08 0.03
```

Plot results (4/4)

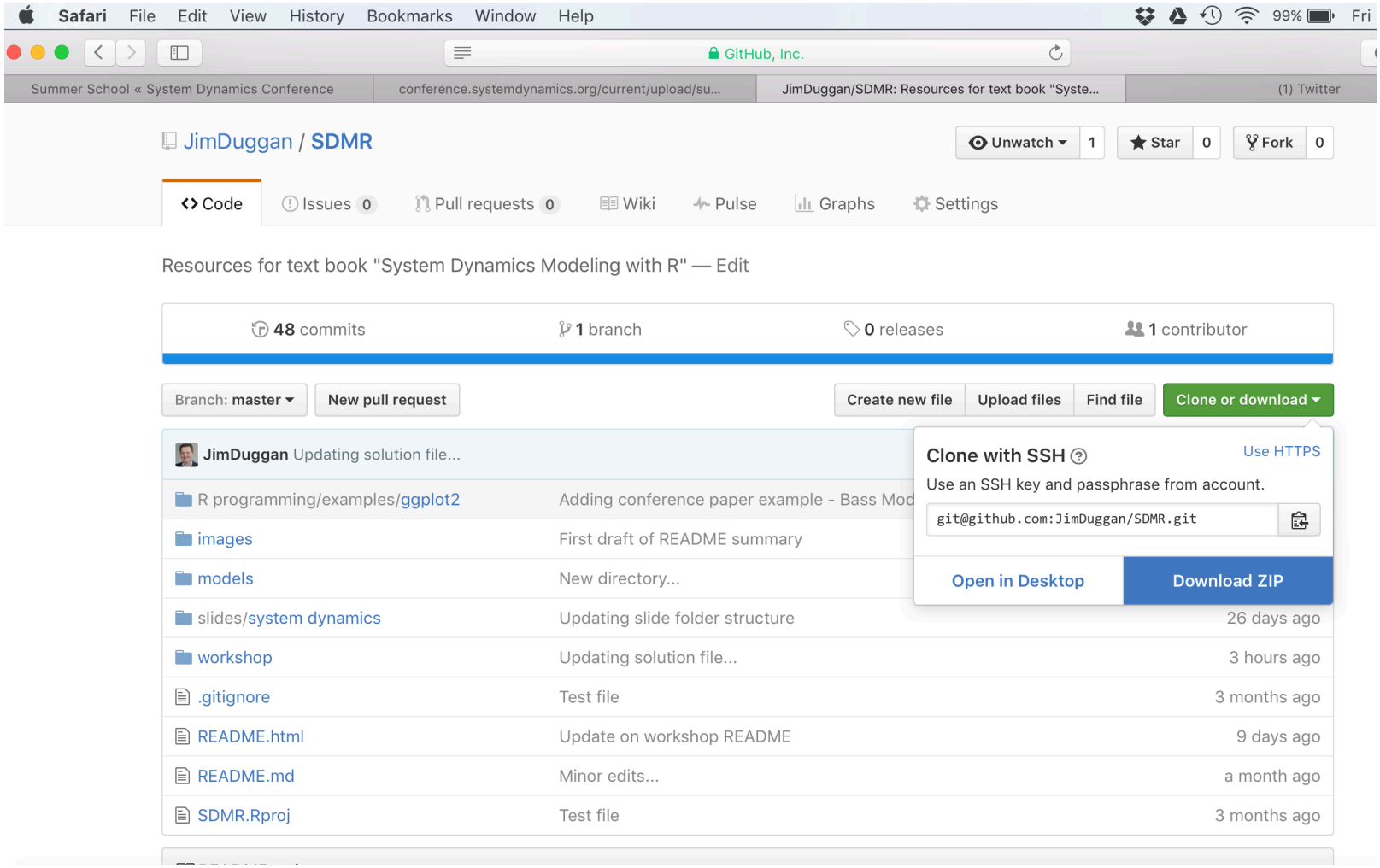
```
qplot(x=time,y=sCustomers,data=o) + geom_line()
```



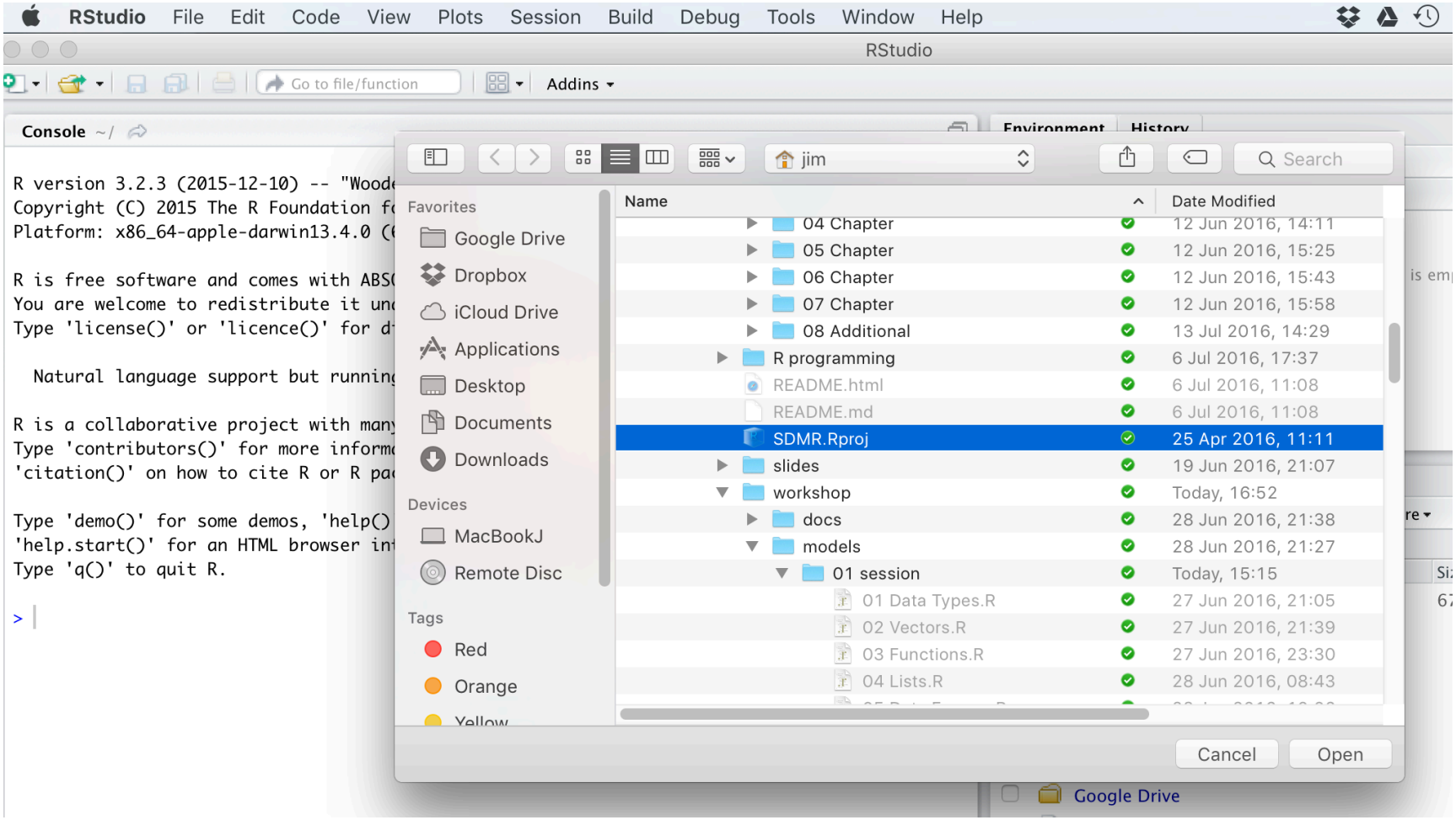
Next Steps…

First step… download materials…

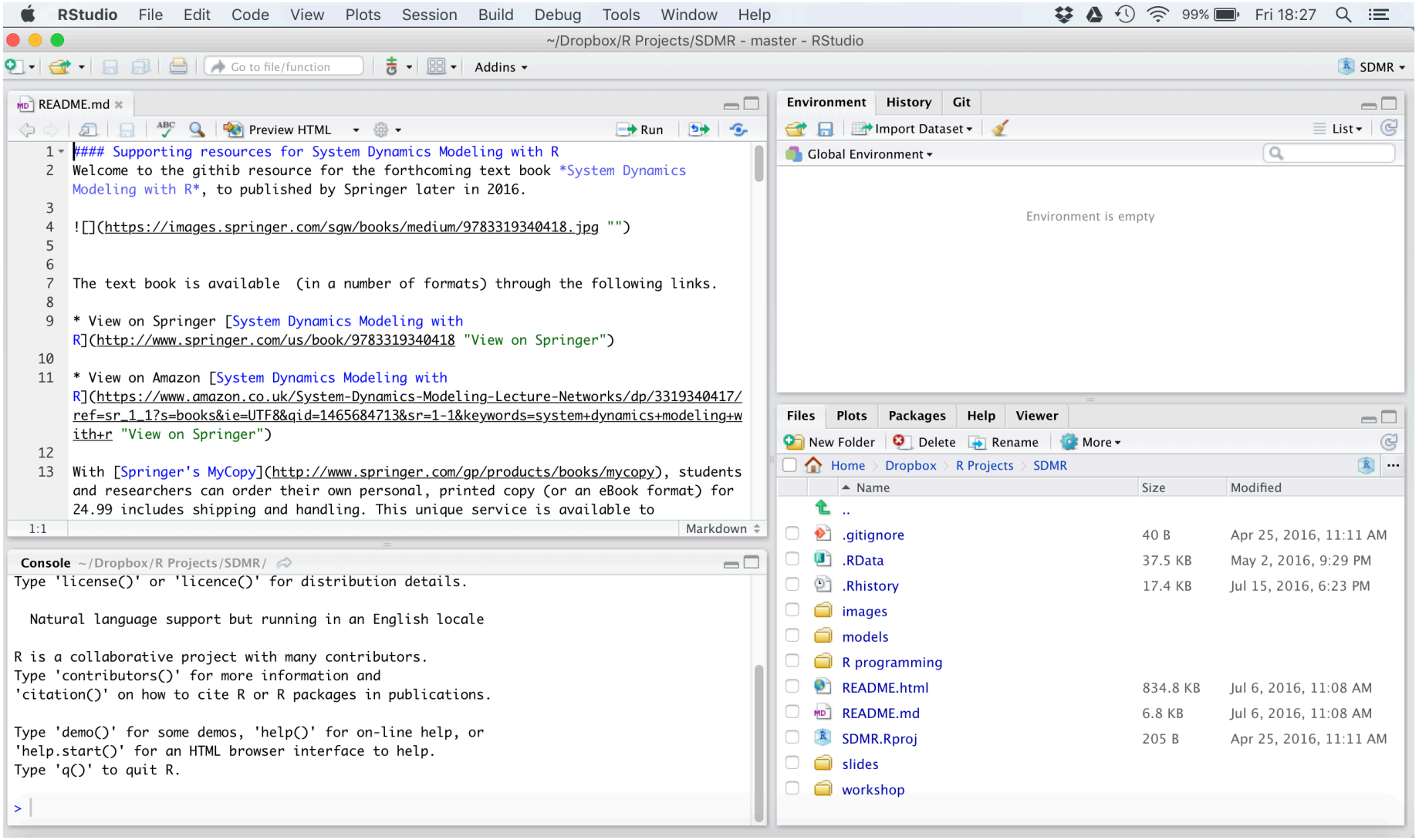
https://github.com/JimDuggan/SDMR



Open Project <SDMR.Rproj>

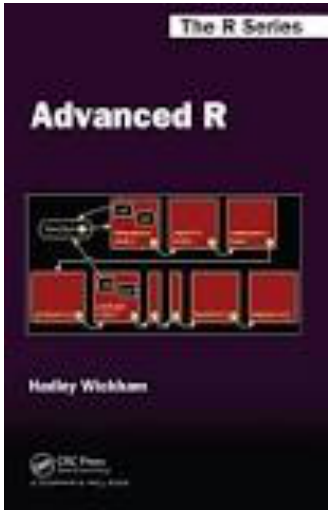


Initial Screen



References

- Wickham, H. 2015. Advanced R. Taylor & Francis



- Duggan, J. 2016. System Dynamics Modeling with R. Springer.