# Urban Transportation Planning

*Gregory Macfarlane, PhD, PE*

*2020-04-30*

# Contents

# Foreword

This book contains course notes and assignments for a senior / graduate class in transportation planning and elementary travel modeling. A description for this course is:

> An advanced course in urban transportation planning. Urban transportation as the outcome of an economic system, details and techniques for four-step travel model development, applications of travel models within a legal and regulatory context.

The book is organized into six units:

1. Building Blocks
2. Trip Generation
3. Trip Distribution
4. Mode and Destination Choice
5. Network Assignment and Validation
6. The Planning Process

It may seem strange to put the chapter covering the planning process at the end of the course, after students have learned the details of quantitative travel modeling. The purpose for this is that I assign a term project where the students build and calibrate a four-step model as they learn the techniques to do so, and then complete an alternatives analysis using their models. To create the time and space to do this project, we cover "softer" and conceptual topics in the second half of the course.

The demonstration model the students calibrate and study is a model built in the Cube travel modeling software for the Roanoke, Virginia, metropolitan region. The model is a relatively advanced four-step, trip-based model with only 250 zones. The limited zone size means that the entire model system runs in approximately 15 minutes on a laptop computer. I am grateful to Virginia DOT for allowing my students the use of this model. Directions on how to use the Roanoke model are given in the Appendices.

A handful of assignments require the students to write numerical programs or estimate statistical models. Some guidance on using R and RStudio to accomplish these assignments is also given in the Appendices.

# Chapter 1

# Building Blocks

This chapter contains concepts, definitions, and mathematical techniques that will be used throughout the semester.

## 1.1 Planning for Human Systems

If you look out on any sufficiently busy road, you will see a steady stream of vehicles passing by. Each vehicle is largely indistinguishable from the others, and it is easy as an engineer responsible for that road to see the cars driving by as little more than an input to a problem. But the *people* inside the cars should not be indistinguishable from each other. Each person who is driving or riding in each of those cars has their own reasons to be driving on that road. One person might be driving to work; one person might be trying to get home to his or her family. Another car might hold a family going on vacation, or a group of friends heading to a movie.

If you don't recognize that each person who travels is If the road is congested, then the only apparent solution is to "fix" the road by expanding its capacity.

## 1.2 Travel Model Building Blocks

### 1.2.1 Travel Analysis Zones

SE Data

### 1.2.2 Highway Networks

Functional Type

Link capacity

Free-flow speed

Centroid connectors

### 1.2.3 Matrices

Skim matrices

OD matrices

## 1.3 Statistical and Mathematical Techniques

Many elements of travel modeling and forecasting require complex numerical and quantitative techniques. In this section we will present some of these techniques.

### 1.3.1 Iterative Proportional Fitting

### 1.3.2 Regression Analysis

### 1.3.3 Numerical Optimization

## Homework

Some of these questions require a completed run of the demonstration model. For instructions on accessing and running the model, see the Appendix

1. With the TAZ layer and socioeconomic data in the demonstration model, make a set of choropleth maps showing: total households; household density; total jobs; job density; density of manufacturing vs office vs retail employment. Compare your maps with aerial imagery from Google Maps or OpenStreetMap. Describe the spatial patterns of the socioeconomic data in the model region. Identify which zones constitute the central business district, and identify any outlying employment centers.

2. With the highway network layer in the demonstration model, create maps showing: link functional type; link free flow speed; and link hourly capacity. Compare your maps with aerial imagery from Google Maps or OpenStreetMap. Identify the major freeways and principal arterials in the model region. *Note*: you will need to run the demonstration model through the network setup step to calculate the capacities and append them to the link.

3. Find the shortest free-flow speed path along the network between two zones. Find the shortest distance path between the same two zones. Are the paths the same? Do the paths match what an online mapping service shows for a trip in the middle of the night?

4. Open the highway assignment report, which shows vehicle hours and miles traveled by facility type. What percent of the region's VMT occurs on freeways? What percent of the region's lane-miles are freeways?

5. Open the output highway network. Create a map of the highway links showing PM period level of service based on the volume to capacity ratios in the table below. How would you characterize traffic in Roanoke? Which is the worst-performing major facility?

# Chapter 2

# Trip Generation

# Chapter 3

# Trip Distribution

# Chapter 4

# Mode and Destination Choice

# Chapter 5

# Network Assignment and Validation

# Chapter 6

# The Planning Process

# Appendix A

# Demonstration Model

## A.1 Running the Model

## A.2 Files and Reports

## A.3 Cube Tips and Tricks

### A.3.1 Shortest Paths

### A.3.2 Working with Matrices

### A.3.3 Writing Custom Scripts

# Appendix B

# R and RStudio Help

R is a powerful, open-source statistical programming language used by both professional and academic data scientists. It is among the computer languages most suited to modern data science, and is growing rapidly in its user base and available packages.

Some students may not feel comfortable working in a programming language like R or a console-based application like RStudio, especially if they have used applications primarily through a GUI. This appendix provides a basic bootcamp for R and Rstudio, but cannot be a comprehensive manual on RStudio, and it certainly cannot be one for R. Good places to get more detailed help include:

- R help manuals
- Stack Overflow

Some of the sections in this appendix are text-based, and some contain little more than links to YouTube videos created by me or someone else.

## B.1 Installing

There are two pieces of software you should install:

- **R** https://cran.r-project.org/: this contains the system libraries necessary to run R commands in a terminal on your computer, and contains a few additional helper applications. Install the most recent stable release for your operating system.
- **RStudio** https://rstudio.com/products/rstudio/download/ is an integrated application that makes using R considerably easier with text completion, file managment, and some GUI features.

Both software are available for Windows, MacOS, and Linux. The videos and screenshots of the application I post will use MacOS; the R code for all systems

is the same, and the RStudio interface all systems is very similar with minor differences.

## B.2 RStudio Orientation

There is a very useful "cheat sheet" for working with RStudio on the RStudio website.

## B.3 R Packages

One of the strengths of R is the ability for anyone to write packages. These packages make it easier to read manipulate, and vizualize data; to estimate statistical models; or to communicate results.

There are a number of ways to install additional packages. The most straightforward is to use the `install.packages()` function in the console. The problems in this book are solved with two additional packages[1]:

```
install.packages("tidyverse") # a suite of tools for data manipulation
install.packages("mlogit") # discrete choice modeling
```

RStudio also contains an interface to install and update packages.

[1] `tidyverse` is actually a collection of very useful packages, and many R users just load them all at once.

Sometimes you want to use a package that has not yet been pushed to CRAN, the international repository of "approved" R packages. This may be because the package is in development, or for one reason or another does not meet CRAN's standards for completeness, etc. Oftentimes, the package has been made available on GitHub. You can install a package directly from GitHub with the `devtools` library. One package you will want for the problems in the book is the `nhts2017` package on the BYU Transportation GitHub account.

```
install.packages("devtools") # tools for installing development packages
devtools::install_github("gregmacfarlane/nhts2017")
```

You only need to **install** a package once on your computer. But every time you want to **use** a function in a package, you need to load the package with the `library()` function. To load the `tidyverse` packages, for instance,

```
library(tidyverse)
```

If you get errors when you run the command above, it means that for some reason you did not install the package correctly. And if you ever get an error like

```
kable(tibble(x = 1:2, y = c("blue", "red")))
```

| x | y |
|---|---|
| 1 | blue |
| 2 | red |

It often means you didn't load the library. In this case, the `kable()` function to make pretty tables is part of the `knitr` package.

```
library(knitr)
kable(tibble(x = 1:2, y = c("blue", "red")))
```

| x | y |
|---|---|
| 1 | blue |
| 2 | red |

## B.4 Working with Tables

Most data you will work with comes in a *tabular* form, meaning that the data is formatted in columns of variables and rows of observations.

### B.4.1 Reading Data

Tabular data is often stored in a comma-separated values `.csv` file. To read a data file like this in R, you can use the `read_csv()` function included in `tidyverse`.

```
trips <- read_csv("data/demo_trips.csv")
```

```
## Parsed with column specification:
## cols(
##   houseid = col_double(),
##   personid = col_character(),
##   trpmiles = col_double(),
##   trippurp = col_character()
## )
```

```
print(trips)
```

```
## # A tibble: 924 x 4
##     houseid personid trpmiles trippurp
##       <dbl> <chr>       <dbl> <chr>
##  1 30182694 01         13.6   HBW
##  2 40532989 02          9.38  HBSOCREC
##  3 40729475 01          5.06  HBSHOP
##  4 40290784 01          0.509 HBSOCREC
##  5 30118876 02          0.599 NHB
##  6 30352119 01         20.3   HBO
##  7 30085077 01         22.5   NHB
##  8 30180962 02          0.581 HBSOCREC
##  9 40155356 01          2.74  HBO
## 10 30069734 01          4.65  NHB
## # ... with 914 more rows
```

This function will make a guess as to what the columns types should be. Often we want to keep ID values as characters, even if they are numeric (this preserves leading `0` values, etc.). We can tell `read_csv()` what types we expect with the `col_types` argument.

```
trips <- read_csv("data/demo_trips.csv", col_types = list(houseid = col_character()))
print(trips)
```

```
## # A tibble: 924 x 4
##    houseid  personid trpmiles trippurp
##    <chr>    <chr>       <dbl> <chr>
##  1 30182694 01         13.6   HBW
##  2 40532989 02          9.38  HBSOCREC
##  3 40729475 01          5.06  HBSHOP
##  4 40290784 01          0.509 HBSOCREC
##  5 30118876 02          0.599 NHB
##  6 30352119 01         20.3   HBO
##  7 30085077 01         22.5   NHB
##  8 30180962 02          0.581 HBSOCREC
##  9 40155356 01          2.74  HBO
```

```
## 10 30069734 01          4.65  NHB
## # ... with 914 more rows
```

You can also write tables back to `.csv` with the `write_csv()` command.

### B.4.2 Modifying and Summarizing Tables

In much of this section, we will work with the `nhts_trips` dataset of trips from the 2017 National Household Travel Survey in the `nhts2017` package you installed from GitHub above.

```
library(nhts2017)
trips <- nhts_trips
trips
```

```
## # A tibble: 923,572 x 62
##    houseid personid tdtrpnum strttime            endtime
##    <chr>   <chr>       <dbl> <dttm>              <dttm>
##  1 300000~ 01              1 2017-10-10 10:00:00 2017-10-10 10:15:00
##  2 300000~ 01              2 2017-10-10 15:10:00 2017-10-10 15:30:00
##  3 300000~ 02              1 2017-10-10 07:00:00 2017-10-10 09:00:00
##  4 300000~ 02              2 2017-10-10 18:00:00 2017-10-10 20:30:00
##  5 300000~ 03              1 2017-10-10 08:45:00 2017-10-10 09:00:00
##  6 300000~ 03              2 2017-10-10 14:30:00 2017-10-10 14:45:00
##  7 300000~ 01              1 2017-10-10 11:15:00 2017-10-10 11:30:00
##  8 300000~ 01              2 2017-10-10 23:30:00 2017-10-10 23:40:00
##  9 300000~ 01              1 2017-10-10 05:50:00 2017-10-10 06:05:00
## 10 300000~ 01              2 2017-10-10 07:00:00 2017-10-10 07:15:00
## # ... with 923,562 more rows, and 57 more variables: trvlcmin <dbl+lbl>,
## #   trpmiles <dbl+lbl>, trptrans <chr+lbl>, trpaccmp <dbl+lbl>,
## #   trphhacc <dbl+lbl>, vehid <chr+lbl>, trwaittm <dbl+lbl>,
## #   numtrans <dbl+lbl>, tracctm <dbl+lbl>, drop_prk <chr+lbl>,
## #   tregrtm <dbl+lbl>, whodrove <chr+lbl>, whyfrom <chr+lbl>,
## #   loop_trip <chr+lbl>, trphhveh <chr+lbl>, hhmemdrv <chr+lbl>,
## #   hh_ontd <dbl+lbl>, nonhhcnt <dbl+lbl>, numontrp <dbl+lbl>,
## #   psgr_flg <chr+lbl>, pubtrans <chr+lbl>, trippurp <chr+lbl>,
## #   dweltime <dbl+lbl>, tdwknd <chr+lbl>, vmt_mile <dbl+lbl>,
## #   drvr_flg <chr+lbl>, whytrp1s <chr+lbl>, ontd_p1 <chr+lbl>,
## #   ontd_p2 <chr+lbl>, ontd_p3 <chr+lbl>, ontd_p4 <chr+lbl>,
## #   ontd_p5 <chr+lbl>, ontd_p6 <chr+lbl>, ontd_p7 <chr+lbl>,
## #   ontd_p8 <chr+lbl>, ontd_p9 <chr+lbl>, ontd_p10 <chr+lbl>,
## #   ontd_p11 <chr+lbl>, ontd_p12 <chr+lbl>, ontd_p13 <chr+lbl>,
## #   tdcaseid <chr>, tracc_wlk <chr+lbl>, tracc_pov <chr+lbl>,
## #   tracc_bus <chr+lbl>, tracc_crl <chr+lbl>, tracc_sub <chr+lbl>,
## #   tracc_oth <chr+lbl>, tregr_wlk <chr+lbl>, tregr_pov <chr+lbl>,
## #   tregr_bus <chr+lbl>, tregr_crl <chr+lbl>, tregr_sub <chr+lbl>,
## #   tregr_oth <chr+lbl>, whyto <chr+lbl>, gasprice <chr>, wttrdfin <dbl>,
```

```
## #   whytrp90 <chr+lbl>
```

#### B.4.2.1 Select, Filter, and Chains

This table is pretty overwhelming. But there are two functions that can help us pare it down:

- `select()` lets you select columns in a table using the names of the columns.
- `filter()` lets you select rows in a table that meet a certain condition.

Let's practice this by selecting our `trips` dataset to only include the id columns, the trip length, and the trip purpose.

```
select(trips, houseid, personid, trpmiles, trippurp)
```

```
## # A tibble: 923,572 x 4
##    houseid  personid  trpmiles trippurp
##    <chr>    <chr>    <dbl+lbl> <chr+lbl>
##  1 30000007 01            5.24 HBO [Home-based trip (other)]
##  2 30000007 01            5.15 HBO [Home-based trip (other)]
##  3 30000007 02           84.0  HBW [Home-based trip (work)]
##  4 30000007 02           81.6  HBW [Home-based trip (work)]
##  5 30000007 03            2.25 HBO [Home-based trip (other)]
##  6 30000007 03            2.24 HBO [Home-based trip (other)]
##  7 30000008 01            8.02 HBW [Home-based trip (work)]
##  8 30000008 01            8.02 HBW [Home-based trip (work)]
##  9 30000012 01            3.40 HBSOCREC [Home-based trip (social/recreatio~
## 10 30000012 01            3.40 HBSOCREC [Home-based trip (social/recreatio~
## # ... with 923,562 more rows
```

Let's also practice filtering the `trips` dataset to only include trips of the purpose "HBO" (home-based other). Notice how the number of rows in the table trips is much smaller.

```
filter(trips, trippurp == "HBW") # use double equals as comparison
```

```
## # A tibble: 117,368 x 62
##    houseid personid tdtrpnum strttime            endtime
##    <chr>   <chr>       <dbl> <dttm>              <dttm>
##  1 300000~ 02              1 2017-10-10 07:00:00 2017-10-10 09:00:00
##  2 300000~ 02              2 2017-10-10 18:00:00 2017-10-10 20:30:00
##  3 300000~ 01              1 2017-10-10 11:15:00 2017-10-10 11:30:00
##  4 300000~ 01              2 2017-10-10 23:30:00 2017-10-10 23:40:00
##  5 300000~ 01              5 2017-10-10 09:00:00 2017-10-10 09:20:00
##  6 300000~ 01              7 2017-10-10 15:30:00 2017-10-10 16:05:00
##  7 300000~ 01              1 2017-10-10 08:00:00 2017-10-10 08:20:00
##  8 300000~ 01              2 2017-10-10 18:00:00 2017-10-10 20:00:00
##  9 300000~ 02              3 2017-10-10 09:00:00 2017-10-10 11:00:00
```

```
## 10 300000~ 02              4 2017-10-10 18:30:00 2017-10-10 20:30:00
## # ... with 117,358 more rows, and 57 more variables: trvlcmin <dbl+lbl>,
## #   trpmiles <dbl+lbl>, trptrans <chr+lbl>, trpaccmp <dbl+lbl>,
## #   trphhacc <dbl+lbl>, vehid <chr+lbl>, trwaittm <dbl+lbl>,
## #   numtrans <dbl+lbl>, tracctm <dbl+lbl>, drop_prk <chr+lbl>,
## #   tregrtm <dbl+lbl>, whodrove <chr+lbl>, whyfrom <chr+lbl>,
## #   loop_trip <chr+lbl>, trphhveh <chr+lbl>, hhmemdrv <chr+lbl>,
## #   hh_ontd <dbl+lbl>, nonhhcnt <dbl+lbl>, numontrp <dbl+lbl>,
## #   psgr_flg <chr+lbl>, pubtrans <chr+lbl>, trippurp <chr+lbl>,
## #   dweltime <dbl+lbl>, tdwknd <chr+lbl>, vmt_mile <dbl+lbl>,
## #   drvr_flg <chr+lbl>, whytrp1s <chr+lbl>, ontd_p1 <chr+lbl>,
## #   ontd_p2 <chr+lbl>, ontd_p3 <chr+lbl>, ontd_p4 <chr+lbl>,
## #   ontd_p5 <chr+lbl>, ontd_p6 <chr+lbl>, ontd_p7 <chr+lbl>,
## #   ontd_p8 <chr+lbl>, ontd_p9 <chr+lbl>, ontd_p10 <chr+lbl>,
## #   ontd_p11 <chr+lbl>, ontd_p12 <chr+lbl>, ontd_p13 <chr+lbl>,
## #   tdcaseid <chr>, tracc_wlk <chr+lbl>, tracc_pov <chr+lbl>,
## #   tracc_bus <chr+lbl>, tracc_crl <chr+lbl>, tracc_sub <chr+lbl>,
## #   tracc_oth <chr+lbl>, tregr_wlk <chr+lbl>, tregr_pov <chr+lbl>,
## #   tregr_bus <chr+lbl>, tregr_crl <chr+lbl>, tregr_sub <chr+lbl>,
## #   tregr_oth <chr+lbl>, whyto <chr+lbl>, gasprice <chr>, wttrdfin <dbl>,
## #   whytrp90 <chr+lbl>
```

One ***extremely*** useful feature of the `tidyverse` functions is the chain operator, `%>%`. This operator basically does the opposite of the assigment operator `<-`. While assignment says "take the thing on the right and put it in the thing on the left," chain says "take the thing on the left and pass it as the first argument of the function on the right." What this means in practice is we can chain R commands together. So we can do the `select` *and* the `filter` statements in sequence,

```
trips %>%
  select(houseid, personid, trpmiles, trippurp) %>%
  filter(trippurp == "HBW")
```

```
## # A tibble: 117,368 x 4
##    houseid  personid  trpmiles trippurp
##    <chr>    <chr>    <dbl+lbl> <chr+lbl>
##  1 30000007 02           84.0  HBW [Home-based trip (work)]
##  2 30000007 02           81.6  HBW [Home-based trip (work)]
##  3 30000008 01            8.02 HBW [Home-based trip (work)]
##  4 30000008 01            8.02 HBW [Home-based trip (work)]
##  5 30000012 01            4.29 HBW [Home-based trip (work)]
##  6 30000012 01            6.82 HBW [Home-based trip (work)]
##  7 30000039 01           11.5  HBW [Home-based trip (work)]
##  8 30000041 01           73.7  HBW [Home-based trip (work)]
##  9 30000041 02           77.9  HBW [Home-based trip (work)]
## 10 30000041 02           77.8  HBW [Home-based trip (work)]
```

```
## # ... with 117,358 more rows
```

Notice that we didn't have to tell the `select` and `filter` functions the name of the table we were selecting or filtering. The `%>%` chain operator did that for us.

Once we have the table we want, we can assign it to a new object called `mytrips` In this case, let's get `HBO` and `HBW` trips.

```
mytrips <- trips %>%
  select(houseid, personid, trpmiles, trippurp) %>%
  filter(trippurp %in% c("HBW", "HBO")) # use %in% for multiple comparisons.
```

### B.4.3 Mutate, Summarize, and Group

Sometimes we want to calculate a new column in a table, or recompute an existing column. We can do that with the `mutate` function, and we can put more than one calculation in a single `mutate` statement.

```
mytrips %>%
  mutate(
    tripkm = trpmiles * 1.60934, # convert miles to km.
    longtrip = ifelse(tripkm > 50, TRUE, FALSE) # is trip longer than 50 km?
  )
```

```
## # A tibble: 307,390 x 6
##    houseid  personid  trpmiles trippurp                    tripkm longtrip
##    <chr>    <chr>    <dbl+lbl> <chr+lbl>                  <dbl+lb> <lgl>
##  1 30000007 01            5.24 HBO [Home-based trip (oth~     8.44 FALSE
##  2 30000007 01            5.15 HBO [Home-based trip (oth~     8.29 FALSE
##  3 30000007 02           84.0  HBW [Home-based trip (wor~   135.   TRUE
##  4 30000007 02           81.6  HBW [Home-based trip (wor~   131.   TRUE
##  5 30000007 03            2.25 HBO [Home-based trip (oth~     3.62 FALSE
##  6 30000007 03            2.24 HBO [Home-based trip (oth~     3.61 FALSE
##  7 30000008 01            8.02 HBW [Home-based trip (wor~    12.9  FALSE
##  8 30000008 01            8.02 HBW [Home-based trip (wor~    12.9  FALSE
##  9 30000012 01            4.29 HBW [Home-based trip (wor~     6.91 FALSE
## 10 30000012 01            6.82 HBW [Home-based trip (wor~    11.0  FALSE
## # ... with 307,380 more rows
```

Other times we want to calculate summary statistics like means. For this we can use the `summarize()` function.

```
mytrips %>%
  summarize(
    mean_trip = mean(trpmiles),
    sd_trip = sd(trpmiles),
    max_trip = max(trpmiles),
```

```
    min_trip = min(trpmiles)
  )
```

```
## # A tibble: 1 x 4
##   mean_trip sd_trip max_trip min_trip
##       <dbl>   <dbl>    <dbl>    <dbl>
## 1      9.81    32.1    5699.       -9
```

Finally, we sometimes want to calculate summary statistics for different groups. We can tell `tidyverse` to group our tables with the `group_by()` function.

```
mytrips %>%
  group_by(trippurp) %>%
  summarize(
    mean_trip = mean(trpmiles),
    sd_trip = sd(trpmiles),
    max_trip = max(trpmiles),
    min_trip = min(trpmiles)
  )
```

```
## # A tibble: 2 x 5
##   trippurp                       mean_trip sd_trip max_trip min_trip
## * <chr+lbl>                          <dbl>   <dbl>    <dbl>    <dbl>
## 1 HBO [Home-based trip (other)]       7.73    32.0    5699.       -9
## 2 HBW [Home-based trip (work)]       13.2     31.9    2927.       -9
```

As you might expect, work trips are on average longer than other kinds of trips. But some people report very long trips! You might want to filter your data more carefully for real analyses.

## B.5 Graphics with ggplot