A Summary of the Workflow of the Model

Peter Benoliel

June 23, 2023

# Model Purpose

The purpose of this version of the model is to determine the optimal location(s) and amount(s) of different type(s) of infrastructure in a bus network, while also determining the sizes of the battery packs of the vehicles that go along with that infrastructure. In the most general sense, the following represents a list of inputs that the optimization part of the model (the GMS file) expects:

* Timeline information (how many ‘time steps’ in an operating day)
* A list of vehicle type names, weights
* A list of route names
* A mapping of which buses are assigned to which routes (this will be explained in greater detail below)
* A list of candidate location names
* The time it takes for each trip on each route
* The energy use per timestep on each route
* The expected number of buses on each route at each time
* An indication of when a trip is starting on a route (I tend to call this a ‘trip start flag’)
* An indication of whether a bus on a particular route at a particular time is co-located with a candidate location (and therefore, is allowed to charge)
* The cost of energy, opportunity chargers, and depot chargers, and batteries (as a $/MWh figure)
* The rate of max charging for depot charging and opportunity charging (as MWh/time step)

This is a list of major outputs that can be read from the resulting file:

* Which vehicles were selected and what routes they are assigned to
* How much the system costs, both in terms of vehicles and infrastructure, as well as a lifetime (12 years of operating days) use of the system (note that no discount rate is applied)
* Energy flows throughout the system over an operating day
* And more…

The specific formatting of these inputs and outputs, as well as the way these are prepared will be discussed in greater detail below.

## A General List of Steps

The general process for running the model is:

1. A GTFS feed is selected and checked to ensure that it is “clean” (that data are in the correct format, that there are no significant missing data, etc)
2. The GTFS feed is ‘handled’ by Hanjiro’s code, developing an energy table (energy use per time step)
3. The number of buses to be used is determined.
4. The candidate locations are identified.
5. A variety of data frames are constructed in the particular way GAMS expects to receive data.
6. These data frames are written to a GDX file using the gdxtools package.
7. R invokes GAMS.
8. GAMS produces a new GDX file that can be read for its data.

Each of these steps is a significant oversimplification, so each will be discussed in greater detail.

# Step 1: Finding and Selecting GTFS Feeds

From the website, “The General Transit Feed Specification (GTFS) is an Open Standard used to distribute relevant information about transit systems to riders. It allows public transit agencies to publish their transit data in a format that can be consumed by a wide variety of software applications. Today, the GTFS data format is used by thousands of public transport providers.”[[1]](#footnote-1) GTFS comes in two ‘flavors’: GTFS Schedule (previously referred to as a Static Feed) and GTFS Realtime (previously referred to as a Dynamic Feed). These two feeds are self-descriptive: GTFS Schedule contains the timetable information, as well as service schedule and other static information about a network, while GTFS Realtime can provide real-time updates on delays and vehicle positions. This model exclusively makes use of GTFS Schedule (the static feed).

These feeds can sometimes be found on the websites of their respective agencies, though there is no standardized place that it is stored. As two examples, Unitrans has its feed available by going from the main website to the About page and scrolling the left sidebar menu to the bottom, where a GTFS page is linked. In contrast, SJRTD’s feed can be found by scrolling to the bottom of the homepage and clicking the “Developer Portal” link, which includes (among other things) a link to the GTFS feed. However, this is a slow and difficult process, so it is preferred to use a database with links that point at these feeds. For my dissertation, I used the TransitFeeds[[2]](#footnote-2) archive, but this archive is no longer updated. The Mobility Database[[3]](#footnote-3) has picked up the mantle of maintaining that. The database is still in development, but the critical functionality for this project (that is, having a list of accessible feed URLs) is in place. The GTFSEnergyAnalyzer package has a function for importing these feeds based on this URL[[4]](#footnote-4).

Once the feed is imported, there are some checks that need to be performed on the feed to ensure that it is suitable (in its current form) for use with this model. The first check is to ensure that a file was actually downloaded; some of the URLs may point to either corrupted files or non-existent files. If a file is downloaded successfully, the data need to be checked to ensure that they are within the GTFS specifications, with all required files present. If it is, there are certain checks for data integrity to ensure that even if the feed ‘looks’ like it is compliant, that the present data actually reflect the information they are supposed to. This requires a degree of familiarity with GTFS data, as feeds that fall through the cracks at this stage tend to do so in subtle ways that may not be immediately obvious (this is an issue with the Mexico City feed, where the file stop\_times.csv did not contain the information it was supposed to, but looked like it did). If it appears that the feed is good, one of the other things that can disqualify a feed from being used is the practice of ‘combining’ transit agencies into a single feed. This is within specification, but a way to handle these combined feeds has not been implemented yet. After downloading the GTFS feed for the first time, the GTFS feed is saved as an RDS (R Data Serialization) file which can be accessed for future runs, and may also be referred to when creating figures at the end.

# Step 2: Develop the Energy Table

The energy table is developed through a series of functions based heavily on code written by Hanjiro Ambrose. First, the process of prepping the GTFS feed for the functions that develop the energy table will be discussed. Then, the theoretical background of the energy table function will be covered.

## GTFS Lengthr

The first function that “doctors” the raw GTFS feed is called gtfs\_lengthr[[5]](#footnote-5). This function takes a raw GTFS feed as an input and adds the length data for the routes using the Google Maps API (note that the user must supply a key before this function is called or it will error out). The function works in “batches”; this functionality was coded by Hanjiro, so I have two guesses as to why it does so: First, it could be a debugging tool that was simply left in the function; Second, it could be an intentional choice to limit the number or rate of “simultaneous calls” to the Google Maps API. One call has to be made to the API per unique trip segment between stops, so for large networks, this can mean thousands of calls. Batching these calls may have (or used to have) advantages either in ensuring a prompt response from the API, or for billing purpose, or for some other reason. I have generally left the batch value at its default value (50) and have experienced no problems as a result. After this process is completed, the function builds a new “stop\_times” dataframe and inserts it into the provided gtfs.obj, which it then returns.

## GTFS Framer

Fundamentally, GTFS Framer is a disentangling function that takes a GTFS object and ‘disentangles’ it from a list of dataframes into a single (large) dataframe that contains the specific information required to build the energy table. Specifically, the route information (including name, distance from gtfs\_lengthr, service and trip IDs, start times, end times, and duration).

## GTFS Calendar

Building the calendar for the network makes use of two functions: year\_tabler and day\_tabler[[6]](#footnote-6). The year\_tabler function uses the calendar\_df to fill in whether or not there is service for each specific day in the calendar based on the default operations for the day of the week. Afterwards, calendar\_dates\_df is used to fill in specific operational exceptions (for example, special service days or holidays). Days where service occurs are labeled with a 1; days with no service are designated with 0. This calendar is used as an input to day\_tabler. This function uses that calendar of service to return a list of trips that occur on each operating day.

## GTFS Summary

The final step for preparing the energy table is to run gtfs\_summer[[7]](#footnote-7). This function takes the day table from day\_tabler as well as the GTFS object produced by gtfs\_lengthr and produces a summary dataframe that contains the relevant values for the energy table to be produced. The methods of this production are relatively straightforward from reading the function directly. Make sure to cross-reference the data names across the relevant functions to ensure that what is being summarized is not lost.

## Modeling the Energy Use and Producing the Table

Modeling the energy use of buses on the route is done via linear regression against data gathered as part of the FleetDNA project by NREL[[8]](#footnote-8). Most of the data in the transit bus dataset are fairly old at this point; we are actively looking for another data source that tracks the relevant parameters to use. So far, one hasn’t been found, so we are continuing to use FleetDNA as it is still a representative dataset for ZEBs. Physically, the model is based on the classic physical model of a vehicle, which has the following equation:

Where m is the vehicle mass, a(t) is the total net acceleration on the vehicle at a given time t, ρ is the density of air (assumed to be the density of air at sea level for this model), CD is the coefficient of aerodynamic drag for the vehicle, Af is the frontal projected area of the vehicle, v(t) is the velocity of the vehicle at time t, Crr is the coefficient of rolling resistance for the vehicle, g is the gravitational constant (9.81 m/s2), θ is the angle of the road the vehicle is driving on, and FT(t) is the tractive force on the vehicle (that is, the force being provided by the engine or motor, or the force of braking, regenerative or otherwise). For this model, we are not considering any hills, so the road grade component of the equation can be ignored.

# Step 3: Determine the Number of Buses

The number of buses is determined by invoking the check\_bus\_number[[9]](#footnote-9) function and multiplying it by an ‘adjustment factor’. check\_bus\_number essentially calculates the minimum number of buses that would be required to service the route given the following assumptions:

1. Each bus is assigned to a single route and serves only that route.
2. Buses have infinite energy capacity and need not recharge.

This is essentially designed to represent ‘ideal’ operating condition, where the number of buses is affected by only the number of routes and the relationship between the time it takes to complete the route and the headway of the route (that is, if the headway of the route is longer than the time to complete the route, only 1 bus is needed. If the headway is shorter, then more buses are required). This ‘minimum number’ is multiplied by an adjustment factor. In my survey of transit agencies, I found that (especially for the agencies of the size I was studying), most agencies operated with about 30% more buses than this theoretical minimum number. During my sensitivity/scenario analysis, I also used values of 1.1 and 1.5 as “less vehicles” and “more vehicles” scenarios.

# Step 4: Identify Candidate Locations

Candidate locations are identified by invoking the cand\_loc\_searcher[[10]](#footnote-10) function. This function identifies bus stops as “candidate locations” in one of two ways: either candidate locations are selected until all routes are served by at least one candidate location, or a specified number of locations to find are selected. In either case, locations are selected by ranking stops based off of the total number of trips that stop at that location (the idea being that if more buses stop at a location, there are more chances to charger a bus, and therefore the chargers are more economically efficient). If a specific number of candidate locations is specified, the function simply selects that number of stops from the top of the ranked list. If the ‘all routes’ method is to be used, then the top selection is added to the list of candidate locations, and the function notes which routes are serviced by that stop. Then the model checks the routes served by the second-most-stopped-at stop. If that stop has at least one route that is not serviced by the route already selected, then that stop is added to the list of candidate locations. If no new routes are serviced, then that stop is skipped and the next stop is checked. The function continues in this manner until all routes are served by at least one candidate location. In either case, the list of locations is then returned. The function also produces helpful visualizations of the network and candidate locations (see the manual entry for more information).

# Steps 5 and 6: Construct Data Frames and Write to a GDX File

GAMS is very picky about the data it receives, and the gdxtools package is likewise picky about the format of the data it is sent. Here are a few notes to follow when prepping data:

1. All data must be sent to write.gdx[[11]](#footnote-11) as a list of data.frames. This includes the sets, even though they can be expressed as vectors.
2. A note must be made on how the variable is defined within the GAMS model. If (for example), a parameter is defined thus:

Param(set1, set2, set3)

The data.frame containing that parameter must have columns named for set1, set2, and set3, specifically in that order, followed by a column named “value” which contains the value of the parameter at that particular combination of set values. The sets must also repeat in a specific way as shown below (assuming that all sets have values 1, 2, 3:

|  |  |  |  |
| --- | --- | --- | --- |
| set1 | set2 | set3 | value |
| 1 | 1 | 1 | V1 |
| 1 | 1 | 2 | V2 |
| 1 | 1 | 3 | V3 |
| 1 | 2 | 1 | V4 |
| 1 | 2 | 2 | V5 |
| 1 | 2 | 3 | V6 |
| 1 | 3 | 1 | V7 |
| 1 | 3 | 2 | V8 |
| 1 | 3 | 3 | V9 |
| 2 | 1 | 1 | V10 |
| 2 | 1 | 2 | V11 |
| 2 | 1 | 3 | V12 |
| … | … | … | … |

Repeating in this way until all values are input.

1. Note that the names of the sets and parameters must match exactly with how they appear in the GAMS model.

Currently this process is automated in the gdx\_prep\_rw2 (rw2 intending to communicate “rewrite number 2” to differentiate it from a “rewrite 1” that is not used at any time in the model)[[12]](#footnote-12). This produces two lists (called sets and params), which are fed into write.gdx to produce the GDX file that the GAMS model expects.

# Step 7: R Invokes GAMS

Invoking GAMS is done using gdxtools. To invoke GAMS, it must first be initialized using igdx and the location of a version of GAMS that has a valid license (this points the gdxtools functions towards the correct GAMS API to call for the rest of its functions). After this is done once, GAMS can be invoked using the gams function, pointing to the .gms file containing the model. GAMS will then run in the background. For this model, the runtime is limited to an hour for each network. GAMS will occasionally throw errors and these codes will be returned to R. A value of 0 indicates a successful GAMS run. Any other number corresponds to some kind of error that must be looked up (for example, 7 will be returned is GAMS is invoked before it is initialized using igdx).

# Step 8: GAMS produces a GDX File Containing Outputs

As written, the model currently produces a file named “outputs.gdx” for every time it is run. To ensure that the next network doesn’t overwrite the output file of the previous network, R is used to rename (and possibly relocate) the file based on the agency name and/or scenario that was run. This GDX file can be read using the gdx function in gdxtools. Note that GAMS must be initialized to read GDX files using igdx. Individual sets, parameters, and variables can be called using the syntax:

gdx(filename.gdx)[varName]

This will load a dataframe of the variable, with all of the values at any set or parameter values it depends on. These dataframes can be used as the basis for any further analysis or graphics production. When I was making graphs based on this model, I also referred to both the energy tables and the network RDS file to get static information about the network (number of routes, operating hours, etc).

1. For more information, see https://www.gtfs.org/ [↑](#footnote-ref-1)
2. https://transitfeeds.com/ [↑](#footnote-ref-2)
3. https://database.mobilitydata.org/ [↑](#footnote-ref-3)
4. Function name: gtfs\_loader. First function in the “loaders.R” document within gtfsenergyanalyzer. [↑](#footnote-ref-4)
5. Located in the “dataPreppers.R” file in GTFSEnergyAnalyzer [↑](#footnote-ref-5)
6. Both located in the “calendar.R” file in GTFSEnergyAnalyzer [↑](#footnote-ref-6)
7. Located in the “dataSummary.R” file in GTFSEnergyAnalyzer [↑](#footnote-ref-7)
8. For project details, see the FleetDNA website: https://www.nrel.gov/transportation/fleettest-fleet-dna.html [↑](#footnote-ref-8)
9. Located in the “utils.R” file in PeterResearchPackage [↑](#footnote-ref-9)
10. Located in the “dataPreppers.R” file in GTFSEnergyAnalyzer [↑](#footnote-ref-10)
11. Part of the “gdxtools” package. [↑](#footnote-ref-11)
12. Located in the “gdx\_preps.R” file in “PeterResearchPackage”. [↑](#footnote-ref-12)