ITHIM CALIFORNIA | Integrated Transport and Health Impact Model

California ITHIM Tool Documentation

California R/Shiny Version

by

University of California, Davis

Kenji Tomari, MS
Neil Maizlish, PhD
Jonathan London, PhD

and

Consultants

Chengsheng Jiang, PhD
Amy Weiher, MA

October 17, 2019

UC DAVIS UNIVERSITY OF CALIFORNIA

CALIFORNIA AIR RESOURCES BOARD

Table of Contents

1. Introduction

The documentation for the California R/Shiny ITHIM has two main components:

1. User's Guide and Technical Manual
2. ITHIM TOOL

The *User's Guide and Technical Manual* describes the conceptual basis of the ITHIM model, web site navigation, data structures for calibration and built-in scenarios, and the R programming of the non-interactive web pages (Home, About, Decision Support, User Support), and the portion of the RunITHIM page above the ITHIM TOOL title. R programmers should read Chapter 3 of the *User's Guide* to get an orientation to the overall structure of the R/Shiny code of the ITHIM website before delving into the ITHIM TOOL.

The ITHIM TOOL is the subject of this document, which provides details of the R code that runs the ITHIM model as an interactive application. The R/Shiny code for the ITHIM TOOL generates the portion of the RunITHIM web page below the ITHIM TOOL title.

Organizing the website into interactive and non-interactive pages allowed the development team at University of California, Davis to divide up the programming tasks without creating potential code conflicts between or within web pages. This separation of programming and web pages has additional advantages:

1. Prevents inadvertent changes in the code of the interactive application when performing updating the content of the non-interactive pages, and
2. Facilitates a development process for customization (for other states or countries) in which the ITHIM TOOL can be embedded in a different website presentation, and visa versa.

Throughout this document, the terms **ITHIM TOOL** and **ITHIM module** are used interchangeably. "Module" refers to the R and shiny code that implements the ITHIM model as an interactive web application. Further, this module fits inside the California ITHIM website. In the following documentation, curly braces are used to denote an R package such as {shiny}. Terms in italics indicate specific objects or commands in R or an R package.

This document does not cover the logic or design of the ITHIM model (e.g. the comparative risk assessment or the data sources). The ITHIM model, data, and methods are summarized in the About>Introduction and About> Methods webpages as well as in the User's Guide and Technical Manual.

This documentation assumes that the reader has a fundamental understanding of the R language, including:

- Basic syntax
- Variables and data types (with a particular emphasis on *vectors*, *data.frames* and *lists*)
- Operators
- Decision Making (e.g., if-else statements)
- Loops

- Functions
- Regular expressions (regex for short). (While the {stringr} package is not utilized here, the regex section of the {stringr} cheat sheet is a good resource. The majority of the regular expressions in this code (and indeed R) uses the conventions established by the *Perl language*.)
- Packages
    - Most packages will be described as they are encountered in the documentation. However, {roxygen2} is used throughout the documentation. The {roxygen2} notation format is implemented in the code before the function definitions in the TOOL. These notations are just an additional form of documentation, they simply describe the function in *# comments* next to the function definition.
- Data reshaping (particularly the *merge* function.)

In addition to basic R language concepts, we will encounter a few more advanced topics, still drawing from the {base} package of R. These include *lapply*, *sapply*, and *mapply*. These {base} functionals operate much like a loop, iteratively applying some function over a *list* or *vector*. Readers should also understand the concept of attributes. In short, attributes allow R objects (e.g. variables) to be assigned name-value pairs that attach metadata. Attributes are also discussed in some length in the chapter 5 of this document. Readers should also have a strong understanding of how the package {shiny} works, especially the concept of "reactivity". Beyond the R language, readers should have some familiarity with HTML (hypertext markup language) and CSS (cascading style sheets), although expertise in these languages is not critical.

To learn more about advanced technical components, such as the functions and packages, we will provide hyperlinks to explanatory materials.

Readers are advised that as they read this documentation they can open the relevant R code and follow along. This is particularly helpful for chapters discussing specific R script files. For instance, as we review the *visualization.R* script, readers should open this file and briefly preview each function group in the code before reading the summary of the function group in documentation.

Chapter References

- *R for Data Science* is an excellent book on data science by Garrett Grolemund & Hadley Wickham (albeit utilizing a different suite of R packages). Notably, it has a strong introduction to the package {ggplot2}
- Advanced R by Hadley Wickham has key sections with respect to this project, including attributes, functionals, and non-standard evaluation.
- TutorialsPoint is a free website to learn the basics of R.

2. Overview

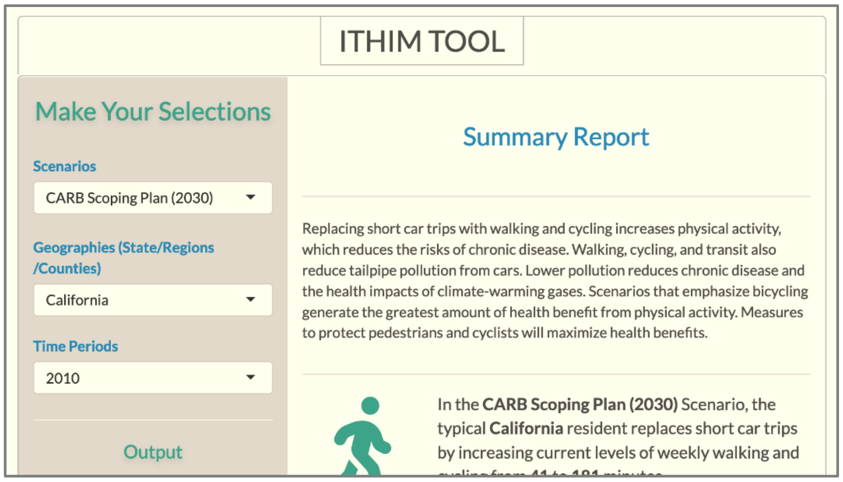
This chapter presents a brief overview of the main components of the ITHIM TOOL.[1] A schematic overview is presented on the next page (Figure 2.1).

2.1. File Organization

The file structure of the ITHIM TOOL is divided into two *.R* files in the "root folder" alongside two sub-folders (depicted below). The *.R* files include the *app.R*[2] and *global.R*, and the sub-folders include the *tool_files* and *www*.

```
root_folder/
  app.R
  global.R
  tool_files/
    ithim_tool.css
    log.txt
    (Various .png files)
    (Various .R files)
    (Various .csv files)
  www/
    bootstrap2.css
    (Various .png files beginning with the word "icon_")
```

The diagram (Figure 2.1) shows the files necessary to make the ITHIM TOOL run properly apart from the overall website. Several files necessary for the entire ITHIM website to function properly are not shown (e.g., webtext.csv, webphotoimage.csv, counties1col.csv. regions.csv). If we want to run the ITHIM TOOL by itself (Fig. 2.2), we would only need the file structure displayed above and a modified version of the *app.R* file without {shiny} code for the non-interactive pages or website page navigation.



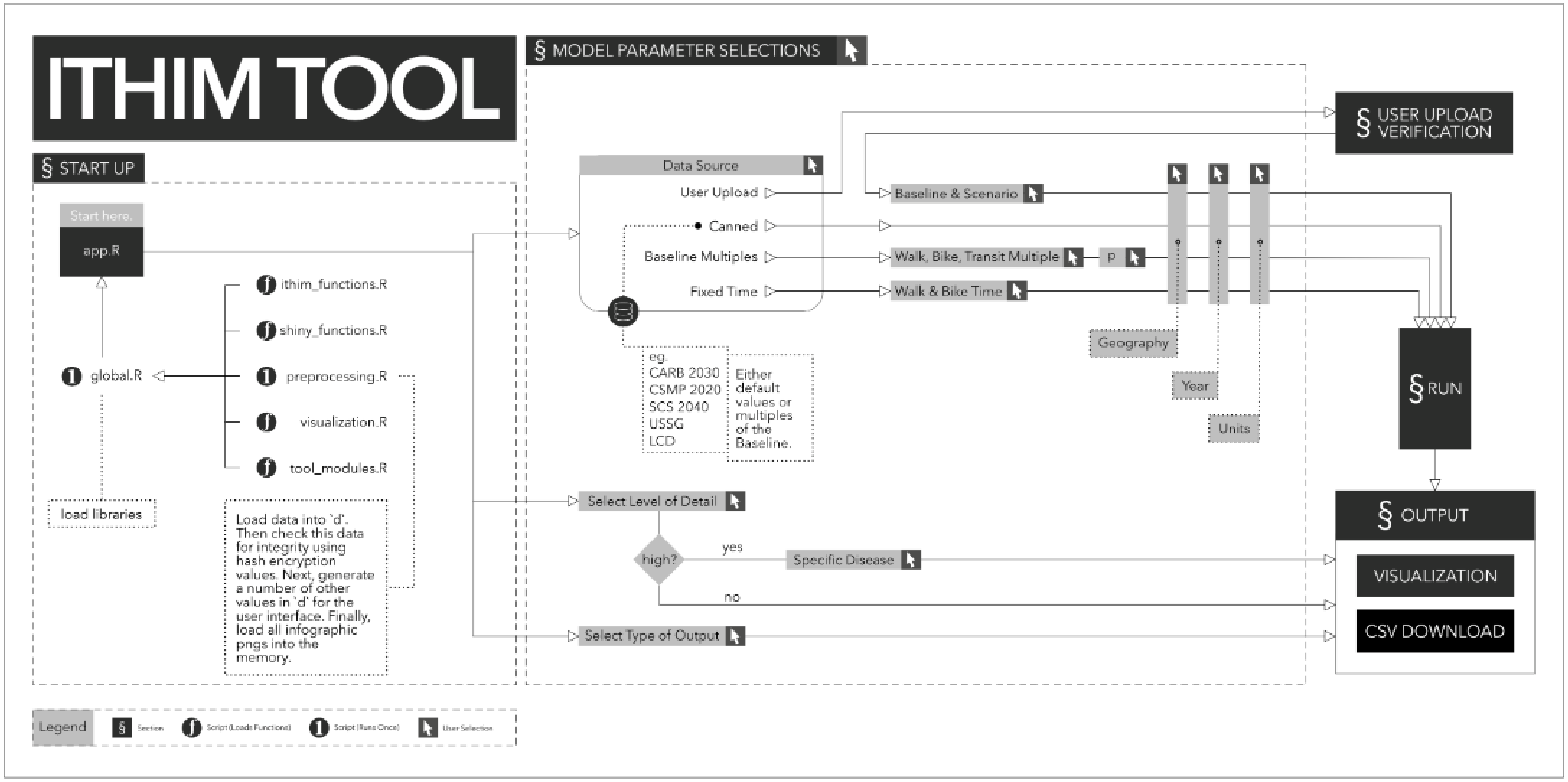
Scenarios: CARB Scoping Plan (2030)
Geographies (State/Regions /Counties): California
Time Periods: 2010

Replacing short car trips with walking and cycling increases physical activity, which reduces the risks of chronic disease. Walking, cycling, and transit also reduce tailpipe pollution from cars. Lower pollution reduces chronic disease and the health impacts of climate-warming gases. Scenarios that emphasize bicycling generate the greatest amount of health benefit from physical activity. Measures to protect pedestrians and cyclists will maximize health benefits.

In the **CARB Scoping Plan (2030)** Scenario, the typical **California** resident replaces short car trips by increasing current levels of weekly walking and

Figure 2.2. "Standalone" ITHIM TOOL

Figure 2.1. Overview of the ITHIM TOOL



Load data into `d`. Then check this data for integrity using hash encryption values. Next, generate a number of other values in `d` for the user interface. Finally, load all infographic pngs into the memory.

Either default values or multiples of the Baseline.

2.1.1. ITHIM TOOL Files

Below are lists of the critical files in the /tool_files directory. The R Scripts contain functions or shiny modules that allow the app.R script to work properly. The comma separated values files are the ITHIM data in tabular form, or specially formatted text that is utilized in the ITHIM TOOL. Certain pieces of text in the {shiny} app are better managed outside of the code, because it makes editing them more convenient. The PNG images are for the infographic. It should be noted that there are another set of images used in the ITHIM TOOL in the directory /www. In the case of the infographic images, the R package {grid} needs the images loaded in binary and to reconfigure them to produce the stitched together infographic; in the case of the Summary Report, the images are simply loaded directly into the page using HTML. In the {shiny} world, HTML images must live in the /www directory. Finally, the CSS, or Cascading Style Sheets, provide some of the important styling to the ITHIM TOOL, and to the ITHIM TOOL alone.

R Scripts

Directory	**Filename**	**Purpose**
tool_files	ithim_functions.R	The Analytic Engine of ITHIM.
tool_files	ithim_metadata_extraction.R	Supports data integrity checks. Run externally from **TOOL**.
tool_files	preprocessing.R	1 of 2 scripts that run just once (the other being *global.R*).
tool_files	shiny_functions.R	Some functions utilized in the operation of the **TOOL**.
tool_files	tool_modules.R	The reactive {shiny} modules.
tool_files	visualization.R	Creates the tables, graphs, report, and infographic.

Comma Separated Values

Directory	**Filename**	**Purpose**
tool_files	WalkBikeTransitRatios.csv	Data for model.
tool_files	age_sex_region_county2010-2050.csv	Data for model.
tool_files	age_sex_region2010.csv	Data for model.
tool_files	APC_Disease_Rates.csv	Data for model.
tool_files	ATmean_min_week_age_sex_baseline.csv	Data for model.
tool_files	Baseline_distance_by_facility_type.csv	Data for model.
tool_files	bike_walk_cv.csv	Data for model.
tool_files	bus_occupancy.csv	Data for model.
tool_files	CalBurdenDisease2010.csv	Data for model.

Comma Separated Values

Directory	Filename	Purpose
tool_files	causelist.csv	Data for model.
tool_files	CO2g_mi.csv	Data for model.
tool_files	COI2010USD.csv	Data for model.
tool_files	data_dictionary.csv	Text for TOOL.
tool_files	default_narratives_2019_07_10.csv	Data for model.
tool_files	DiseaseRiskAdjuster.csv	Data for model.
tool_files	Infographic_ussg_lcd.csv	Data for infographic.
tool_files	ithim_tool_strings.csv	Data for model.
tool_files	METminWalk_Bike.csv	Data for model.
tool_files	nonTravelMETS.csv	Data for model.
tool_files	PA_RR.csv	Data for model.
tool_files	ParameterDefaults.csv	Data for model.
tool_files	PM25_RR.csv	Data for model.
tool_files	PM25CARB2010_2050.csv	Data for model.
tool_files	regions.csv	Data for model.
tool_files	report_template.csv	Text for TOOL.
tool_files	rti_baseline.csv	Data for model.
tool_files	tool_table_strings.csv	Text for TOOL.
tool_files	user_error_codebook.csv	Text for TOOL.

PNGs

Directory	Filename	Purpose
tool_files	01 - Header.png	Infographic
tool_files	02 - Transportation Impact.png	Infographic
tool_files	03 - Direct Impacts.png	Infographic
tool_files	04 - Scenario Summary.png	Infographic
tool_files	05 - GHG Emissions.png	Infographic
tool_files	06 - Years of Life.png	Infographic
tool_files	07 - Health Cost Savings.png	Infographic
tool_files	08 - Air Pollution.png	Infographic

PNGs

Directory	Filename	Purpose
tool_files	09 - Outcomes.png	Infographic
tool_files	10 - Footnotes.png	Infographic

Cascading Style Sheets

Directory	Filename	Purpose
tool_files	ithim_tool.css	Adds style to TOOL.

2.2. Modularity

A {shiny} app is typically launched from a single *R* script with the command:

shinyApp(ui = ui, serve = server).

Above this line of code, the script will typically contain a ui variable and the definition of a server function for the above shinyApp() function to take as arguments. However, unlike a conventional {shiny} application, the ITHIM TOOL utilizes an "abstraction" designed by the creators of {shiny} to facilitate the incorporation of reactive code saved in another *.R* script. This abstraction is called a shiny module. Just as a function can be defined in a different *.R* script file, and then loaded into another *.R* script (much as our *app.R* does with the line source("global.R")), a shiny module may be inserted into our ui and server from a different *.R* file. Therefore, we can have a perfectly functioning ITHIM TOOL with the following code in our *app.R*:

```
# Run *one-time* only code.
source("global.R")

# Define UI for application
ui <- fluidPage(
  theme = "bootstrap2.css",
  ithim_toolUI("TOOL")
)

# Define server logic
server <- function(input, output) {
  callModule(ithim_tool, "TOOL")
  }

# Run the application
shinyApp(ui = ui, server = server)
```

Notice the bit of code in the ui that calls the function ithim_toolUI() and the bit of code in the server function definition that calls the function callModule(). Two important steps are occurring in the above code. First, the line that reads source("global.R") is *reading R code from a file called global.R.*

(A more thorough explanation of the *global.R* file will be described later.) If you take a look at the bottom of the *global.R* file briefly, you should notice a line that reads `source(file.path(pth, "tool_modules.R"))`. This line yet again reads in another file, *tool_modules.R*. Indeed, this command composes the second important step, *the definition of two functions* (in *tool_modules.R*).

Here we depart from the *app.R* file to briefly review *tool_modules.R*. Despite the greater than 1,600 lines of code in this script, only two functions are being defined here. You'll note that these functions are in fact the aforementioned "bits of code" in the beginning of the previous paragraph. One defines the content to be inserted into the `ui` and the other to be called from the `server` function. In fact, for the most part, the form of the code presented here in these two functions appear to be the same form that might conventionally appear in the *app.R*. For instance, much like any `ui` there are a number of calls to the function `fluidRow()` (which creates a container for some content in the `fluidPage()`). However, you will note two differences: first, that the function call `fluidRow()` is occurring in the separate *.R* file; second, that a new function appears, `NS()`.

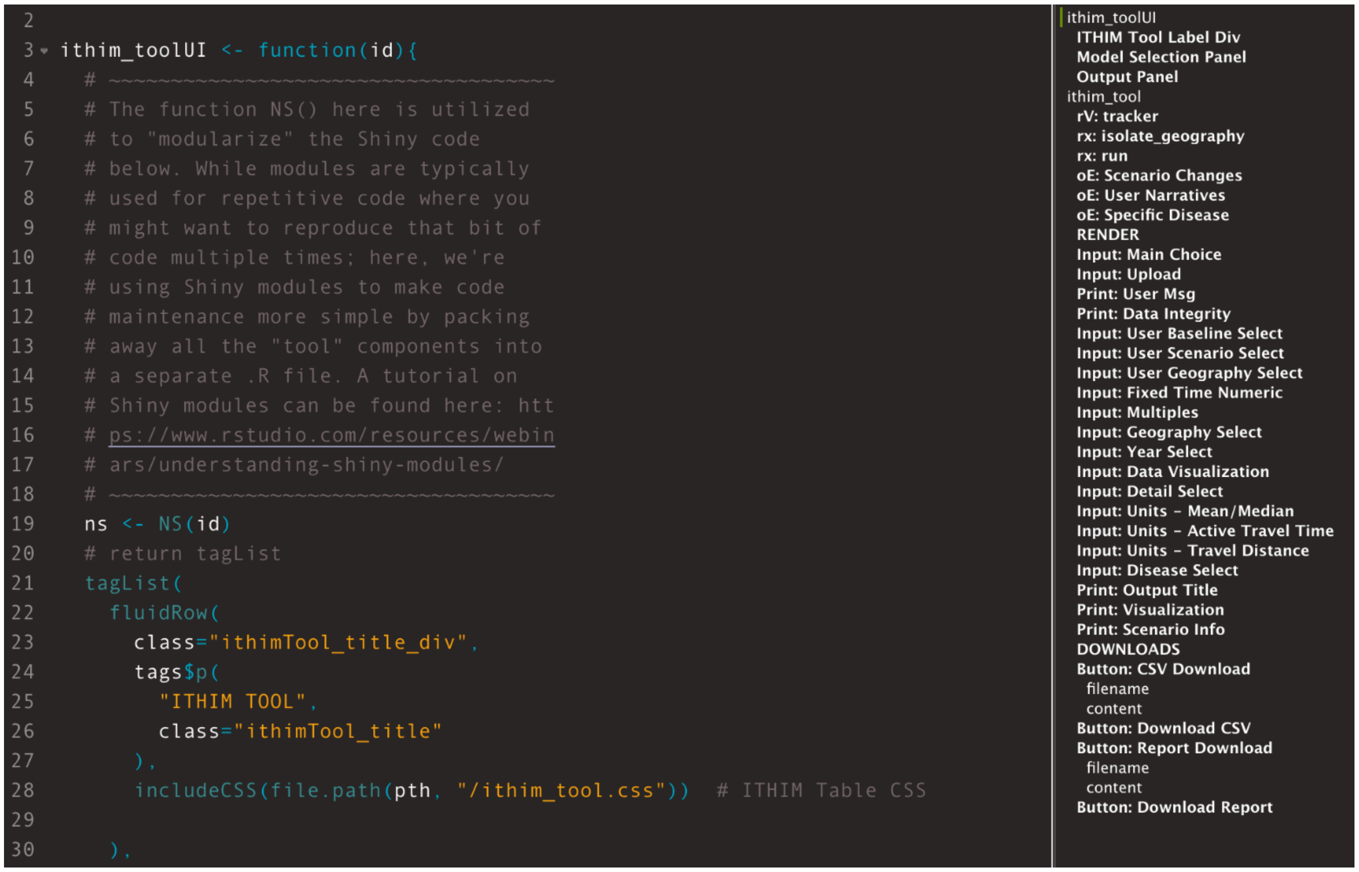
A thorough explanation of what exactly this new function does is beyond the scope of this documentation (again, please review the provided links to the tutorials on {shiny} modules)^3. However, in brief, `NS` is reserving a "namespace" (a method to refer by name to the argument being passed from our `ui` and `server` to the newly introduced functions `ithim_toolUI()` and `ithim_tool()`). The reason why this layer of complexity is required is due to the principle motivation for the developers of {shiny} to create "modules": the module (typically) allows programmers to repeat chunks of reactive code in a single page. However, in our use case, it allows us to separate the ITHIM TOOL from the rest of the ITHIM website, in effect allowing us to:

1. Divide the code in such a way as to facilitate the Cal-ITHIM team to split up our own work in the development process
2. Better organize our files
3. Isolate code to pass along to other coders only interested in the tool that executes the *ITHIM*

So, at the cost of a little complexity, we gained a great deal in terms of cleaning up our code. Moreover, this documentation can now point to specific files where the ITHIM TOOL "happens. One could cut much of the code in *tool_modules.R* and paste it directly into the *app.R* script. The only thing you would need to do is remove all the references to `NS()`. More about modularity is dedicated to the documentation page about the *tool_modules.R* script later.

2.3. The Shiny App

Now that the {shiny} module system has been described, we'll briefly review the organization of the functions defined in the *tool_modules.R*. Please note that many of the *.R scripts* in the ITHIM TOOL utilize "document outlines," which should enable coders to view the organization of code in RStudio. This is certainly the case for the *tool_modules.R*, and readers are advised to enable document outlines when reviewing this script for both organizational and navigational purposes (see Fig. 2.3).



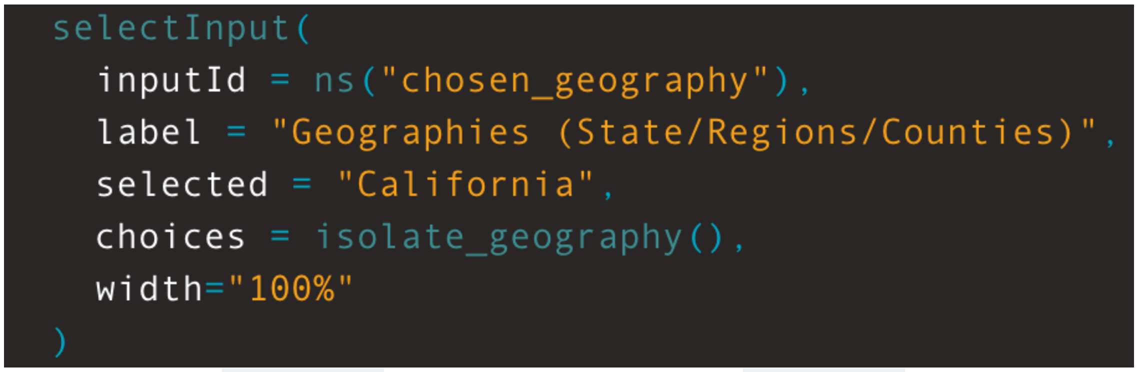
```

ithim_toolUI <- function(id){
  # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  # The function NS() here is utilized
  # to "modularize" the Shiny code
  # below. While modules are typically
  # used for repetitive code where you
  # might want to reproduce that bit of
  # code multiple times; here, we're
  # using Shiny modules to make code
  # maintenance more simple by packing
  # away all the "tool" components into
  # a separate .R file. A tutorial on
  # Shiny modules can be found here: htt
  # ps://www.rstudio.com/resources/webin
  # ars/understanding-shiny-modules/
  # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ns <- NS(id)
  # return tagList
  tagList(
    fluidRow(
      class="ithimTool_title_div",
      tags$p(
        "ITHIM TOOL",
        class="ithimTool_title"
      ),
      includeCSS(file.path(pth, "/ithim_tool.css"))  # ITHIM Table CSS

    ),
```

Figure 2.3. On the left is lines 2-30 of the tool_modules.R, while on the right, the Document Outline is visible. On macOS, type cmd + shift + O to toggle it on and off.

2.3.1. The User Interface (UI)

The ithim_toolUI() function takes some "id" as an argument and outputs an object from shiny::tagList().[4]. This tagList object, is itself composed of multiple fluidRow() objects. Principally, there are two calls to fluidRow() here: the first creates a "banner" (a simple HTML *div*) that just presents the title "ITHIM TOOL"; the second creates the remaining content of the ITHIM TOOL, and is thus quite complex. Indeed, this second fluidRow object is the main interface for ITHIM, with all its options and output. It is composed of two column() objects: the left hand side has "model selection" options for running the model, and the right side has the output (e.g. the infographic). To clarify, this set of "options" and "output" is not technical **R** or {shiny} language. Rather, we use it here in this documentation in the vernacular sense to refer to what the end user will see.

Both columns (i.e. the model selections/options and the output) contain multiple calls to the function uiOutput(). This may be misleading as simple {shiny} applications typically prompt users for information (i.e. inputs) more directly (for instance, by calling the function selectInput(), see Figure 2.4), and thus exclusively reserving uiOutput() to what you might normally consider to be "outputs" like tables and graphs. However, due to our need for "model selections" to react to user choices throughout the end user's process of exploring model options, the UC Davis ITHIM team decided to have selection options appear or disappear *conditionally*. Therefore, the function renderUI() is utilized liberally to support this more dynamic functionality.



```
selectInput(
  inputId = ns("chosen_geography"),
  label = "Geographies (State/Regions/Counties)",
  selected = "California",
  choices = isolate_geography(),
  width="100%"
)
```

Figure 2.4. A simple selectInput() function. In fact, this simple selectInput() call is embedded in a renderUI() function in the actual *tool_modules.R*, so that it may be conditionally invoked.

There is a good deal of *CSS* present in this piece of code. Ideally, this would be transferred to a CSS document to be by itself. Indeed, a greater portion of the CSS used in this tool is found in file *ithim_tool.css*. Thus, the reader will note that at times the style will be defined within the R script, and at other times an HTML class will be applied to various {shiny} HTML tags. See the sample code below.

```
# Style is embedded:
column(
   width = 4,
   offset = 0,
   style = paste(
     "background-color: transparent; ",
     "color:",
     otros_colores[7],
     "; ",
     "margin: 0; ",
     "color: #1F8BBF; "
   ),
# Style is pulled from ithim_tool.css
tags$h3(
     "Make Your Selections",
     class="ithimTool_modelSelection_headers"
   ),
```

2.3.2. The Server Function

This function is more complicated and a more detailed description of the *tool_modules.R* will be provided later in this documentation. However, here we will provide a brief overview of the server function.

The ithim_tool server function can be conceptually divided into three parts:

- The processing functions.
- The user input functions.
- The output for user consumption.

Rather than technical terms specific to R or shiny, the parts terminology provide convenient descriptions to help you navigate the mesh of code.

2.3.2.1. The Server Function: Processing

The processing functions include the "tracker", "isolate_geography", `run`, and three `observeEvent()` functions. As these are all functions with some form of reactivity, they respond to various other components spread throughout the `ithim_tool` server function. The most important processing function, and the best place to begin, is the `run` `reactive()` function.

The `run` `reactive()` function takes all the various inputs to the ITHIM (e.g. which Scenario, what year, and what geography to model), reformats these parameters/inputs, and sends them through a chain of functions that compose the ITHIM (i.e. the functions from *ithim_functions.R*). One of the key tasks that `run` accomplishes is to tell the *ITHIM functions* (i.e. the Analytic Engine) what kind of parameters and data are being passed to it. Are the data from a user? Is the scenario some version of a "baseline multiples" scenario[5]? And so forth. Certainly, other things are going on in "run," but taking inputs and then setting them up to work is the main idea.

By the end of the process, `run` returns a single `list` (see more here). The `list` is filled with numerous bits of data, and contains the essential products of the calculations of the ITHIM. Not only does it have `data.frames` with such things as the Road Traffic Injuries Parts Attributable Fraction table, but also all the stepping stones it took to get there. However, nothing in the output of `run` is readily consumable for users. That task is left to other functions. It should be noted that the "tracker" and "isolate_geography" functions helped implement the `run` `reactive()` function. Whereas, the three `observeEvent()` functions are only invoked if the user wants to upload their own data.

(In the Document Outline panel in RStudio, these functions are prepended with "rV", "rx", and "oE".)

2.3.2.2. The Server Function: Inputs

The inputs/parameters that a user selects occurs within bits of code that begin like this: `output$NAME_OF_PARAMETER <- renderUI({…`, where `NAME_OF_PARAMETER` is some parameter like the *year* that the user wants to run. The bulk of the inputs appear or disappear on a conditional basis, and therefore we utilize the `renderUI()` function. These functions allow us to produce drop down menus, number input boxes, or radio buttons based on certain conditions. Take the *year selection drop down menu* declared in `output$year_select` (see the code chunk below). It requires that a scenario have been selected first, hence the bit of code, `req(input$chosen_scenario)`.

When you run the ITHIM TOOL, the year selection menu appears right away. This is explained by the fact that the ITHIM TOOL selects a scenario by default to run at start up. (Indeed, it does this for the year selection, too.) Even when you change the scenario, the year selection menu never seems to go away. This is may "appear" to be the case, but in fact, this is just an artifact of fast computers; in other words, it happens so quickly that a user would not take notice that the menus appear in succession.

```
output$year_select <- renderUI({
 req(input$chosen_scenario)
 if (input$chosen_scenario != "User Upload" |
   input$chosen_scenario == "User Upload" &
   !is.null(tracker$usr_narratives)) {
  # If user upload.
  selectInput(
   inputId = ns("chosen_year"),
   label = "Time Periods",
   selected = 2010,
   choices = seq(2010, 2050, by = 5),
   width = "100%"
  )
 }
})
```
(In the Document Outline panel in RStudio, these functions are prepended with “Input”.)

2.3.2.3. The Server Function: Outputs

These functions take the result of the `run` `reactive()` function and produce outputs for users to consume. These include the tables, the graphs, the infographic, and the summary report (as well as titles of graphs, error messages when users upload faulty data, and quick facts about the scenario that the user selected). The most complicated of these functions is the visualization function, `output$vis`. It takes the output of `run` and then sends it through various functions from *visualization.R* to produce outputs. In most cases, the output of these functions is some form of `list` or `tagList`, which in turn are composed of some sort of graphical or HTML textual items. Additionally, the outputs also include a download button that allows users to retrieve the tables produced by functions from *visualization.R* as a csv, or comma separated values file that can be opened in spreadsheet software.

(In the Document Outline panel in RStudio, these functions are prepended with “Print” or “Button”.)

2.4. Visualization

The *visualization.R* script is also a complex script and comprises all the functions necessary to produce the attractive graphics in the ITHIM TOOL: the HTML tables, the graphs, the summary report, and the infographic. It contains 18 functions divided into six groups:

- string substitution
- `data.frame` making
- table making
- graph making
- report making

- infographic making

A key function is fn_dataframer. Most other functions depend on the fn_dataframer function either directly or indirectly. It takes the output of the run reactive() function, outlined above, and produces a number of data.frames. It takes the raw output of run and distills key ITHIM results into one of three sets of tables: *Summary*, *Medium*, or *High* detail. The code reveals that each level of detail follows a standard nomenclature, STab, MTab, or HTab, with each appended by a number that uniquely identifies it. For instance, STab4 will always be the "Annual Change in the Burden of Disease by Health Pathway;" or HTab5 will always be the "Rate of Fatal and Serious Injuries." The run function feeds the parameters that instruct fn_dataframer to either produce a set of Summary, Medium, or High tables. In turn, the data.frames that fn_dataframer produces may later be used to manufacture presentable tables, graphs, reports, or infographics. While these data.frames are, in essence, tables, the actual production of HTML tables occurs in fn_tableTagger. In other words, fn_dataframer reorganizes the raw results of the ITHIM, but does not make a visually presentable table for a web page (rather, these tables simply live in the RAM). The process of converting raw ITHIM results into a presentable table was split up into two functions precisely because other functions need those data.frames, but not the HTML tags that go along with it.

The next tier of important *visualization.R* functions include: fn_tableManager (for HTML tables), fn_graphManager (for graphs), fn_reporter (for Summary Reports), and fn_ig_pngMaker (for infographics). After the data.frames have been constructed in fn_dataframer, each one of these four functions represent the beginning and the end for the four types of visualizations. From inside these functions, they call the other functions in *visualization.R* to perform various tasks. Whatever visualization is desired will be produced in the last line of these four functions, and be subsequently passed along to output$vis (in *tool_modules.R*) to be rendered in {shiny} (see above).

Read more about visualizations here.

2.5. The Analytic Engine

Up until this point, we've taken the actual ITHIM for granted. Here we will explore the bit of code necessary to run the model by itself. In a previous section we discussed the run reactive() function, which takes the parameters chosen by the user (e.g. the year, geography, scenario), reformats them, and feeds them into the Analytic Engine. It does this by creating a list object that contains a number of pieces of data and parameters. This list is sent to the first function from the Analytic Engine (i.e. from *ithim_functions.R*); this function called fn_prep, then returns the same list object, but with modifications. In fact, this protocol is followed for the remainder of the run reactive() function, with each subsequent function from *ithim_functions.R*.

The *ithim_functions.R* script is principally divided into two parts. First is a group of functions that either play a role in preparing the data to be processed in ITHIM initially, or functions used sporadically throughout the ITHIM. The second part, beginning with fn_distances, composes the heart of ITHIM. The heart of ITHIM is comprised of 10 functions, each largely organized around a theme (scenario-specific travel distances by mode, active travel times, PAF calculations, change in burden of disease, etc.). This R code is modeled after a simplified R script called the "Long Engine" which does not have reactive inputs for scenario, geography, or time period. It was developed to validate the analytic engine against outputs of spreadsheet ITHIM.

One key difference is that the "Long Engine" includes some bits of code that exist outside of any defined functions, but as the ITHIM TOOL lives in a {shiny} app, it demands that all components of ITHIM exist within some function. Thus some parts of the "Long Engine" have been incorporated into various functions. It should also be noted that much of the conditional statements that help the "Long Engine" decipher scenarios has been reconfigured to utilize attributes. This topic is explored in further detail in later parts of the documentation.

One can run the functions in *ithim_functions.R* in a static environment (i.e. not in a {shiny} app) if one desired to do so. To facilitate this process a script from the UC Davis development team is available to navigate through a static version of *ithim_functions.R*.

Read more about the Analytic Engine here.

2.6. Startup Scripts

The above sections have described bits of code that have been designed to be called multiple times. For instance, you may want to run ITHIM with different scenarios, and the ITHIM TOOL accommodates this. However, parts of the ITHIM TOOL only need to run a single time. These bits of code are found in the *global.R* and *preprocessing.R* scripts. The *global.R* runs first, and thus follows the R {shiny} convention of having one-time-only code reside in a file named "global." Consequently, *global.R* must reside next to the *app.R* script in the file system. Again, this follows {shiny} convention. The *global.R* has a simple task, to load all the relevant packages, and then to load the R scripts with the functions the ITHIM TOOL requires. The order in which R scripts load matters, even though the bulk of R scripts only contain function definitions. This is because the *preprocessing.R* script requires functions from other scripts.

The *preprocessing.R* script importantly sets up all the pre-established data for the ITHIM TOOL. It loads all the csv data files, including values for various ITHIM parameters; it also runs a data integrity check (to detect if any bit rot has occurred); and it loads the png files for the infographic. This draws out an important distinction. The bulk of the ITHIM TOOL has hitherto relied on a `list` object named `l` (created in `run`)[6] , however another important `list` object exists in this {shiny} app, `d`. This latter `list` object stores all the data loaded into the RAM by *preprocessing.R*. This data is not changed once it has been created. In turn, this allows the ITHIM TOOL to run multiple scenarios without the worry of changing the core bits of data that stay consistent from scenario to scenario. For instance, the names of the regions that the ITHIM TOOL uses stays the same irrespective of the scenario.

Chapter References

1. Whenever the term "ITHIM TOOL" appears, it refers to the "standalone" tool without the contextual content (Home Page, About, Decision Support, User Support).

2. Note, the *app.R* file is downloadable from our website.

3. Please refer to this webinar for a basic introduction to {shiny} modules.

4. The `shiny::tagList()` is invoking the function `tagList()` from the {shiny} package. This is both working **R** code and a shorthand used in this documentation to specify the package and function.

5. If what a “baseline multiple” is not clear to you, please see the general documentation about the ITHIM. In short, this term signifies that we want to multiple current levels of walking or cycling. More to the point, this term is not a “coding” term (which is the subject of the ITHIM TOOL documentation), but is a term from the conceptual ITHIM.

6. The `l ("el")` object has nothing to do with *preprocessing.R*. It is covered in Chapter 5.

3. Reactive ITHIM

Much of the programming innovation of the ITHIM TOOL occurs in the *tool_modules.R* script. This document reviews {shiny} Modules, the UI Function (User Interface), and the Server Function.

3.1. Modules

For those unfamiliar with **{shiny} modules**, reference materials produced by R Studio are available:

- Webinar
- Introductory article

We begin to explore {shiny} modules by focusing on the introduction of a new function, NS(), into the code[1] . This function reserves a "namespace" throughout the shiny module, allowing multiple instances of a module to exist simultaneously (presuming that each is given a unique ID). While the use of {shiny} modules in the California ITHIM project differs from this classic use-case of multiple instances, the fact remains that you could run multiple instances of the ITHIM TOOL on the same webpage if you so desired. In which case, you would need to be able to identify which copy of the UI interacts with which copy of the Server function. Therefore, by utilizing NS() and reserving a "namespace," one can effectively label each function so they may communicate and share data with their proper corresponding partner. In the *app.R* script, you'll note that we've used the name, "TOOL" (see the code chunk below).

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Selections from the app.R script
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

# the ui
ithim_toolUI("TOOL")

# the server
callModule(ithim_tool, "TOOL")
```

Let's continue by examining the use of NS() and its constituent bits of code in the UI function. The following code chunk reveals three distinguishing features of {shiny} modules as they regard the UI function.

```
# 1.
ithim_toolUI <- function(id){ ... }

# 2.
ns <- NS(id)

# 3.
uiOutput(ns("user_choice")),
```

In the first sample line of code, the UI function defines its only argument (or "formal") as id. In the second line, the NS() function takes id and returns a new function created around id. This new function ns() is then utilized in the third line to retrieve the "user_choice" from the Server function. In short, this last step facilitates this UI function to share data with a Server function—and not just any Server function, but the one with the same id.

Correspondingly, the Server function also differs from the typical {shiny} app by integrating code needed for modularity.

```
# 1.
ithim_tool <- function(input, output, session){ ... }

# 2.
ns <- session$ns
```

Here, the function definition of the ithim_tool Server requires the argument (or formal) session. The session refers to a semi-permanent bit of information used between the user's browser and the server. In this case, it appears to be holding onto the relevant "namespace" information. The second line retrieves the ns() function we created in the UI. The ns() function needs to be retrieved in the Server function in an effort to mimic the UI as renderUI() functions utilize ns() in much the same fashion. If this part is unclear, please refer to the section below on the renderUI().

In summary, the NS() function and its constituent bits of code are the key mechanisms by which modularity is implemented in {shiny}. Please be conscious that as renderUI() functions are added to the ITHIM TOOL that outputs must necessarily be wrapped in ns() functions (for example, see the code chunk below).

```
output$user_choice <- renderUI(
  selectInput(
   inputId = ns("chosen_scenario"),  # ns() called.
   label = "Scenarios",
   selected = "CARB Scoping Plan (2030)",
   choices = d$main_choice,
   width="100%"
  )
 )
```

TIP: Spend some time looking at the *tool_modules.R* script. Do a search (ctrl+f or cmd+f) through the document, and locate all the instances of NS() and ns().

3.2. UI Function

The UI function returns a shiny::tagList() object, i.e. HTML script stored in an R list object. It has a fairly simple structure: two fluidRows, with the second row being divided into two columns. The first fluidRow merely displays the title, and loads ithim_tool.css stored in the tool_files directory. The second row is admittedly complex. It begins with a lengthy bit of CSS (see the fluidRow argument style).[2] Next it creates the first, left-hand side column with a width of 4. This column is where the user selects their model parameters, hence it includes some HTML tags for labels (e.g. tags$h3("Make Your Selections", ...)) and calls to the uiOutput function. Rather than detailing the way in which the `uiOutput` and `renderUI` functions are utilized in the ITHIM TOOL here, we'll reserve that discussion for the Server function when we'll see where it originates in the code.

```
# Model Selection Panel ----
column(
  width = 4,
  ...)
```

Finally, the second column, at a `width` of 8, produces the output. This "output" actually includes four possible components: (1.) a message about the integrity of the data (`"data_integrity"`), (2.) a message about quality of data uploaded by the user (`"msg"`), (3.) the title of the output (`"title"`), and (4.) the content (`"vis"`). While these four output components are not described here in length, readers should note that the bulk of the data integrity check happens in the *preprocessing.R* file, the code that produces the message exists in the *shiny_functions.R* file, and the last two items are produced in the Server function.

```
# Output Panel ----
column(width=8,
  offset=0, # see docs on column()
  style=
    paste(
      "background-color: white; ",
      "color:",
      colores[5], # this is just a character vector of hex colors.
      "; "
    ),
  uiOutput(ns("data_integrity")), # (1.)
  uiOutput(ns("msg")), # (2.)
  uiOutput(ns("title")), # (3.)
  uiOutput(ns("vis")) # (4.)
)
```

3.3. Server Function

Unlike the UI function, the order of the Server function is arbitrary. Indeed, as {shiny} is reactive, each component of the Server function may be called upon conditionally. As such, this section will describe each object or event handler inside the Server function by its category.

3.3.1. reactiveValues

The `tracker` is the only `reactiveValues` object in the whole of the Server function. We placed the `tracker` at the beginning of the Server function to suggest to future code maintainers the importance of it as backbone to the program. Indeed, as users click on various options in the model selections panel in the UI, it may trigger certain bits of information to be stored in the `tracker`. For instance, if the user would like to see the "High" detail tables, the user implicitly wants to look at one of the many possible "disease" tables (i.e. `HTab3`). In this case, the tracker is notified that the user wants to view a specific disease table by assigning a boolean value to the tracker, i.e. tracker$specific_disease <- T.

3.3.2. reactive

There are two reactive function calls in the Server function, isolate_geography and run. The former behaves conditionally based on the choice of the user (i.e. depending on input$chosen_scenario). It then attempts to pull an array of regions (e.g. the San Francisco Bay Area or California). From here it retrieves the geographies which are sub-units of regions. Depending on the baseline/scenario chosen, these regions and geographies are sourced from different places (e.g. the default narratives csv in tool_files).

The second reactive function creates run, perhaps the most important reactive value. The object run takes all the inputs the user has selected, assembles it in a manner legible to the Analytic Engine, and then runs the Analytic Engine. (See more on the Analytic Engine here.)

The reactive that creates the run object begins by determining what inputs it needs as requisites to operate correctly (see req() calls), which in turn depends on the value of the input$chosen_scenario. The following set of if-statements (see the code chunk below) conditionally produces a baseline object and a scenario object. These objects include the standard ITHIM input data, variably including the “Per Capita Mean Daily Travel Distance” and possibly the “Proportion of Vehicle Miles by Mode and Facility Type.” (The latter occurs when the User Upload scenario has been selected.)

```
# User Upload Scenario
if(input$chosen_scenario == "User Upload") { ... }

# Baseline Multiples Scenario
else if(input$chosen_scenario == "Baseline Multiples") { ... }

# Low Carbon Driving Scenario
else if(input$chosen_scenario == "Low Carbon Driving") { ... }

# Either CARB Scoping Plan or CSMP Scenario
else if (input$chosen_scenario %in%
        c("CARB Scoping Plan (2030)",
          "Caltrans Strategic Management Plan (2020)")) { ... }

# Fixed Time Scenario
else if(input$chosen_scenario == "Fixed Time") { ... }

# One of the remaining Default Scenarios
else { ... }
```

Importantly, the baseline and scenario objects are imbued with “attributes,” namely the "scenario_type". This attribute instructs the Analytic Engine (i.e. the functions in *ithim_functions.R*) on how to handle these inputs. There are 4 possibilities for "scenario_type":

- "user_defined"
- "multiples"
- "default"
- "absolute_time"

The manner in which these scenario types are processed is left to the documentation on *ithim_functions.R*. Based on these four scenario types, the Analytic Engine will utilize a different algorithm (i.e. different bits of code). An example of a baseline object being imbued with the "default" value in the attribute "scenario_type" is presented below.

```
attr(baseline, "scenario_type") <- "default"
```

Once the baseline and scenario objects have been constructed, run then constructs a geography and region object. Again, both of these are selected by the user from a list. These selections are derived from a list of possible geographies created earlier by the isolate_geography reactive value. Immediately after, a p object is created, which merely stores a value for the car miles being substituted in this model run.

At this penultimate stage, we compile all the objects we've made inside run and place them in a list object named l (see more here).

```
l <- list(
   baseline = baseline,  # the baseline created above.
   scenario = scenario,  # the scenario created above.
   fRegion = region,
   fGeography = geography,
   fYear = input$chosen_year,
   table_units = "default",  # regards footnotes for Tables.
   p = p,
   is_formatted = F
 )
```

Left un-presented in the code chunk above, we also append() some parameters regarding the "Units Selection." If the reader inspects the model selection column in the UI, they should note that users are given choices as to how the products of ITHIM are relayed in terms of units: Is the mean or median calculated? Are the distances in kilometers or miles? What are the time units? In days, weeks, or by year? The reader will note that the "Units Selection" only matters if the output will be a table or a graph.

Finally, this list is submitted to the Analytic Engine by invoking a series of function calls originally defined in *ithim_functions.R*. (Note, the order *does* matter.)

```
l <- fn_prep(l)

l <- fn_distances(l)

l <- fn_scenario_time(l)
```

```
l <- fn_pa_quantile(l)

l <- fn_pa_paf(l)

l <- fn_pm_paf(l)

l <- fn_rti_paf(l)

l <- fn_burden_causes(l)

l <- fn_monetization(l)

l <- fn_carbon_emissions(l)

l <- fn_units_conversion(l)
```

The `run` reactive value thus concludes by returning a `list` object with all the parameters created in `run` as well as the products of the calculations in the Analytic Engine.

3.3.3. observeEvent

The ITHIM TOOL has three event handlers, all of which are `observeEvent` type handlers.
The first as written in *tool_modules.R* reacts to changes to `input$chosen_scenario`. Essentially, it aids in the feature of the ITHIM TOOL that allows users to upload their own data. In order to accommodate this feature, the `tracker` keeps track of whether the TOOL should render the user upload widget by storing a boolean value in `tracker$ready_for_user`. If it's true, the user upload widget will appear, otherwise it will not be produced.

Relatedly, the next `observeEvent` handler reacts to changes to `input$user_upload` (i.e. if something has been uploaded to the ITHIM TOOL). If it does, this handler first splits up the csv by scenario/baseline, and then goes through a series of checks on the quality of the data uploaded. The checks inquire of the upload: Is it only one file? Is it blank? Does the csv follow the format the Cal-ITHIM team established for data inputs? Part of this work is completed by functions in *shiny_functions.R*. If the handler verifies all of this, it stores the user's data in the `tracker` and sets `tracker$ready_for_user` back to `FALSE`.

The last event handler reacts to changes in `input$chosen_detail`, informing the `tracker` whether or not we need a "specific disease" (any specific disease!) to be calculated by the Analytic Engine. This only occurs if the `chosen_detail` is high. To see this in action, run the ITHIM TOOL and observe what happens when you choose "tables" as your output at a "high detail." From there, a new selection widget pops up, allowing you to specify what disease you're interested in. Now, this event handler doesn't directly create this new selection widget, but it does trigger `output$disease_select` which does indeed render it.

3.3.4. renderUI

This {shiny} function occurs repeatedly in the ITHIM TOOL. The renderUI function is utilized whenever the ITHIM TOOL conditionally produce a "widget" or prints out an "output". If you inspect the document outline of *tool_modules.R*, you should recognize a number of subheadings titled "Input" and "Print". These subheadings indicate the presence of a call to renderUI with the express purpose of either producing a "widget" or some "output" (respectively).

Regarding "widgets[3]," the simplest case is the very first one written in the script which produces output$user_choice. Indeed, while the entire purpose of utilizing renderUI is to produce something (widgets or outputs) *conditionally*, here we just produce a widget conditionally or not. However, in the next implementation of renderUI (see the code chunk below), which creates output$user_input, you'll first notice a few calls to req() which demand some prerequisite values, and the presence of an if-statement.

```
output$user_input <- renderUI({
 # require these two things...
 req(input$chosen_scenario)
 req(tracker$ready_for_user)

 # Look! An if-statement!
 if(input$chosen_scenario=="User Upload" &
   tracker$ready_for_user==T){
  tagList(
   fileInput(
     inputId = ns("user_upload"),
     label = "Select a file to upload.",
     multiple = F,
     accept = "text/csv"
   ),
   tags$br()
  )
 }
}
)
```

In essence, this code states, if the chosen_scenario is a user upload, and the tracker says it's okay to create a widget, then produce a fileInput() widget[3] .

The renderUI function also serves another purpose in the ITHIM TOOL, it "prints" things (i.e. outputs) out to the screen. For instance, a renderUI is used to warn the user that their uploaded data doesn't fit the format as outlined by the UCD development team. (You'll note that in the code chunk below, the actual creation of the content is not included–the content being the warning message. Rather, the content is created elsewhere and then this renderUI prints that message to the screen.)

```
output$msg <- renderUI({
 req(tracker$msg)
 if(!(is.na(tracker$msg)))
  tagList(
    tags$div(
     tags$p(HTML(tracker$msg)),  # msg is created elsewhere.
     class = "error_msg"
    )
   )
})
```

The above code chunk requires tracker$msg. It then inspects tracker$msg for some text. If text is present, it prints that text out to the screen with tags$p(HTML(tracker$msg)).

renderUI Visualizations

Most of the "Print" oriented renderUI() functions are fairly straight forward. However, the output$vis object requires additional explanation. The output$vis object is designed to take the output of run (i.e. the Analytic Engine), take the user's parameters for the type output desired (e.g. high detail graphs), and then generate some sort of visualization using functions from *visualizations.R*.

It begins by obtaining the output of run, and storing it into the list object l (see more here). It then appends a default value for specific_disease and the chosen type of visualization. It then proceeds by appending the level of detail based on the user's specifications. In the next series of if-else statements, the function creates the visualization using functions from *visualizations.R* based on the type of output selected by the user (i.e. tables, graphs, infographic, or summary report). The functrion generates a tagList object with bits of HTML. It may be instructive to place a browser() call before output$vis is returned, that way you can View() the variable o to get a sense of what is included.

In the case of the summary report or the tables, the process of making the visualization is starightforward. The function that creates output$vis relies on the functions from *visualizations.R*. However, both the graphs and the infographic require a bit more work. Both of these outputs rely on functions from *visualizations.R*, but importantly need to **render** the visualizations within the function local(). The important distinction between these two sets of visualizations can be illustrated by the composition of the actual visualization. The end products of both the summary report and the graphs are simply composed of HTML and CSS, whereas both the infographic and the graphs are {grid} objects **rendered** to produce an image (e.g. a PNG). The need to **render** a {grid} object requires the creation of an "environment" separate from the rest of the ITHIM TOOL. As such, they are rendered in a local() function call.

3.3.5. downloadHandler

The final component of the Server function is the `downloadHandler`. which focuses on downloading the contents of Tables outputs as a csv file.

The "Downloads" are actually split into two parts, the creation of a simple button (printed conditionally based on the type of output), and the literal `downloadHandler`. The former is a simple implementation of `downloadButton`. Whereas, the literal `downloadHandler` is considerably more complex.

In short, the csv `downloadHandler` takes three arguments, a `filename`, some `content`, and a description of what the `contentType` is.

```
output$CSV_prep <- downloadHandler(
 filename = function() { ... },
 content = function(fname) { ... },
 contentType = "application/csv"
 )
```

The creation of the `content` is the most complex part. It takes the output of `run` and then generates a series of tables fit to be bound together into a single csv document, alongside all the "breadcrumbs" or model parameters. In effect this means there are a number of consistent columns: the scenario name, the name of the table, the geography, etc. However, this also means there are some inconsistent columns. Rather, each table has its own list of headers, as such there is no set of consistent headers that are shared for this part of the tables. Therefore, each column is merely specified by a letter in the alphabet. The headers for each table are thus embedded in the table. The structure ends up looking something like this:

CSV Download Sample

Scenario	Geography	etc.	Table	a	b	c	etc.
CARB Scoping Plan	California	…	Per Capita Mean Weekly Active Travel Time (minutes) by Age and Gender	Sex	Age.Group	Population	…
CARB Scoping Plan	California	…	Per Capita Mean Weekly Active Travel Time (minutes) by Age and Gender	Male	00-04	1211552	…

3.3.6. Table of Reactive Components

Server Function

Type	Key Variable	Purpose
reactiveValues	tracker	Store values for other reactive functions to check.
reactive	isolate_geography	Generate a list of possible geographies from which users may choose.
reactive	run	Utilizing user parameters, run the Analytic Engine.
observeEvent	input$chosen_scenario	Prepares tracker for user upload feature based on key variable.
observeEvent	input$user_upload	Processes csv that a user uploads with narratives.
observeEvent	input$chosen_detail	Prepares tracker for cases where a specific disease is desired.
renderUI	output$user_choice	Creates primary Scenario selection drop down widget.
renderUI	output$user_input	Creates User Upload widget for csv upload.
renderUI	output$msg	Generates an error notice depending on if tracker$msg is present.
renderUI	output$data_integrity	Creates HTML message if data integrity check fails.
renderUI	output$user_base_select	Creates a widget for baseline selection for User Uploads.
renderUI	output$user_scenario_select	Creates a widget for scenario selection for User Uploads.
renderUI	output$user_geography_select	Creates a widget for geography selection for User Uploads.
renderUI	output$absolute_time_walk	Creates a widget for entering Fixed Time walking.
renderUI	output$absolute_time_bike	Creates a widget for entering Fixed Time cycling.
renderUI	output$multiples	Creates a widget for entering multiples of baseline for walking, cycling, and transit.
renderUI	output$geography_select	Creates a geography selection widget for non-User Upload scenarios.

Server Function

Type	Key Variable	Purpose
renderUI	output$year_select	Creates a 5-year increment selection widget between the years 2010-2050.
renderUI	output$data_vis_select	Creates a selection widget for data visualization.
renderUI	output$detail_select	Creates a level of detail selection widget.
renderUI	output$units_centrality	Creates a units of centrality radio button.
renderUI	output$units_active_travel	Creates a unit of active travel radio button.
renderUI	output$units_travel_dist	Creates a unit of travel distance radio button.
renderUI	output$disease_select	Creates a specific disease selection widget for high-detail tables and graphs.
renderUI	output$title	Creates an HTML title for the output panel, describing the page as tables, graphs, summary report, or infographic.
renderUI	output$vis	Creates the content for the output visualization (see documentation).
renderUI	output$info_box	Creates the informational box in the model selection panel that describes the scenario.
downloadHandler	output$CSV_prep	Creates the csv for the download button to serve.
renderUI	output$downloadData	Creates the download button.

Chapter References

1. Strictly speaking, it appears that NS() is a "function factory".
2. CSS styling is divided between the *ithim_tool.CSS* and the *.R* script.
3. A number of "input" functions (which we've been largely describing as "widgets") are utilized in *tool_modules.R*, however their uses are not described in this documentation. Indeed, there exists ample documentation on {shiny} inputs online.

4. User Upload

In the previous section, we explored the *tool_modules.R* script, examining the modularity of the ITHIM TOOL, the UI function, and the Server function. However, we now discuss how the user of the TOOL submits their own data for analysis.

In general, users may upload their own data following the (csv) template provided under the “User Support” page on the (full) California ITHIM website. That data is then directly received by the {shiny} server and undergoes a series of checks.[1] The code for handling User Uploads lies in two places, in the *tool_modules.R*, but also in *shiny_functions.R*. In fact, the latter script is largely geared towards the User Upload (although, its general purpose is to provide a place to define functions that thematically sit outside of *ithim_functions* and *visualization.R*.)

The User Upload begins in the *tool_modules.R* with a `fileInput` call (in `output$user_upload`). Upon a successful upload, an `observeEvent` is triggered (see the comment `# oE: User Narratives ----` in *tool_modules.R*). After an error check, it reads the csv into a `data.frame`. This new `data.frame` is then passed to a few functions in *shiny_functions.R*.

4.1. shiny_functions

- fn_narrativeSplitter
- fn_user_data_verify
- fn_pullRegions
- fn_user_data_subs
-

The function `fn_narrativeSplitter` is utilized both in *preprocessing.R* and in *shiny_functions.R*. It provides a means of doing a basic error check for the correct column headers, splits them by the `Scenario_ID`, and then returns these `data.frames` as a list to the `observeEvent` in *tool_modules.R*. However, the next line simply calls the next function in *shiny_functions.R*, `fn_user_data_verify`. This last function has the role of going through the User Upload and checks for a number of common errors. If there are errors, upon encountering the first error, `fn_user_data_verify` immediately exits and returns an error message (in `out`) that is passed back to the aforementioned `observeEvent`, triggering a message that appears in the TOOL (see `output$msg`) and disabling the TOOL until the issue is remedied.

There are 12 possible errors, examples of which are provided in the User's Guide & Technical Manual Table 2.5). More details of errors are listed in `user_error_codebook.csv` . The in-code comments also provide a fair amount of information. Note that in `fn_user_data_verify` each possible type of error contains two parts: a test to see if there is an error, and then the returning of an error message (or not). The error message is often customized to reflect the variables where the error is occurring (using `fn_keyworder` to substitute variable names into the messages stored in `user_error_codebook.csv`). Generally speaking, this structure is followed for most of the error checks.

If the User Upload passes the verification step, then the user’s data is passed to the `fn_user_data_subs` function to fill in any missing data. Indeed, the ITHIM TOOL accepts a limited amount of missing data, and then backfills it with the baseline data that we have for that geography. Also in this function, the units for the data are corrected to match the rest of the default used by the ITHIM TOOL. This data is then returned and passed to the reactiveValues object named tracker.[2]

Chapter References

1. Checking for malicious code is outside the scope of our R programs and threats are minimized by not storing user uploaded data on the {shiny} server (rather, everything is in the web session).

2. Note, the function fn_pullRegions has not been called yet. This function is utilized (much like fn_narrativeSplitter) on canned data as well as User Uploaded data. To see it in action, check out isolate_geography in *tool_modules.R*.

5. ITHIM Functions

The functions defined in *ithim_functions.R* represent the heart of the ITHIM. The functions in this R script are divided into two groups, setup and core functions. The setup functions prepare the input data or create functions necessary for the proper functioning of the core functions, whereas the latter conduct the primary calculations of the ITHIM.

The conceptual detail of how the ITHIM operates is described in *the User's Guide and Technical Manual*. This documentation simply describes notable and higher-level coding approaches and the organization of the code.

The functions in *ithim_functions.R* are defined in the order in which they need to be executed—this is particularly the case for the core functions.[1] For those readers familiar with the "Long Engine," the core functions should be quite familiar—indeed, these scripts are nearly identical. With the exception of the first two functions (`is.nan.data.frame` and `fn_age_groups`), all the functions in this R script require the input `l`, which is a `list` object that gradually gains elements as it is passed from one ITHIM function[2] to another. Initially, the `l` object contains only the parameters it is assigned in the `run` reactive function in *tool_modules.R*. This includes such things as the baseline data, the scenario data, the region, the geography, the year, the car miles substitution (`p`), and the measure of centrality (i.e. mean or median).

A summary description of each function is provided in the table below. While a basic understanding of the ITHIM and R should be a sufficient foundation for deciphering the *ithim_functions.R* script, a few points will be addressed here for clarity.

5.1. Attributes

Attributes in R serve as a means to attach metadata to some "object," with an object being a variable like a `data.frame`, `vector`, or `list`. Essentially, these two bits of data (baseline and scenario) are treated as being instilled with a particular attribute. This particular attribute for the baseline and scenario data answer the questions: Is the data user defined? Is the data meant to be augmented by some multiplier? Should the data be processed normally?

In the ITHIM TOOL, by applying metadata in the form of an attribute (in `run`), we help functions (in *ithim_functions.R*) determine how to handle the data. With that said, future iterations of ITHIM functions could reconsider the approach taken in the ITHIM TOOL by either stripping it of its pseudo-object–orientation or fully implementing object-orientation. (The advantages of object-orientation lead to simpler code to read and a simpler organization scheme, but this occurs at the cost of comprehension by novice programmers.) The following code chunk demonstrates a conditional statement that does *something* if the data was uploaded by the user:

```
if(attr(l$scenario, "scenario_type") == "user_defined") { ... }
```

In the above code chunk `attr()` is called to retrieve the attribute for the "key" `"scenario_type"` of the piece of data stored in `l$scenario`. If `attr()` returns the value `"user_defined"`, the program does *something*. The *something* could be some bit of code that only runs when the data is user defined. (Note, the use of ... is not recognized R code, but rather is a shorthand used here to indicate some code belongs here.)

5.2. The l ("el") Object

It should be noted that aside from "attributes," the ITHIM TOOL[3] also mimics object-orientation by passing the list object l through most of the TOOL, from run all the way through its endpoints, such as output$vis or output$CSV_prep (including the functions in *ithim_functions.R*). Rather than passing specific pieces of data from one function to another as declared arguments (or formals), it bundles everything up into a single list object whose breadcrumbs persist through the entire lifespan of the TOOL. (For instance, the data.frame stored in l$rti_RR_mode is first created in fn_rti_paf, but continues to exist long after it is finally used in fn_data_framer.) This approach greatly simplifies calling upon the breadcrumbs later in the TOOL.

5.3. Table of ITHIM Functions

Function Name	Arguments	Purpose
is.nan.data.frame	x - data.frame	Adds functionality to is.nan to serve a data.frame.
fn_age_groups	x - integer orcharacter vector	Converts standard age integers to age ranges.
fn_prep	l - list	Prepares the l list object by subsetting data from d and inputs from run.
fn_distances	l - list	Inputs baseline per capita mean annual distances (miles) by mode (walk, bike, car-driver, car-passenger, bus, rail, motorcycle, and truck) and calculates distances for the scenarios: multiples of the baseline, California Strategic Management Plan, and 2017 California Air Resources Board Scoping Plan Update. Increases in walking, cycling and transit replace miles traveled by cars.
fn_scenario_time	l - list	Calculates per capita mean weekly minutes of active travel physical activity based on walking distances divided by average walking speed (3 mph) and cycling distances divided by average cycling speed (12 mph)
fn_pa_quantile	l - list	Calculates the age and sex specific per capita mean physical activity times for walking and cycling at the 10th, 30th, 50th, 70th, and 90th percentile of the physical activity distribution based on the overall mean, standard deviation of walking and cycling, and the inverse transform of their lognormal values. The

Function Name	Arguments	Purpose
		physical activity times are then weighed by age-sex specific MET values based on an average of 3 METS for walking and 6 METS for cycling.
fn_pa_paf	l - list	Calculates the age-sex-and disease specific population attributable fraction based on disease-specific dose response coefficients and the change in physical activity METS from baseline to scenario travel.
fn_pm_paf	l - list	Calculates the change in ambient mean PM2.5 concentration (air basin) based on changes in car emissions from baseline to scenario and the concentration response coefficient for cardiovascular disease, cardio-pulmonary disease, and lung cancer.
fn_rti_paf	l - list	Inputs baseline fatal and serious injuries stratified by facility type (local, arterial, and highway) and ravel distances by mode for each pairwise combination of striking vehicle and victim vehicle, and outputs the number of scenario injuries, population attributable fraction, and the relative risk by mode of victim due to the change in baseline and scenario travel distances.
fn_burden_causes	l - list	Inputs the age-sex-disease specific burden of disease and corresponding population attributable fraction to calculate the change in deaths, years of life lost, years living with disability, and disability adjusted life years.
fn_monetization	l - list	Calculates willingness-to pay estimate based on inputs of the change disease-specific numbers of death (scenario vs. baseline), which is multiplied by the value of a statistical life (default value $7.4 million). Calculates the cost of illness based on the disease-specific population attributable fraction, which is multiplied by share of national costs in constant 2010 dollars, prorated to the population of the geographic area of analysis.
fn_carbon_emissions	l - list	Calculates the aggregate change in CO2-equivalent car emissions based on the CO2 emissions factor (CO2/mi), the change in per capita mean car travel distances (baseline vs. scenario) multiplied by the population at the scenario time horizon.
fn_units_conversion	l - list	Converts per capita mean distances by mode from miles to kilometers (1.62 km/mi); Expresses per capita mean (or median) distances on a daily, weekly, or annual basis (365 days per year, 52 weeks per year);

Function Name	Arguments	Purpose
		Converts per capita mean active travel time between minutes per day to minutes per week. Expresses central tendency of active travel time as a per capita mean or median, using the transverse normal function of the log of the mean and standard deviation).

Chapter References

1 . Examine the run reactive (see here). In the run reactive, the functions from *ithim_functions.R* are called upon sequentially.

2. "ITHIM function" largely refers to any of the "core" functions in *ithim_functions.R*, although there is an exception in this specific case for fn_prep.

3. This section about the I object refers to the ITHIM TOOL as a whole, and not simply about *ithim_functions.R*.

6. Visualizations

In this chapter we'll examine the functions in *visualization.R*. While these functions are not necessary for the proper working of the ITHIM, they are responsible for the attractive graphics and tables. As described in the "Overview", there are five groups of functions present in this script. This chapter will explore each group, beginning with the root function group in this script that generates a list of data.frames.

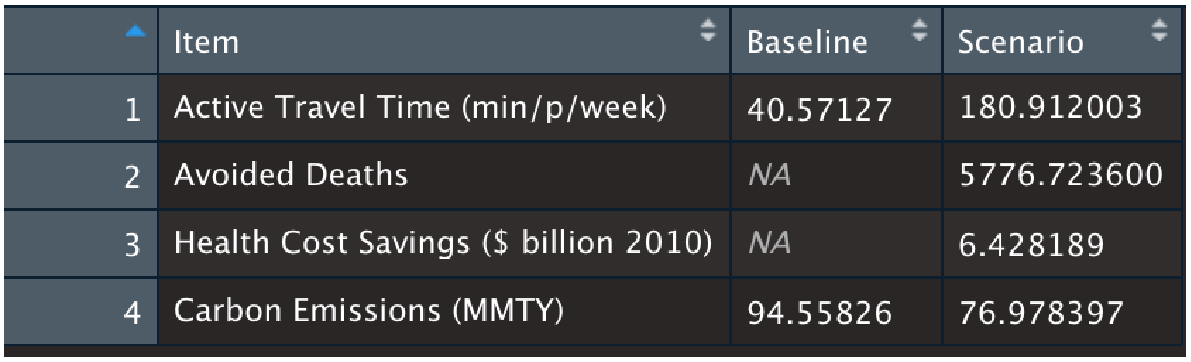
6.1. Group 1: The Root

- fn_specific_disease
- fn_dataframer

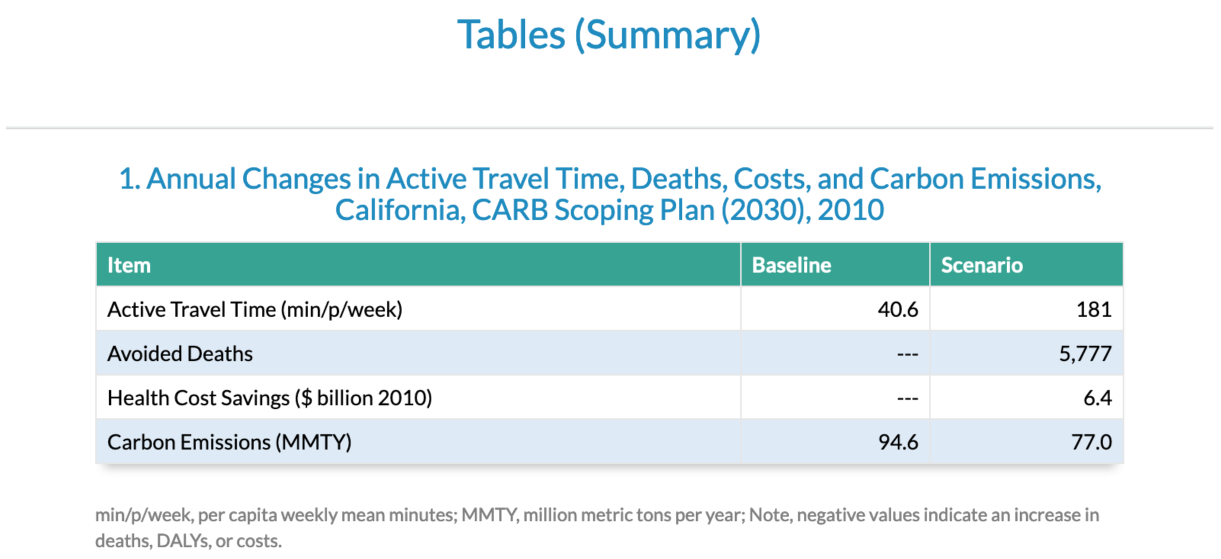
These 2 functions generate the data.frames necessary to draw the tables and graphics in the rest of the *visualization.R* script. The most important function here is fn_dataframer as it returns a list of formatted data.frames. This step is an intermediary between the generation of raw output from the Analytic Engine (Chapter 2) and the HTML tables or graphics users experience in the ITHIM TOOL. As described previously, fn_dataframer will produce a list of either Summary, Medium, or High detail tables, each with a unique name, like STab4 (Summary Table 4). In the final step, each data.frame in the list object is tagged with an attribute indicating this unique identifier (e.g. STab4), as well as the printable "title" of the data.frame (as described later).

The functions in *visualization.R* are called upon in the creation of the output$vis {shiny} object. However, this is not always the case, as users may be interested in downloading the tables (i.e. the data.frames) in the form of a CSV file. In which case, the functions in this group are also utilized, as the process of generating a CSV file re-creates the data.frames from scratch each time it is called. Indeed, the {shiny} object output$CSV_prep (of *tool_modules.R*) is created in order to facilitate the download of a CSV file. This is notable as it operates slightly differently than in the creation of output$vis.

These two {shiny} objects differ, fundamentally, when calling upon the functions in *visualization.R* to generate High detail tables. In the ITHIM TOOL readers will note that when both "tables" and "high" detail are selected, that a new drop-down menu widget is generated which allows users to select a specific disease. Whereas, when users then hit the "download CSV" button, the generated CSV file contains every possible specific disease. This is possible courtesy of the fn_specific_disease function. The {shiny} output$CSV_prep object (1.) obtains the results of run, (2.) obtains the results from fn_dataframer, (3.) runs a loop to get the data.frames for each disease by utilizing fn_specific_disease (see the code chunk next page).



	Item	Baseline	Scenario
1	Active Travel Time (min/p/week)	40.57127	180.912003
2	Avoided Deaths	NA	5776.723600
3	Health Cost Savings ($ billion 2010)	NA	6.428189
4	Carbon Emissions (MMTY)	94.55826	76.978397



Tables (Summary)

1. Annual Changes in Active Travel Time, Deaths, Costs, and Carbon Emissions, California, CARB Scoping Plan (2030), 2010

Item	Baseline	Scenario
Active Travel Time (min/p/week)	40.6	181
Avoided Deaths	---	5,777
Health Cost Savings ($ billion 2010)	---	6.4
Carbon Emissions (MMTY)	94.6	77.0

min/p/week, per capita weekly mean minutes; MMTY, million metric tons per year; Note, negative values indicate an increase in deaths, DALYs, or costs.

Figure 6.1. The image on the top demonstrates a data.frame of STab1 , whereas the image on the bottom demonstrates STab1 as an HTML table. The image on the left is an example of the object returned by this function group (as presented in RStudio using View()). The latter is an example of an HTML table discussed in the subsequent function group.

```
# retrieve data.frames sans specific disease
df_list <- fn_dataframer(I)$df_list

I$specific_disease_title <- T

# retrieve ALL specific disease tables
df_list2 <- lapply(seq_len(nrow(d$causes)), function(i){
 I$specific_disease <- d$causes[i, ]
 fn_specific_disease(I)$specific_disease_df
})
```

Now that the purpose of both fn_dataframer and fn_specific_disease has been reviewed, we'll now explore how each of these functions operate. The fn_dataframer begins by taking the results of run (i.e. the results of the Analytic Engine) as a list object (l) as its only argument. Its first task is to figure out the title for each table that will be created. It uses a template from "tool_table_strings.CSV" in the directory /tool_files. Specifically, it draws from the column labeled "title." Readers should note that in this template, that there are certain words wrapped by curly braces, {}. The use of curly braces in the ITHIM TOOL indicates that some sort of string substitution will occur. For instance, Summary Table 2 (STab2) has the template title "Per Capita {centrality} {at_time_unit_ly} Active Travel Times (minutes)," where the word "{centrality}" will be replaced with either the word "Mean" or "Median", and similarly "{at_time_unit_ly}" will be replaced with "Daily" or "Weekly." In order to do this we create a "lookup table" (a data.frame) and then pass this along with the title template to the function fn_keyworder. In turn, fn_keyworder utilizes regex to replace the matching terms in the template title with the lookup table values (see the code chunk below).

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Create lookup table.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
lookup <- data.frame(
 key = c(
  "{centrality}",
  "{centrality2}",
  "{at_time_unit_ly}",
  "{td_unit}",
  "{td_time}",
  "{td_time_ly}",
  "{at_time_unit1}",
  "{at_time_unit2}"
 ),
 value = c(
  # For Table/Graph Titles
  centrality,  # These are variables
  centrality2,
  at_time_unit_ly,
  td_unit,
  td_time,
  td_time_ly,
  at_time_unit1,
  at_time_unit2
 ),
 stringsAsFactors = F
)

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Run table_str titles through
# fn_keyworder to replace {keywords}.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
table_str <- fn_keyworder(table_str, lookup)
```

In the above code chunk, table_str is character vector containing the template titles of every table/data.frame we're about to construct. In the declaration of the lookup data.frame, a number of variables are passed to column value that happen to share the same name as the key. Rest assured these variables equate to strings with values such as "Median" or "Kilometers".

Now that we've generated a vector of tailored table titles, fn_dataframer continues by entering a set of conditional statements (i.e. if-else statements) based on the level of detail. In fact, fn_dataframer only creates a set of tables/data.frames for one level of a detail at a time (e.g. the Summary level). In each expression (of the conditional statement), fn_dataframer generates between 4-6 data.frames (e.g. STab1, STab2, and so forth). It then places them in a list object named core. At the end of fn_dataframer, it adds two attributes to each data.frame in core, df_id and df_name, or the ID (e.g. STab2) and the name of the table (e.g. "Per Capita Mean Weekly Active Travel Times (minutes)"). It closes by adding this newly created list of data.frames with two attributes each into the argument l as df_list and returns l (see the code chunk below).

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Add the attributes of df_id and
# df_name to each table in core.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
core <- lapply(seq_along(core), function(i){
 structure(core[[i]],
        df_id = table_str$id[i],
        df_name = table_str$title[i]
 )
})

l$df_list <- core
l # return
```

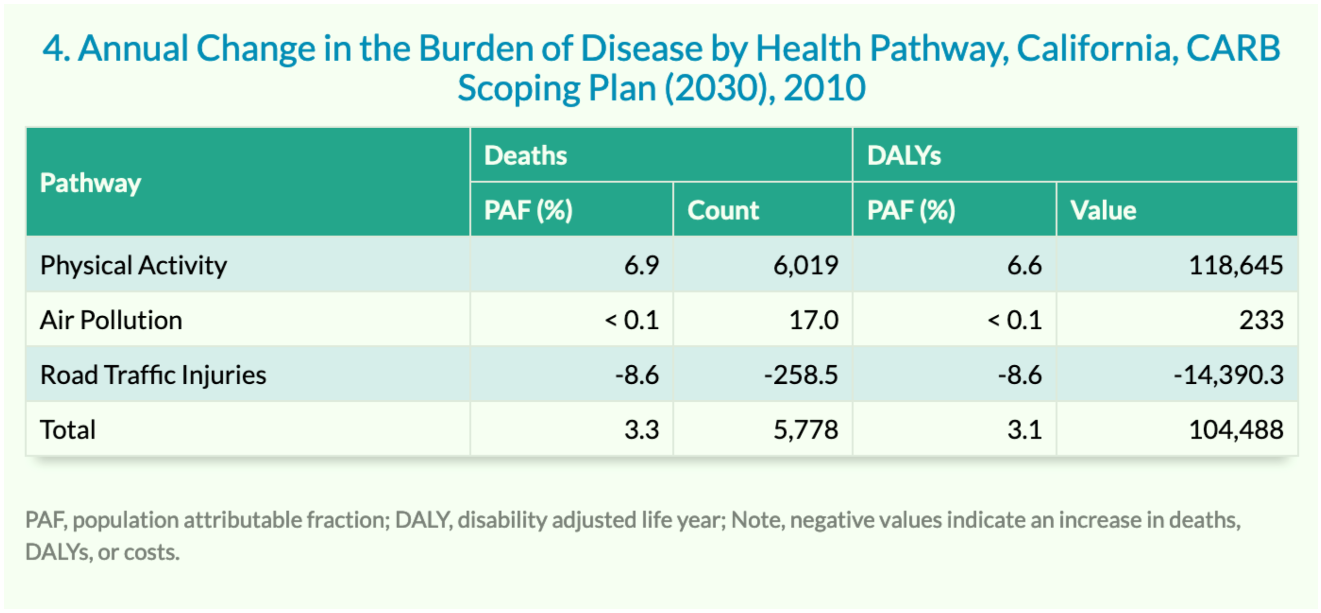
(Note, the structure function is one way to add attributes to an object.)

Readers will note that in fn_dataframer, in the High level of detail, HTab3 is conditionally generated. It will only be generated if output$vis is calling fn_dataframer as it needs just 1 disease. Otherwise, if output$CSV_prep is calling fn_dataframer, it supplies an argument l$specific_disease that informs fn_dataframer to skip the creation of HTab3. This occurs as output$CSV_prep needs to create data.frames for every disease! Rather than running fn_dataframer multiple times, output$CSV_prep simply runs fn_specific_disease multiple times.

The function fn_specific_disease does not do anything remarkably different from the other data.frame creating components of fn_dataframer. It is supplied with the l object (generated by run/the Analytic Engine), which also includes l$specific_disease, a data.frame with a single row. In turn, this data.frame supplies fn_specific_disease with all the necessary information to construct a specific disease data.frame. The one novelty fn_specific_disease presents is the option to create a table title. If fn_dataframer is calling fn_specific_disease (i.e. the output is for output$vis), then we'll generate a title in fn_dataframer. Otherwise, output$CSV_prep will request fn_specific_disease to generate the table titles (using the argument l$specific_disease_title).

In conclusion, both of these functions generate at least one data.frame with two attributes each. The returned variable will the be list object l, with either the element df_list or specific_disease_df disease appended to it. From here, every other function group will utilize the returned l$df_list to create the appealing visualizations users are familiar with in the ITHIM TOOL, or output$CSV_prep will utilize this returned l to generate a CSV file.

6.2. Group 2: HTML Tables



4. Annual Change in the Burden of Disease by Health Pathway, California, CARB Scoping Plan (2030), 2010

Pathway	Deaths		DALYs	
	PAF (%)	Count	PAF (%)	Value
Physical Activity	6.9	6,019	6.6	118,645
Air Pollution	< 0.1	17.0	< 0.1	233
Road Traffic Injuries	-8.6	-258.5	-8.6	-14,390.3
Total	3.3	5,778	3.1	104,488

PAF, population attributable fraction; DALY, disability adjusted life year; Note, negative values indicate an increase in deaths, DALYs, or costs.

- fn_formattr_col
- fn_formattr_df
- fn_colspans
- fn_tableTagger
- fn_tableManager

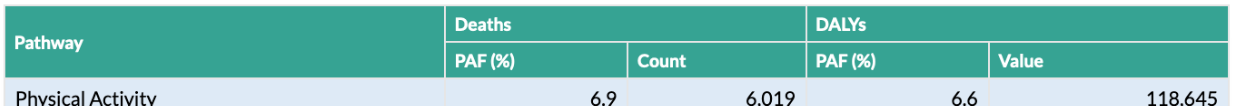
The key function in this group is fn_tableManager. It calls other functions in this group. It is however, *the* function that the {shiny} modules call in order to generate HTML formatted tables. As with the remaining function groups, fn_tableManager relies on the list of data.frames generated by fn_dataframer. It takes this list and then runs this list through fn_formattr_df and fn_tableTagger. The first function, fn_formattr_df, and the related fn_formattr_col, have fairly straight forward purposes, to convert numeric columns in each data.frame from fn_dataframer to character columns with those numbers in a preferred, presentable format. For instance, the value 3.141592654 might be simplified to 3.14. Whereas, the primary purpose of fn_tableTagger serves is to generate HTML code to represent the tables (i.e. the data.frames).

The function fn_formattr_df formats a single a list of data.frames (in l$df_list). It begins by storing the attributes of the data.frames it is going to format (as the formatting process deletes the attributes). It then applies fn_formattr_col to each column of each data.frame, while renaming each column name such that all periods are replaced by spaces (e.g. from “column.name” to “column name”).

Then, it reapplies the original attributes (i.e. the ID and table title). Finally, it conditionally fixes any certain table cell values (namely dollar values that are much less than $1 Billion).

The function fn_formattr_col takes a double (the data type; in this case, a column from a data.frame) and applies the base::format function to those (floating point) number values to make them more aesthetically pleasing. In essence, it shortens long decimal values to shorter numbers (with less than 3 digits following the decimal point). It will also replace NA values with the string "---", and NaN values with 0.

Now that the number columns in the data.frames from fn_dataframer have been formatted to look nicer, fn_tableTagger can turn the data.frame into an HTML table. However, before proceeding a few caveats should be addressed. First, there is one feature that has been left in place in fn_tableTagger that is not actually utilized: a citation system. Indeed, tables (including headers and rows), may include a citation (a superscripted number and footnotes). In fact, fn_tableTagger still checks for the presence of "XX" and will replace these values with a number if it is present. However, as of this writing, none of the column or row headers contain the string "XX". Second, some tables have spanning column headers (see figure below). As such, fn_tableTagger accommodates this in the construction of the HTML tables.



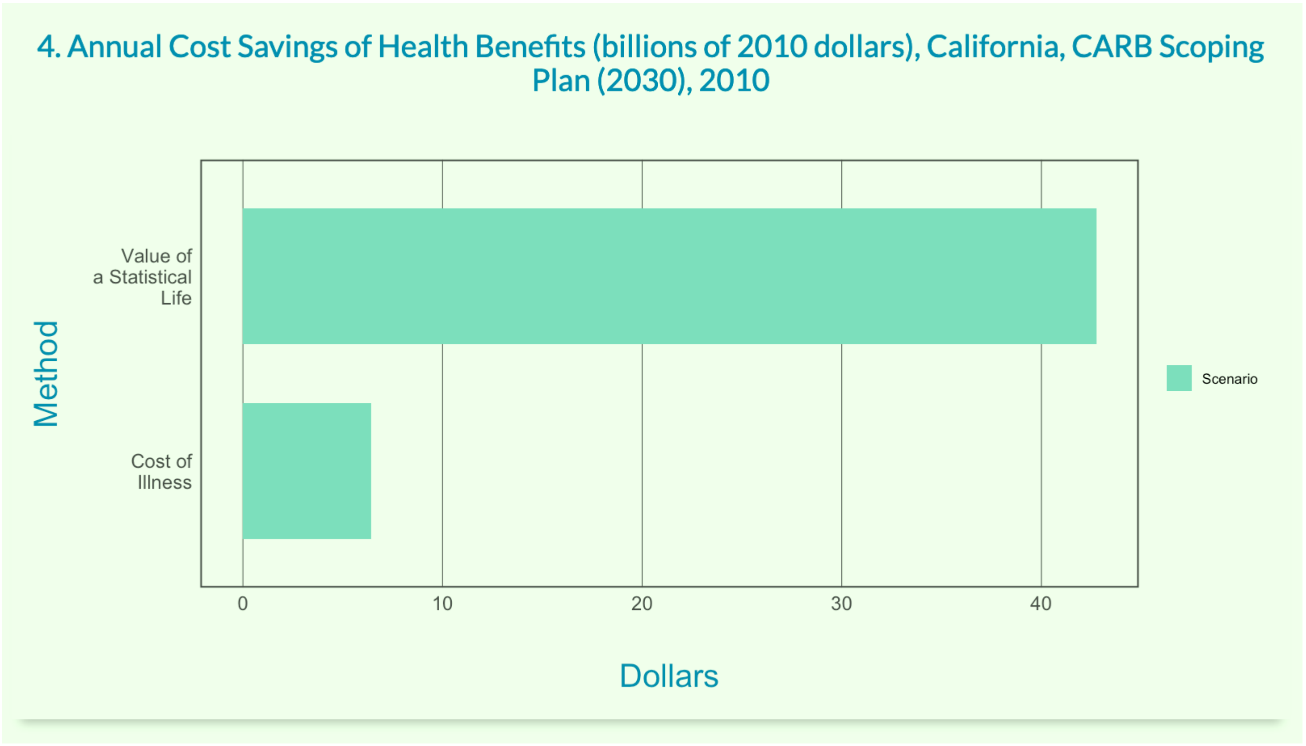
Pathway	Deaths		DALYs	
	PAF (%)	Count	PAF (%)	Value
Physical Activity	6.9	6,019	6.6	118,645

Figure. An example of a spanning column header. Note how both "Deaths" and "DALYs" span across two columns each.

The function fn_tableTagger begins by fixing geographies that are California counties by appending the word "County" to it. It then enters a loop to create one table at a time. It stores some attributes of the data.frame into variables to be utilized later. It then checks if it needs to build a spanning column header; if it does, it calls fn_colspans, which takes values from the "spanning_header" column in "tool_table_strings.csv" and splits this value into a useable data.frame. It then proceeds through an algorithm to assign each value in the spanning header data.frame to a table row (i.e. <tr>), and then generates the table header (i.e. <th>), all the while checking for citations along the way. Otherwise, if no spanning column headers are necessary, a simpler approach is utilized to make the table header. In both cases, the operative functions that actually generates the HTML tags are tags$th and tags$tr.

Following the creation of headers, the table-making loop continues by creating table rows, utilizing the tags$td function. Again, it checks for the presence of a citation. Next, the full title of the table, with all the breadcrumbs (e.g. the geography), is created (as a string). Then, it checks for footnotes. Finally, all the components are strung together in a list object. Here, the table title is placed in HTML tags utilizing tags$h4, and the headers and rows are combined in tags$table(tags$body(...)). If there are footnotes, they are added at this stage as well utilizing tags$p. (The footnotes are derived from "tool_table_strings.csv".) The function fn_tableTagger ends by returning the l object with the new element tagList. (The {shiny} renderUI function can take this tagList directly and then actually generate the HTML code in its final form.)

6.3. Group 3: Graphs



- fn_plotter
- fn_graphManager

Before proceeding with this section, the reader should be familiar with the {ggplot2} package. Grolemund & Wickham's R for Data Science contains an introduction to {ggplot2}, however, read Wickham's ggplot2: Elegant Graphics for Data Analysis for a more detailed guide to the package. Further, the cheatsheet on {ggplot2} may also be useful.

Unlike the Tables function group, the Graphs function group has no primary managing function. At its heart, the graphs are actually generated in their own base::local environments in the {shiny} app (and thus prevents the actual plot rendering from occurring in a function called in the global environment). Essentially, in order to render plots iteratively, {ggplot2} requires its own empty environment in a {shiny} app as it renders the images. Thus, the two functions defined in this function group help set up the data to be rendered as a plot, but the actual plotting is conducted by the function shiny::renderPlot, called in *tool_modules.R*.

```
for(ii in seq_along(o$objects)){
 local({
  local_i <- ii

  df <- o$objects[[local_i]]

  plotname <- paste(
   "plot",
   local_i,
   sep="")

  output[[plotname]] <- renderPlot({
   fn_plotter(df)
  })

 }) # end of local
} # end of loop
```

In the above code chunk (which occurs in *tool_modules.R*), the program iterates over a list of graph data.frames (produced by the fn_graphManager which is discussed below). For each plot, a local environment is created (in {shiny}). Eventually a new element is created in the output variable using renderPlot and fn_plotter (the latter is also discussed below).
Now, let's return to beginning of the creation of graphs (in *visualization.R*) now that we understand why the plots aren't being rendered in any of the *visualization.R* functions. The function fn_graphManager takes the output of fn_dataframer and then utilizes reshape to make it legible to {ggplot2}. However, fn_graphManager is responsible for other things as well. Much like fn_tableTagger it also creates the full title for each graph, appending such things as the geography. It also adds an attribute to each data.frame that will supply fn_plotter with the information it needs to either render a line plot or a bar plot. The function fn_graphManager also looks out for issues with units (e.g. kilometers vs miles). It finally stores this information back into l as the element objects, and then returns l.

The function fn_plotter on the other hand takes a data.frame as its argument (instead of l). Fundamentally, what fn_plotter returns is the object created by ggplot2::ggplot(). However, what happens in between is a matter that requires a brief explanation to casual {ggplot2} coders as in most cases, casual {ggplot2} coders are not familiar with non-standard evaluation. In most plots, {ggplot2} requires the names of certain elements of the graph, like what data represents the X and Y axes, or what the labels should read. However, as fn_plotter must be generalizable, it requires some dynamism. This does not present a problem for creating plot labels, but it does for accessing the data for X and Y (i.e. the columns from the data.frame being submitted for plotting). In order to get around this, we must first extract the relevant information about the columns to plot and then supply them in a fashion that ggplot() can understand. In the following code chunk we demonstrate the first part of this process by retrieving the column name for the X-axis. (Note, we arbitrarily used the Spanish words for X and Y, "equis" and "igriega" respectively, to label these variables. Also, please take note that the variable df is the data.frame argument supplied to fn_plotter.)

```
# X-axis
# Get the name of first column that fulfills these conditions:
# it is a character & is not ScenarioName
equis <- names(which(!(names(df) %in% "ScenarioName") &
            sapply(df, function(x) is.character(x))))[1]
```

After this bit of code runs, equis is a single value in a character vector (i.e. a single string or word). In order to dynamically insert these column names into ggplot to tell it to draw a graph, we have to use the function aes_string (see the code chunk below). The function aes_string takes a string and then uses non-standard evaluation on this expression. In the case of the example code chunk below, beside from evaluating equis we are also using the function reorder to change the order in which bars are drawn on the plot.

```
ggplot(
 data = df,
 # use aes_string to pass name of column in quotes as arguments.
 aes_string(
  x =
   # read following as: reorder(`x_name`, `y_name`)
   paste0(
    "reorder(`", # reorder by size of y
    equis,
    "`, `",
    igriega,
    "`)"),
  y =
   igriega
 )
) + ...
```

Hopefully it is self-evident that the above code chunk is only a snippet from the whole plotting process for bar plots. At the end of fn_plotter, a ggplot() object is returned, and then rendered by shiny::renderPlot. The renderUI for output$vis in *tool_modules.R* later constructs a tagList with the title of the plot, a horizontal rule (i.e. <hr>), and a call to the recently created plots using plotOutput(ns(nm)).

6.4. Group 4: Summary Report

- fn_report_calc
- fn_reporter

The Summary Report of the ITHIM TOOL is a fairly straight forward visualization. Essentially it creates an HTML table and fills it with either an image-icon or some text. The text is drawn from a template file, "report_template.csv", which contains a row for each row of the Summary Report. The text contained in that template has certain keywords that can be substituted with the scenario values. For instance, the first row describes an increase (or decrease) in the current levels of walking and cycling. The numbers are present somewhere in the output of run/the Analytic Engine, and which are then taken to replace the keywords in the report template. In order to accomplish this, the ITHIM TOOL needs to make use of fn_keyworder again, and a series of if-else conditional statements and comparators. For instance, if the value for the number of minutes of active travel is positive, the report should read an "increase" in active travel, whereas a negative value would have the report reading a "decrease" in active travel. This simple algorithm occurs in fn_report_calc, whereas the HTML is stitched together in fn_reporter.

6.5. Group 5: The Infographic

- fn_ig_DataAssembler
- fn_ig_textFormatter
- fn_ig_textGrobber
- fn_ig_panelArborist
- fn_ig_pngMaker

The Infographic has 4 components: the background images (png files); the text that is conditionally generated and placed over the background images; the way in which all of this is laid out with respect to each other; and finally the manner in which it is rendered. While the Infographic is not a plot, it does rely on some of the same dependencies of {ggplot2} in its construction. Namely, the Infographic utilizes the {grid} package.

The {grid} package is challenging package to master. It is highly recommended that readers look through chapters 6 and 7 of Paul Murrell's R Graphics (3rd Edition) prior to altering the Infographic (beyond modifying some of the basic content of the text). It is beyond the scope of this documentation to describe the finer points, such as what the grid::textGrob function achieves, or how a grid::gTree is utilized. The Infographic is quite sensitive, such that by even changing some of the background images by a few rows of pixels that it will alter the image noticeably, or by changing the background images to different aspect ratios, it may simply break the Infographic entirely.

6.5.1. The Design of the Infographic

This section will summarize the general {grid}-based design of the Infographic. Neither this nor the following sections are meant as a comprehensive introduction to {grid}, although they will endeavor to clarify parts that may be opaque to novice {grid} users where possible.

It is important to understand the way in which the Infographic is put together on a conceptual level first. Essentially, the Infographic is constructed from a number of "panels." This term, "panels", should not be misunderstood to be a term originating from {grid}, but rather a term used here to help describe the building blocks of the schema of the ITHIM TOOL Infographic. There are as many panels in the Infographic as there are PNG images in the /tool_files directory. Indeed, users of the ITHIM TOOL see a single Infographic image as the output of the TOOL, however that image is in fact a composite of 10 images stitched together vertically. As readers perusing the /tool_files may notice, the original PNG images are largely devoid of text as much of it is dynamically generated. In the R code, each panel's text and background image are constructed together first, and then stitched together. In fact, some of the panels are actually constructed in three parts, one for each scenario (i.e. the selected scenario, the US Surgeon General scenario, and the Low Carbon Driving scenario); the three parts are then stitched to form a single panel (which is then quilted together with the other panels).

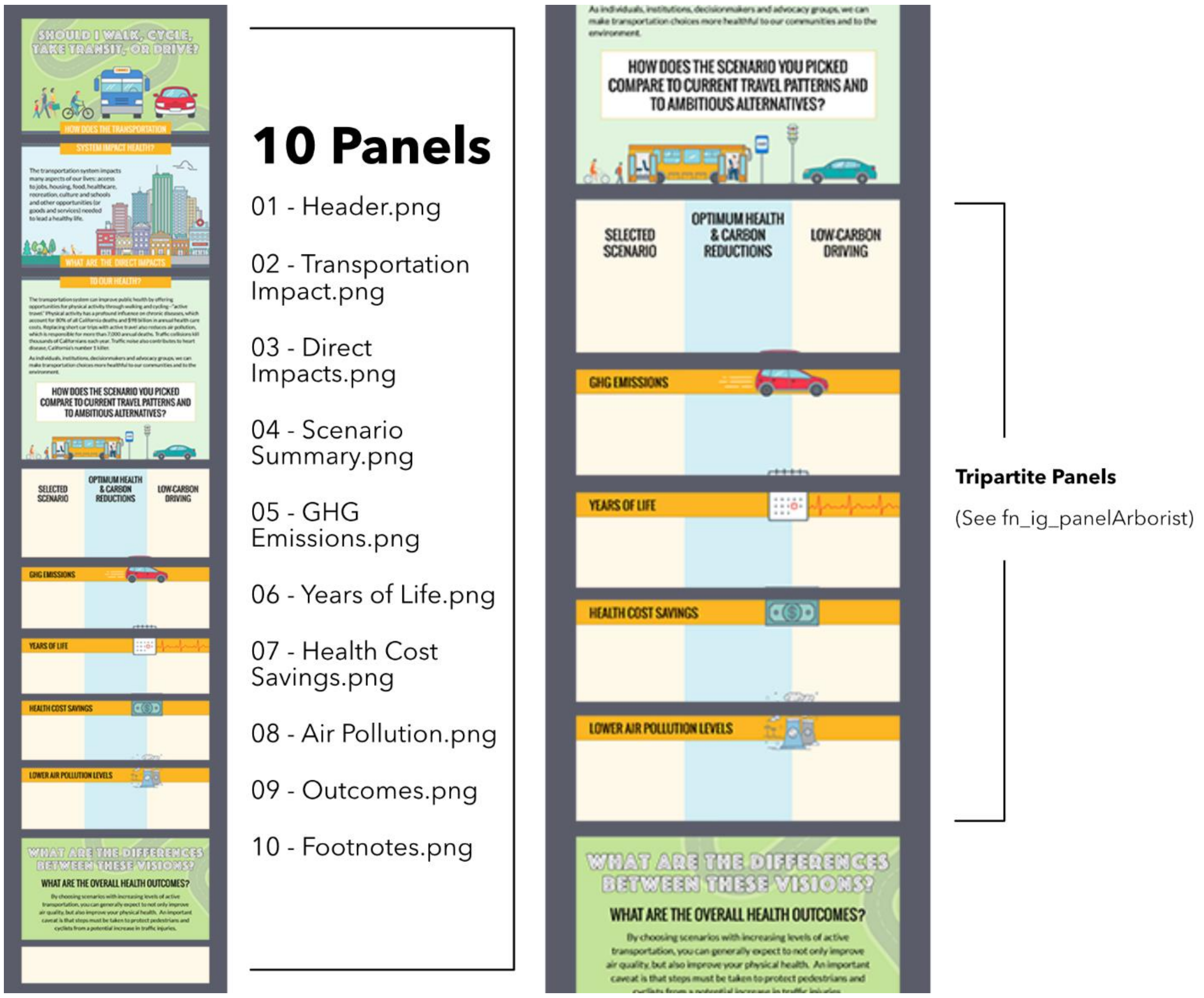

10 Panels
01 - Header.png
02 - Transportation Impact.png
03 - Direct Impacts.png
04 - Scenario Summary.png
05 - GHG Emissions.png
06 - Years of Life.png
07 - Health Cost Savings.png
08 - Air Pollution.png
09 - Outcomes.png
10 - Footnotes.png
Tripartite Panels
(See fn_ig_panelArborist)
HOW DOES THE SCENARIO YOU PICKED COMPARE TO CURRENT TRAVEL PATTERNS AND TO AMBITIOUS ALTERNATIVES?
SELECTED SCENARIO
OPTIMUM HEALTH & CARBON REDUCTIONS
LOW-CARBON DRIVING
GHG EMISSIONS
YEARS OF LIFE
HEALTH COST SAVINGS
LOWER AIR POLLUTION LEVELS
WHAT ARE THE DIFFERENCES BETWEEN THESE VISIONS?
WHAT ARE THE OVERALL HEALTH OUTCOMES?

Figure 6.2. An illustration of how the panels are divided. Note the lack of text in the Tripartite Panels (discussed below).

Both fn_ig_DataAssembler and fn_ig_textFormatter have little to do with {grid}. They are conceptually similar to what many other functions in *visualization.R* do: they take the scenario data (the output of run), process it, and then compile some data.frames or lists to be utilized by some other function(s) to eventually make a visualization. In fn_ig_textFormatter, readers will note that the function is building a list object named r (which incidentally stands for "row," but could be interpreted as "panel"). Once constructed, the r object has a series of parameters for each panel—upon which it can now place the text. Although this function does not call upon any {grid} functions, it does prepare for them.

The product of fn_ig_textFormatter is eventually submitted to our first function that makes calls to {grid} functions, fn_ig_textGrobber. This function calls grid::textGrob to build a *graphical object* (or grob) with text for each sub-panel in the tripartite panels. ("Tripartite" as there are three scenarios.) To clarify, in this function, the portion of the Infographic we're constructing is for the panels of various themes depicting scenario values, e.g. for Greenhouse Gas Emissions, Years of Life, etc.

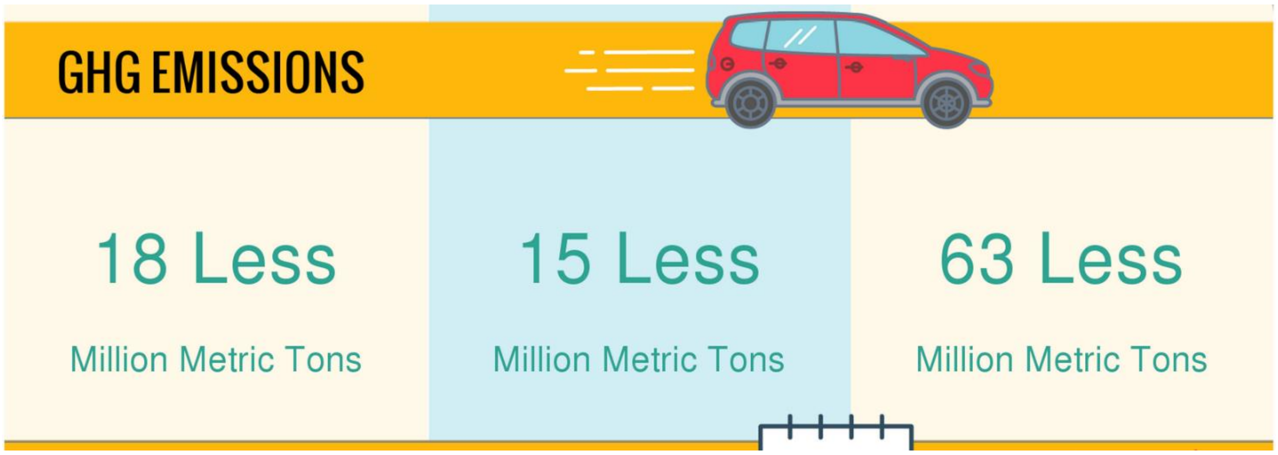


Figure 6.3. An example of a tripartite panel. This is one "panel" as described above, however it is composed of three parts: the selected scenario, the US Surgeon General, and Low Carbon Driving.

The function `fn_ig_panelArborist` takes each *text graphical object* (text grob) produced by `fn_ig_textGrobber` and sticks them onto their respective background PNG image (see the above figure). The actual output is a `grid::gTree` object with 5 `grid::rasterGrob` objects (i.e. the panels numbered 4-8).

Finally, in the function `fn_ig_pngMaker`, we combine the rest of the panels with the panels we made in `fn_ig_panelArborist` and draw it using `grid::grid.draw`. As a side note, readers should be aware that the actual "drawing" or rendering of the Infographic occurs in a `local` environment in {shiny} (see the section on renderUI). Please also note that {grid} objects are only "theoretical," so to speak, until they are drawn or rendered. Indeed, the image is not created until `grid::grid.draw` is invoked, and instead the {grid} object is merely a set of instructions that exist in the RAM. With all of this said, `fn_ig_pngMaker` does return a drawn graphical image, but is only turned into a PNG in {shiny} (namely, in `output$vis` in *tool_modules.R*).

6.5.2. Implementation of {grid} Functions

There are a myriad of ways to construct an image using {grid}. This section briefly outlines the main components of the use of {grid} in the ITHIM TOOL Infographic. Consequently, this section primarily examines the workings of `fn_ig_pngMaker`.

1. A new page is created with `grid.newpage`.
2. The first set of "header" panels are created using `rasterGrob`.
3. Next the tripartite panels are created by first creating the text using `textGrob`, then placing them onto the background PNG images using `rasterGrob`, and finally compiling them into a `gList` and placed into a `gTree`.
4. Then a simple `rasterGrob` (for "09 - Outcomes.png") is created.
5. The last panel is constructed for the "footnotes" as a `gTree`.
6. A layout object is created, using `grid.layout` with the dimensions of all 10 PNGs stacked vertically.
7. In the penultimate step, a `frameGrob` object is created, with the layout object from the previous step, and all the panels designed above.
8. Finally, the image is generated using `grid.draw`.

7. Startup Scripts

There are two sets of startup scripts in the ITHIM TOOL, the *global.R* script that resides in the root folder alongside the *app.R* file and the *preprocessing.R* script inside /tool_files. These start up scripts only operate once. Unlike the other scripts, they do not define functions utilized in the {shiny} environment, but rather load libraries, load these other R scripts, process data, and generate new variables in the global environment. It is customary in {shiny} to place the startup script in the same directory as the *app.R*, and the fact that the ITHIM TOOL split startup script into two is partly arbitrary, and partly a reflection of the desire to segregate TOOL-specific code in /tool_files to keep things organized.

7.1. global

The *global.R* script is fairly simple. It loads requisite libraries for the ITHIM TOOL and then loads the R scripts in /tool_files. One novelty worthy of mention is the design of the loading of libraries:

```
if(!("shiny" %in% .packages())) # is it already loaded in workspace?
 if(require(shiny) == F) { # load it, but if you can't...
  install.packages("shiny") # install it.
  library(shiny) # try loading it again.
 }
```

The script checks to see if the package is already loaded in the environment using `.packages()`; then tries to load the package using `require`, which always returns a boolean value; if that fails, it then attempts to install the package; and then it tries loading the library again. Generally speaking it is advisable to omit code that automatically installs things without the code user's knowledge (not to mention, the possible issues that may arise from conflicting or versioned dependencies). However, as this code was originally written to be executed by novice R users, it was designed to operate with the most ease.

7.2. preprocessing

The *preprocessing.R* script is loaded by *global.R*, and executes the startup code unique for the ITHIM TOOL. The general outline is thus:

1. Load the "data_dictionary.csv", which contains all of the csv files to read.
2. Isolate the `default_narratives` from the rest of the csv files.
3. Load the `default_narratives` (into a variable of the same name) and then separate them (into baseline and scenario data) using `fn_narrativeSplitter`.
4. Load the rest of the csv files into the global variable `d`.
5. Run the Data Integrity Check.
6. Create a scenario selection list for the dropdown menu in the TOOL (and place it in `d`).
7. Create the disease/cause lists for the Analytic Engine (and place it in `d`).
8. Create the variables for the Scenario Information Box in the left-hand column of the ITHIM TOOL (and place it in `d`).
9. Read the Infographic PNGs and record some metrics about them (and place it in `d`).

Most of this code should be self-evident in terms of what each step is attempting to accomplish. However, in the following sub-sections we'll explore two less transparent parts, the "data_dictionary.csv" and how it is generated; and the related matter of Data Integrity.

7.2.1. Data Dictionary

The description of *preprocessing.R* script is facilitated by an overview of the Data Dictionary. The Data Dictionary serves as a meta-data repository, containing such things as the name of the file that should be read for each variable,[1] the code levels, and other descriptive information. The Data Dictionary is automatically generated by the only R script that *global.R* does not automatically read. Indeed, this R script, *ithim_metadata_extraction.R*, must be run by the coder or user in developer mode. Whenever a change is made to the data, the coder/user/developer must run the *ithim_metadata_extraction.R*. [2] While the Data Dictionary is generated automatically, certain parts of it must be manually entered. For instance, the variables into which each file is read, and the column titled "ITHIM_Critical" must be written manually by the coder. The latter column represents variables that the Data Integrity algorithm is required to check upon startup. More on the Data Integrity algorithm is described in the next sub-section.

7.2.2. Data Integrity

The Data Integrity algorithm is a feature of the ITHIM TOOL that checks that the data (i.e. the csv files) have not deteriorated or changed between instantiations of the {shiny} app. In essence, data can deteriorate on web servers through a phenomena called bit rot. While the chances of this happening are quite low, the intention of this code is to prevent an error from going unnoticed and providing incorrect analyses to ITHIM users. This is particularly a concern where we use flat files (i.e. csv files) instead of an advanced database server that would check the integrity of the data on a regular basis automatically.

The algorithm utilizes the package {digest} to generate a cryptographic hash using SHA-1 . Each hash is a more or less unique signature of the data. This hash value is then stored in the Data Dictionary (by the *ithim_metadata_extraction.R* script). During the *preprocessing.R* script, the saved hash value in the Data Dictionary is checked against another generation of hash values (using `digest::sha1`) of the data being loaded in the following code chunk:

```
# Load all (non-narrative) data to default data.
d <- lapply(tmp3$Filename,
      function(x) read.csv(file.path(pth, x),
                      stringsAsFactors = F))
```

7.3. Changes

Make sure to follow this protocol each time you make changes to any of the csv files in /tool_files.

1. Make sure an existing “data_dictionary.csv” is present in /tool_files. (While the *ithim_metadata_extraction.R* script can easily generate a new “data_dictionary.csv” from scratch, you would then need to manually input all the values for the “Variable_Name” and “ITHIM_Critical” columns.)
2. Run the *ithim_metadata_extraction.R* script.
3. Check the “data_dictionary.csv” for missing values in the “Variable_Name” and “ITHIM_Critical” columns.
4. Re-upload the /tool_files directory to the {shiny} server (assuming there is a web server running the ITHIM TOOL).

Chapter References

1. Note, you can edit the filenames. Just rename the actual file, and then change the corresponding value in the data_dictionary.csv.

2. The *ithim_metadata_extraction.R* script must be run in the root folder of the project (i.e. where the app.R file resides). You can check that you’re in the right directory by using the command getwd. However, readers are strongly encouraged to create an R project in this directory and always run the app while this project is open. (The other assumption here is that you run this script from RStudio.)