A Prototype City Generation Framework for Simulating Future Mobility Scenarios Across Global Urban Typologies

by

Iveel Tsogsuren

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Moshe Ben-Akiva (Jimi Oke)
Edmund K. Turner Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Masters of Engineering Thesis Committee

A Prototype City Generation Framework for Simulating Future Mobility Scenarios Across Global Urban Typologies

by

Iveel Tsogsuren

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The goal of this project is to develop prototype cities that represent urban typologies worldwide, for the purpose of simulating future mobility scenarios. In ongoing efforts, we have discovered nine driving factors based on data from 331 cities across the world. Using these, thirteen distinct urban typologies resulted, each representing a unique mobility outcome. In order to assess the impacts of future vehicle technologies and environmental policies in these typologies, simulation-ready prototypes are required as test-beds in our state-of-the-art urban simulator, SimMobility. In my thesis, I outline the data and methods harnessed in building a pipeline for the generation of these prototype cities. As a realization of the proposed pipeline, I synthesize the Auto-sprawl prototype city, which represents the urban typology where cars are the dominant modeshare across a large metropolitan area, and public transit availability is limited. The candidate real-world city used for generating this prototype is Baltimore, Maryland. I show consistency of the generated results by comparing the generated data with the real statistical data. Finally, I demonstrate that this work is being utilized for running simulations in SimMobility and generating additional simulatable prototype cities.

Thesis Supervisor: Moshe Ben-Akiva (Jimi Oke)
Title: Edmund K. Turner Professor of Civil and Environmental Engineering

Acknowledgments

I dedicate this thesis to my grandfather Altangerel Khand.

I would like to express my sincere gratitude to my advisor Jimi Oke for all the support and wonderful discussions. To Carlos, Youssef and Eytan, I am grateful for your ideas, support and contributions. To Professor Moshe Ben-Akiva, thank you for giving me this opportunity.

To my family, thank you so much for your unconditional support through all these years. I would like to thank my amazing friends Surya Tripathi, Lkhamnyam Turbat, Zixi Liu, Ishwar Kohale, Chandan Sharma Subedi, Sapna Kumari and Kishore Patra for their love and care.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Chapter 1

Introduction

Agent-based microsimulation model systems for land use and urban and transportation planning have come into widespread use [2], [32] [31], [10], [8], [4]. One of the unavoidable steps in these studies is to generate baseline data: population individual and physical environment. Variety of studies and methologies have been developed on the synthesis and generation of cities for agent-based mobility simulation. However, these are mostly city-specific or only sythesize population or road network. My thesis project aims to create an automated pipeline for generating urban typology prototypes. These prototypes are simulation-ready environments that represent outcomes for the urban typologies discovered from global data collected in the first phase of the Mobility of the Future project [26]. As synthesizing urban typology provides flexibility over choosing cities and data, my goal is to develop highly generalized approach and rely only on open data. The outputs from my project serve as inputs in our state-of-the-art integrated laboratory−SimMobility [2]. As a realization of the proposed pipeline, I synthesize Auto-sprawl typology for which Baltimore metropolitan area with 2.5 million inhabitants is chosen as the candidate city. I show consistency of the generated results by comparing the generated data with the real statistical data.

1.1 Motivation

Transportation accounts for about 19% of energy consumption worldwide [26]. The energy demand for transportation is substantially growing to an extent that the existing urban structure cannot support. This problem can be addressed with a better assessment of future mobility. "Mobility of the Future" project under the Massachusetts Institute of Technology Energy Initiative is studying impacts of future mobility services and vehicle technologies around the world [26]. The main approach of the study is to conduct urban simulations under variety of government policies, technologies, new services and evolving attitudes, and evaluate alternative transportation models, and discover possible scenarios.

In ongoing efforts, [19] has discovered nine driving factors based on data from 331 cities across the world and identified thirteen distinct urban typologies resulted, each representing a unique mobility outcome. In order to assess the impacts of future vehicle technologies and environmental policies in these typologies, simulation-ready prototypes are required as test-beds in our state-of-the-art urban simulator, SimMobility.

My role in this project is to create a pipeline for synthesizing simulatable prototype cities: the road networks and populations to conduct simulations. With some degree of freedom, we choose a city with high membership probability from each typology and generate their simulation ready road network and population. The urban typology and simulation framework are discussed in the following sections.

1.1.1 Global Urban Typologies

The urban typology, for which I create prototype cities, is a probabilistic city classification which deploys a novel supervised approach, i.e. Latent Class Choice Model, [19]. Based on urban and individual-specific behavioral data for 331 worldwide cities, this study recognized nine most relevant factors: metro propensity, bus rapid transit (BRT) propensity, bike share, network size, growth, congestion, network density, auto-industrialization and efficiency. Given these factors, thirteen typology classes

are identified with distinct city characteristics and travel-related choice preferences, which are listed in Table 1.1. Further normalized nine factor in these typologies are shown in Figure 1-1. Moreover, these thirteen prototypes reveal variations in preferences for future vehicle technologies, and can be used for variety of mobility assessments and modeling around the globe. The validation of these thirteen typologies is still ongoing.

Table 1.1: Prototype Candidates for Supply

#	**Urban typology**	**Example cities**
1	Hybrid Moderate	Havana, Panama City, San Jose,
2	BRT Giants	Samara, Durban, Shiraz
3	Emerging Green	Daejeon, Lisbon
4	Congested Boomer	Bangalore, Chennai
5	Metro-Bike Giant	Shenzhen, Guangzhou
6	Metro-Bike Emerging	Ningbo, Zhenzhou, Shenyang
7	BRT Giant	Istanbul, Rio de Janeiro, Tehran
8	BRT Moderate	Leon, Guayaquil, Tabriz
9	Auto Innovative	San Diego, Miami, Toronto
10	Auto-sprawl	Baltimore, Raleigh, Richmond
11	Innovative Heavyweight	Singapore, Madrid, Seoul
12	Sustainable Moderate	Sapporo, Sendai, Daegu
13	Sustainable Anchor	Dublin, Helsinki, Tel Aviv, Copenhagen

Figure 1-1: Thirteen typology profiles across the nine factors obtained from the data. The pilot typology for the generation process described in this thesis is Auto-sprawl [34]

1.1.2 SimMobility: an agent-based urban simulation platform

Generated prototype cities serve as inputs in SimMobility urban simulation platform. This platform integrates various mobility-sensitive behavioral models with land-use, transportation, and communication interactions and predicts the impact of mobility demands on transportation networks, intelligent transportation services, and vehicular emissions [2]. Thus, it enables the simulation of a portfolio of technology, policy, and investment options under different future scenarios. An overview of this process is described in Figure 1-2.

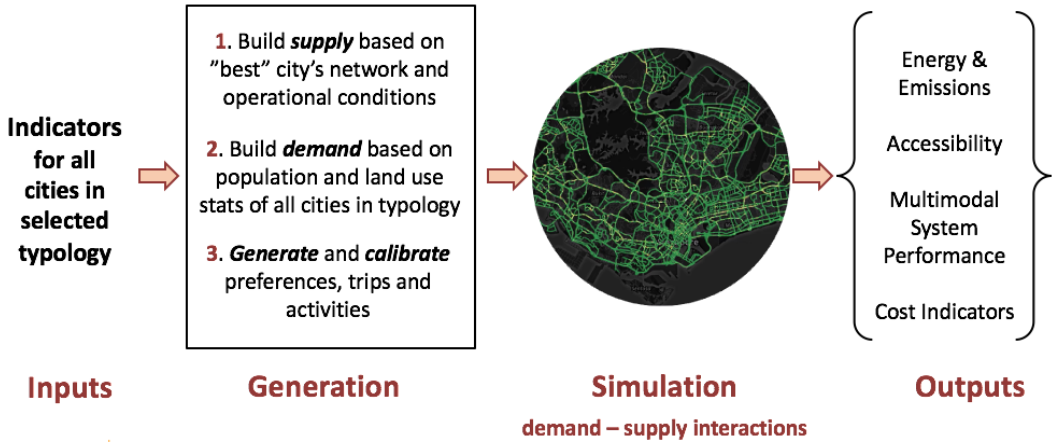


Figure 1-2: Procedure: Quantitative mobility measurements from urban typologies

SimMobility uses millions of agents from pedestrians to drivers and performs second-by-second to year-by-year simulations across the area of interest [2]. As described in Figure 1-3, this framework integrates different inter-connected temporal levels: short-term (ST), mid-term (MT) and long-term (LT). The ST simulator represents events and decisions on the order of tenth of a second such as lane-changing, braking or accelerating. The MT simulator represents events on the order of seconds or minutes such as daily activity schedule, mode, route, destination or departure time choices. The LT simulator represents long-term choices such as house or job relocation. Each of multi-level simulators requires different data. For example, the ST simulator needs road network, traveler trip chain, and driver characteristics while MT simulator needs synthetic population with socio-demographics and derived day

activity schedules. The focus of this project is on generating inputs for the MidTerm simulator.

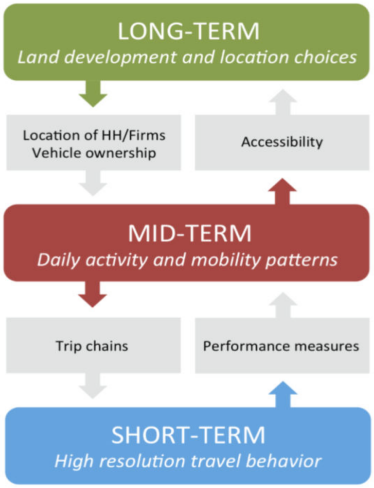


Figure 1-3: SimMobility Framework [2]

Chapter 2

Literature Review

Variety of studies and methodologies have been developed on synthesis and generation of cities for urban studies and computer graphic researches (e.g.: gaming and movies). In computer graphics research, Parish and Muller initiated a procedural approach to urban modeling and many derivative works have since been created [30]. They created urban road network as well as buildings in 3D layout. However, their purpose is to mainly mimic city views rather than accurately represent city road network and population layout. Xiaoming Lyu et al. presented procedural modeling to generate three urban layouts: population, land-use and road network on the selection of population density map and road network map [25]. This method gives a explicit control over outputs, i.e. population, area of the city, and desired land-use percentages. However, their procedure is still under development and does not provide any way to develop transportation network, which is essential for urban simulations.

The existing studies for an agent-based simulation on road network, population synthesis and population allocation are generally separated. I discuss each of them separately in the following subsections.

2.1 Road network

An agent-based simulation requires both road network and public transportation network, which are always connected. To the best of our knowledge, there is not any

solid methodology on synthesizing representative road and public transportation networks. Thus, I rely on using real road network and overlay transportation system on top of the road network in order to have both road network and public transportation network.

2.2 Population synthesis

Population is defined by individuals with socio-demographic attributes grouped by households. However, disaggregated data of population is not available due to privacy concerns. Population synthesizing methods have been developed for disaggregating population attributes based on known population distribution, using population samples. There are two dominant approaches for population synthesis: reweighting and synthetic construction [20].

Synthetic construction can generate micro data with only aggregated totals ([24], [5]). This approach is used when sample data is not available. Reweighting method scales samples while matching their totals to aggregated totals. Since sample data has information of interconnection of socio-demographic attributes, reweighting approach is preferred for population synthesis. The original work in this approach is Iterative Proportional Fitting (IPF), which iteratively expands a small microdata sample to match marginal totals and gives a unique solution. However, IPF fills only either household or person level marginal totals. In reality, household structure is necessary to simulate personal decisions which are affected by their household membership. There are distinguished approaches such as Hierarchical Iterative Proportional Fitting, Entropy Maximization and Iterative Proportional Updating which extend IPF and satisfy both household and person level controls.

Another method used is the MCMC-based sampling [14]. This method estimates true joint distribution of population attributes. Farooq uses Gibbs sampling to generate a population based on prior knowledge of joint distribution and shows the sampled population is indeed statistically quite similar to the original distribution. They find that getting full conditionals was not trivial, and therefore proceed by replacing some

conditional distributions with marginal distributions. However, when the number of attributes increases, conditional distributions are not trivial to find.

2.3 Population allocation

Population distribution is strongly related to data sources such as land use, mobility data, travel flows, employment density, road network density and land prices [35], [7], [13]. There are lots of studies on mapping population distribution [22], [23]. The most common and simple method used is interpolating population density with spatial factors and population census data. Night-time satellite imagery and statistic aggregate in administrative units are used to refer population distribution. They have a coarse resolution scale, which is not fine enough for micro-level urban models. Using high resolution remote sensing images and datasets for individual buildings, some studies generate population distribution on a fine scale, like 100 meters [3], [6], [33]. However, data for these methods are limited and it is impossible to apply this approach on developing countries. Some other studies use OpenStreetMap tags, road network density and population, but they require lots of data processing and can be time consuming. Moekel et al. (2003) present an intuitive approach for assigning households to zones [27]. First, they obtain land-use data, which they use to disaggregate zonal characteristics (e.g. household location or firm location) into raster cells. The size of these cells is chosen based on computational tractability and desired resolution. Weights are then assigned to the cells based on the corresponding land-use category.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Overview of SimMobility Data Requirements

This thesis project involves utilizing open source data to synthesize city population and road network. The outputs from this are fed to SimMobility simulator. SimMobility takes a range of input such as socio-demographic population, road network, public transit network and various other factors. I focus on synthesizing population, road network and public transit network as inputs for SimMobility. In the following sections, I provide an overview of required input data to SimMobility and various available open resources used to create those data.

3.1 Road Network Data

There are accurate and rich real-world road network sources, but most of them are copyrighted and owned by companies, which includes Google Maps and Apple Maps. These sources have limited access and restrictive terms of service. OpenStreetMap is a collaborative mapping project that provides a free and publicly editable map [18]. Its data quality is generally high although data coverage varies worldwide [17]. Furthermore, OSM provides important metadata of roads, like type of road, speed limit and number of lanes. These are essential for synthesizing SimMobility road network. Thus, I deploy OSM as a source and create SimMobility specific road

network. SimMobility road network layer is composed of nodes, links with segments, polylines and lines as described in Figure 3-1 and listed in Table 3.1.

Component	Definition
Links	Contiguous streches of single-direcional roads
Nodes	End points of links
Segments	Sub-divisions of a link. These are typically based on changes in the link geometry, such as changes in the number of lanes
Lanes	Parts of segmetns that are designated for use by a single line of vehicles
Connectors	Connection between lanes of two consecutive segments within the same link
Turning path	A path connecting specific lanes of two connected links
Turning group	A set of all turning paths that connect the same pair of links
Turning conflicts	Overlapping points of two turning paths that have the same traffic phases
Poly-line	A sequence of points defining the shape of road network constructs such as segments, lanes, links and turning paths

Table 3.1: SimMobility road network building block

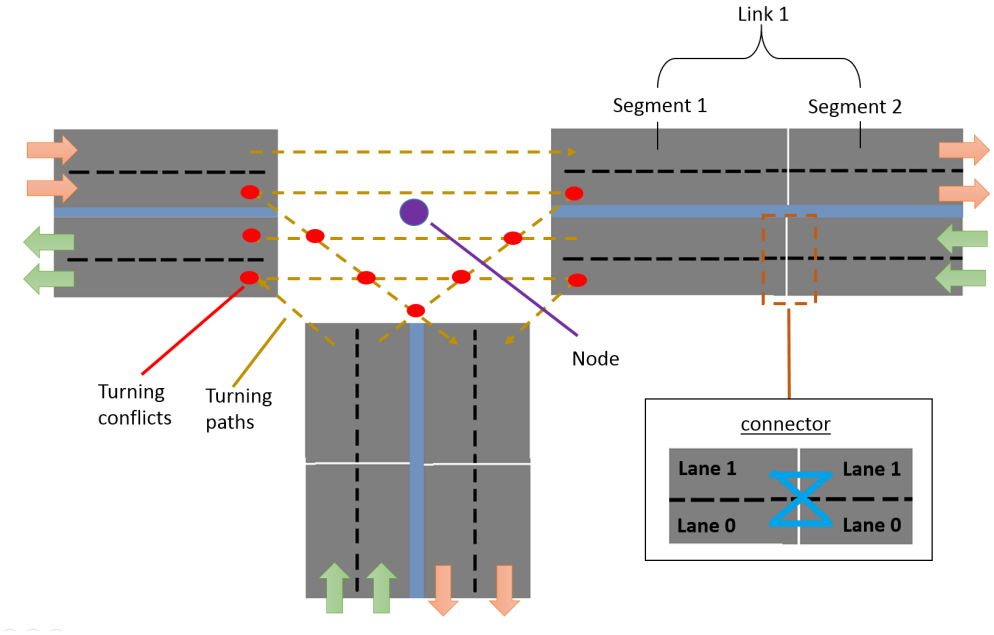


Figure 3-1: SimMobility road network network building blocks [15]

3.2 Public Transit Network

The General Transit Feed Specification (GTFS) provides public transportation schedules and associated geographic information in a common format and it is publicly available. It is a collection of related CSV file and describe a transit system's sched-

uled operations. These files are enough to unambiguously determine bus and train lines in SimMobility. Yet, expressing bus routes in SimMobility segment components, which represent sections of homogeneous roadway, is not straightforward.

3.3 Population Data

Population with socio-demographics in SimMobility is an entire population within which each individual has a unique ID and socio-demographic attributes, e.g., age, home, work/school locations and a grouped household, as listed in Table 3.2.

	column	description
Individual	id	individual (person) id
	employment_status_id	person type id
	gender_id	gender of person
	education_id	student type category id (mapped to education level)
	vehicle_category_id	vehicle ownership type
	age_category_id	age category id (mapped to age range)
	income	income of person
	work_at_home	whether the person works from home
	car_license	whether the person has licence to drive car
	motor_license	whether the person has licence to drive motorcycle
	vanbus_license	whether the person has licence to drive heavy vehicles
Work/school	time_restriction	whether the person has strict work hours
	fixed_workplace	whether the person has a fixed work location
	is_student	is the person a student
	sla_address_id	address of primary activity location (school location if student, work location if employed)
Household	id	id of household to which the person belongs
	sla_address_id	person's home address id (mapped to geographical location)
	size	size of person's household
	child_under4	number of household members under 4 years of age
	child_under15	number of household members under 15 years of age
	adult	number of adults in household
	workers	number of earning individuals in the household

Table 3.2: SimMobility socio-demographic population table [15]

In practice, data for a full population is unavailable due to privacy concerns. In order to synthesize this data, researchers use commonly available data that can be divided into two types: aggregate and disaggregate. Aggregate data are the numerical totals of records, often by category (e.g., total number of men and women), and are referred to as marginal data [21]. Aggregate data come in the form of one, two, and/or multi way cross tabulations describing the joint aggregate distribution of socio-demographic variables at the household and individual levels. Disaggregate data, on the other hand, are composed of individual records (e.g., persons, households, or vehicles), and are referred to as microdata, i.e., anonymized samples of individual responses.

In the following subsections, I present available marginal data and microdata for cities in the US as an illustration.

Marginal Data Most cities have available marginal census totals. In the U.S., marginal census totals are available from the following sources provided by the U.S. Census Bureau:

1. The decennial census is a complete census produced every 10 years.

2. The ACS is an ongoing sampling program which publishes estimated tables every year.

Microdata sample Microdata may be available from a census or from a traditional activity-travel survey that may have been conducted by a planning agency in a region. Microdata is compiled and anonymized for public use. In the United States, microdata is available through the Public Use Microdata Sample (PUMS). The PUMS are five percent disaggregate population samples collected as part of the ACS. The smallest spatially allocated areas available in the PUMS are Public Use Microdata Areas (PUMA). PUMAs are areas that contain at least 100k persons in the population, which is relatively large compared to census tracts used for synthesis, but smaller than most counties.

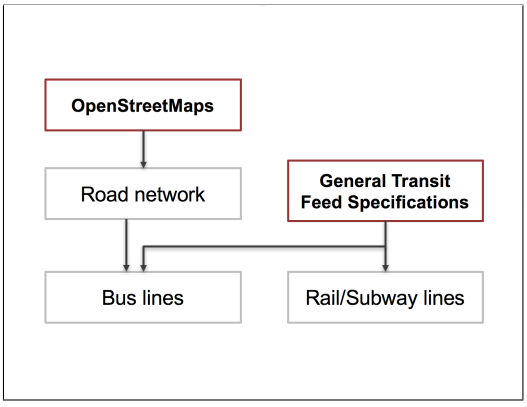
Land-Use Data We deploy land-use data to spatialize population realistically. One of the applicable sources is Traffic Analysis Zone (TAZ), which is a unit of geography used in transportation planning models. TAZ is constructed by census block information, and contains socio-economic data. Furthermore, land-use and cover maps, which classify land area into distinct types of land-use, such as low-density residential, medium-density residential or high-density residential, could be used to geographically locate home, school, and work locations.

THIS PAGE INTENTIONALLY LEFT BLANK

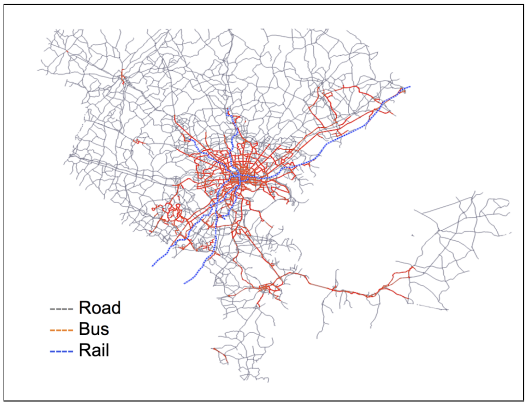
Chapter 4

Road Network and Public Transit Network

I generate road network and public transportation network (bus and train lanes) for SimMobility. Using OpenStreetMap, I create a road network. I then use this road network and General Transit Feed Specifications to create Bus lane network. Similarly, I create Rail/Subway lane network using General Transit Feed Specifications only. These steps are shown in Figure 4-1 (a). The final output is presented in Figure 4-1 (b).



(a) Road network generation pipeline



(b) Baltimore Metro Area Transit Network

Figure 4-1: GTFS process flow and result

4.1 Road Network Construction

SimMobility requires road network whose lanes and turning paths are precisely specificied. Using OpenStreetMap, I build a road network tool, which allows users to query a particular road network by specificying an area and road type, and create SimMobility compatible road network, which is represented by Tables: A.17, A.16, A.14, A.19, A.21, A.22, A.23, A.18, A.15, A.20 and A.24.

Creating SimMobility compatible road network from OSM has multiple steps. I download a road network map from OSM, make network topology corrections and represent the road network as a graph. Generating a graphical representation allows to make necessary modifications conveniently. To convert this into SimMobility compatible road network, I synthesize some road network features such as lanes and turning paths. These procedures are described in details in the following subsections.

4.1.1 Road network extraction

I extract (download) from OSM because of its flexibility and quality. There are a few ways to download OSM road networks and I choose Overpass API because it allows to filter out unwanted information in the query process. I define two kinds of queries: "all roads" and "main roads," and resulting road network map are shown in Figure 4-2. 1. The "all roads" queries every road, except specific types.

Listing 4.1: All roads query

```
["area"!~"yes"]["highway"!~"cycleway|footway|path|pedestrian|
steps|track|proposed|construction|bridleway|abandoned|
platform|raceway|service"]["motor_vehicle"!~"no"]
["motorcar"!~"no"]["access"!~"private"]["service"!~"parking|
parking_aisle|driveway|private|emergency_access"]
```

2. The "main roads" queries only certain types of roads.

Listing 4.2: Main roads query

```
["area"!~"yes"]["highway"~"motorway|trunk|primary|secondary|
tertiary|motor_link|trunk_link|primary_link"]
["motor_vehicle"!~"no"]["motorcar"!~"no"]["access"!~"private"]
["service"!~"parking|parking_aisle|driveway|private|
emergency_access"]
```

(a) All roads

(b) Main roads

Figure 4-2: Road network of Baltimore city, Maryland, the U.S. obtained from OSM

4.1.2 Conversion from extracted OSM to Graphical representation

OSM represents road networks by ways and nodes. From OverpassAPI, I receive JSON data of OSM ways and nodes along the ways as shown below.

Listing 4.3: OSM node

```
{"type": "node",
"id": 61328038,
"lat": 42.360057,
"lon": -71.107667,
"tags": {
    "attribution": "Office of Geographic",
    "created_by": "JOSM",
    "source": "massgis_import_v0.1_20071008165629"}}
```

Listing 4.4: OSM way

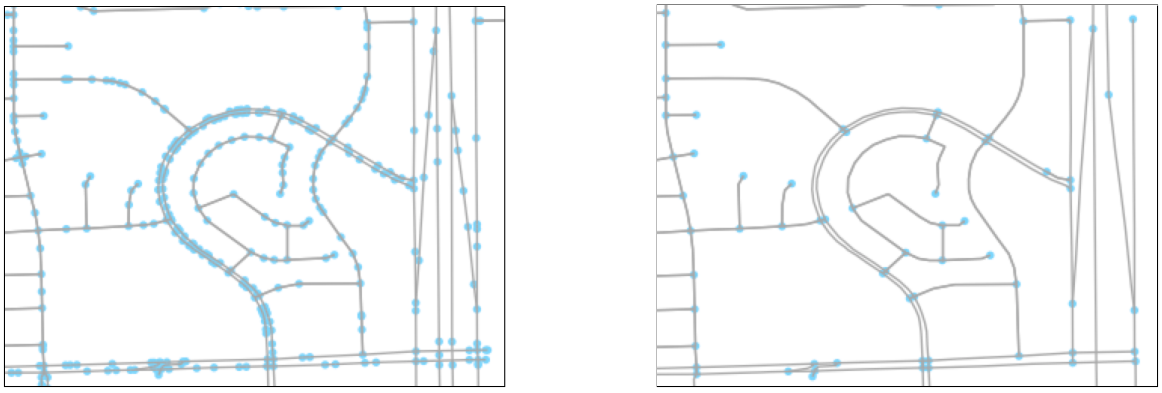
```
{"type": "way",
```

```
"id": 387051949,
"nodes": [61327276, 597845243, 61327122, 61320984],
"tags": {
    "attribution": "Office of Geographic",
    "condition": "fair",
    "highway": "residential",
    "lanes": "2",
    "massgis:way_id": "129000",
    "name": "Franklin Street",
    "oneway": "yes",
    "source": "massgis_import_v0.1_20071008165629",
    "width": "12.2"}}
```


OSM nodes include both intersections and the points along a single street segment where the street curves. Since the non-intersection points are not nodes in the graph theory, I remove them while preserving shape of the segment delineated by them. I do this using OSMnx, a Python package for downloading administrative boundary shapes and street networks from OpenStreetMap. However, I modify OSMnx functions so that I can apply only necessary modifications [9]. Illustrative OSM and graphical road networks are shown in Figure 4-3.

4.1.3 Feature synthesis

A simMobility road network is highly detailed, covering lanes, lane connections, and turning paths. Using OSM lane tags, I create multiple lane curves for roads. For turning paths at a specific intersection, I shorten road ends at the intersection such that these roads (lane areas) do not overlap, and then connect every outgoing road to every incoming road. The resulting road network is shown in Figure4-4.



(a) Downloaded OSM ways (grey) and nodes (blue)

(b) Derived MultiDiGraph: roads (grey) and junctions (blue)

Figure 4-3: Different road network representations

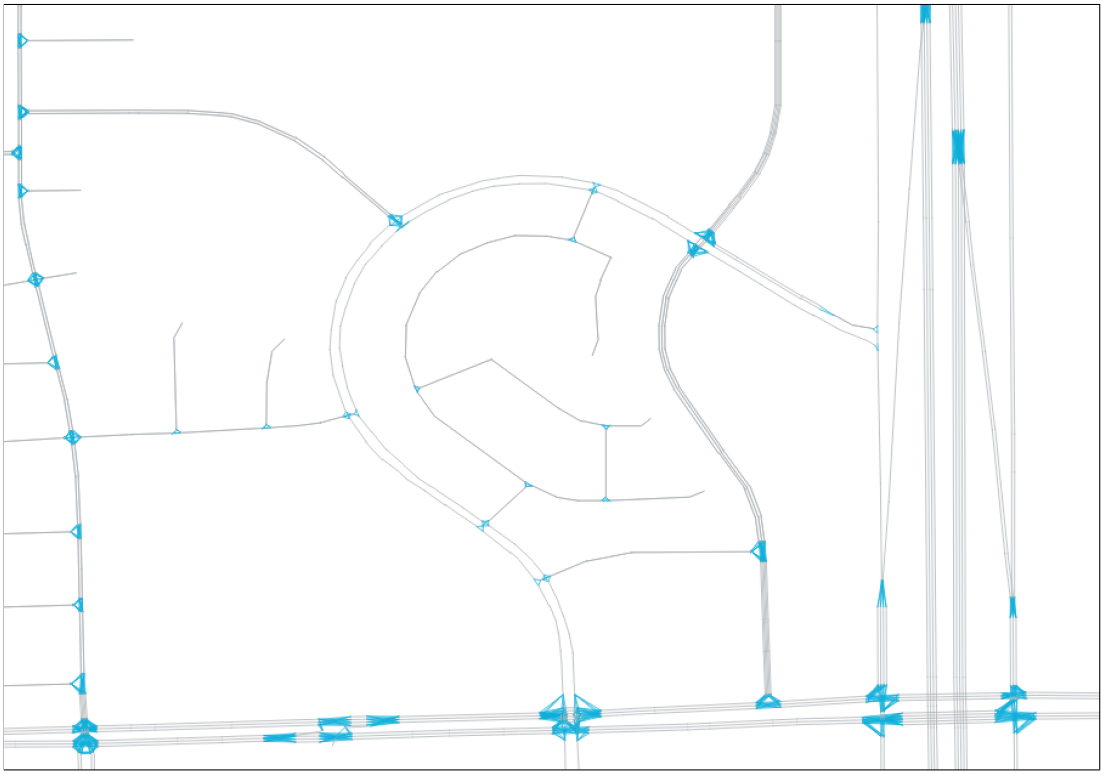
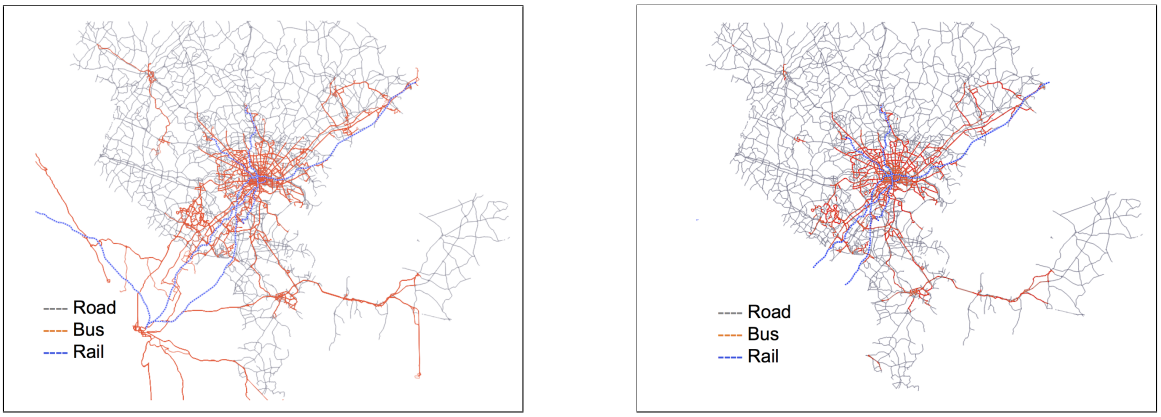


Figure 4-4: SimMobility road network representation: lanes (grey) and turning paths (blue)

4.2 Public Transportation Network Construction

Public transportation includes city buses, trolleybuses, trams (or light rail) and passenger trains, rapid transit (metro/subway/underground), categorized into Bus and Train for SimMobility. I synthesize bus network based on both the SimMobility road network and GTFS, and train network using GTFS only. Using the followed approach, I create public transit networks and get results, listed in Table 6.1.



(a) Original GTFS routes

(b) Synthesized public tranist network

Figure 4-5: Public Transportation Network

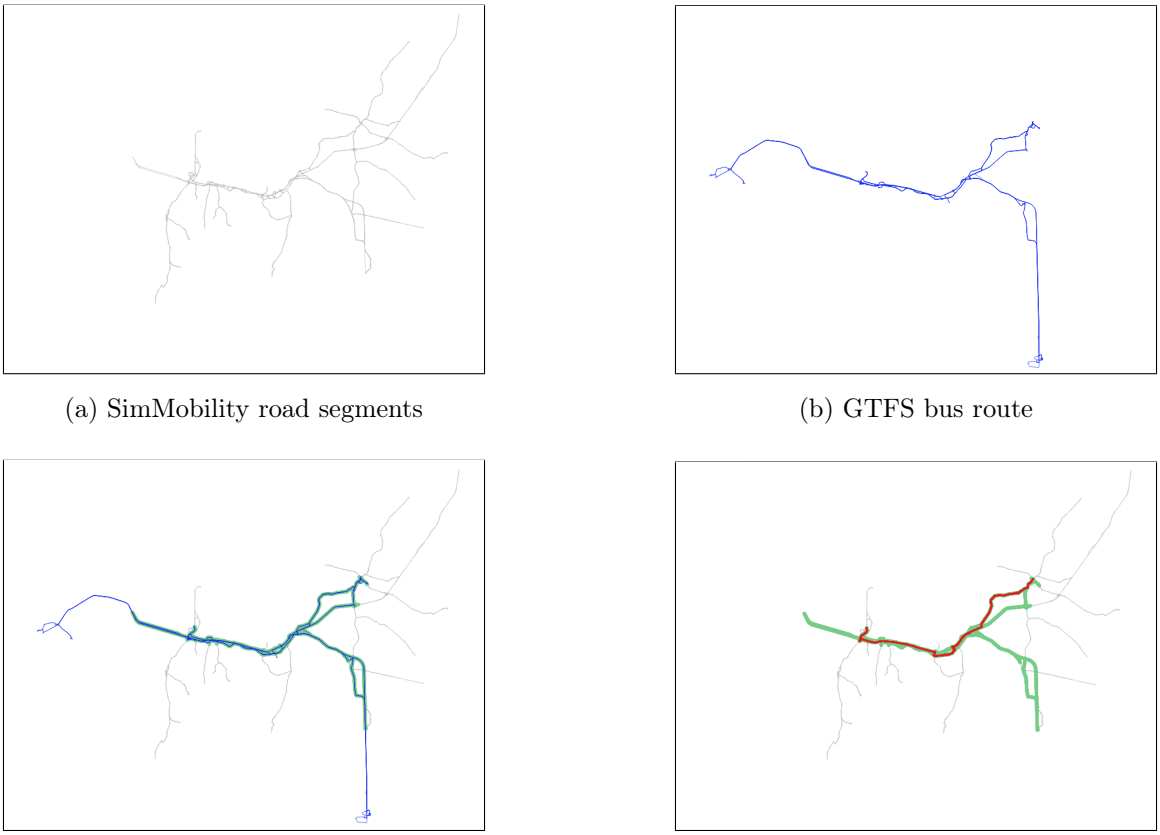
	Processed	**Real**
Number of bus stops	870	2500
Number of bus lanes	150	300
Number of train stops	76	76
Number of train lanes	10	12

Table 4.1: Synthesized prototype city public transit state

4.2.1 Bus lanes

I map bus lanes from GTFS to SimMobility compatible road network to synthesize Bus lane network compatible for SimMobility. However, available road network does not necessarily cover all roads. As a result of this, some bus routes might possibly not be mapped to any road network. I trim out those bus trips and synthesize the rest. SimMobility bus lane tables are defined in Appendix: A.3, A.1 and A.2.

In Figure 4-6, an extreme scenario is shown where a big portion of bus route is completely off from the road network.



(a) SimMobility road segments

(b) GTFS bus route

(c) Intersection of SimMobility road segments and GTFS bus route

(d) Bus route along the road segments

Figure 4-6: Road segments and bus routes

My procedure for mapping bus route to roads is descried below.

1. Convert road network segments into a graph so that we can search shortest paths through segments. Since SimMobility segments do not have explicit from and end vertices, we create new vertices.

2. Find segments which are in (50 meters) buffered bus routes.

3. Find segment end a representative (nearest) vertex for each bus stop. We prefer vertices because it does not constrain direction of bus stop while segments do.

4. Find an unique bus stop sequences whose paths we construct by segments.

5. Find shortest segment paths for each consequent bus stops.

6. Construct bus trips and prune out short trips which have less than 5 stops.

4.2.2 Train lanes

Train road network, which is treated as an independent layer, is synthesized from GTFS and converted to SimMobility compatible format (i.e. Tables: A.4, A.6, A.5, A.6. A.12, A.9, A.7, A.8 and A.10)

My procedure for mapping bus route to roads is described below.

1. Clean out GTFS data

2. Identify road network route shapes.

3. For given shapes and train stops, create train route network with pairs of opposite directions

4. Split the road into blocks such that each stop is on a unique block. Except end stops, stop is in the middle of block. Terminal stops are at at an end of its corresponding block.

Chapter 5

Population Synthesis and Allocation

In this section, I describe the process of population synthesis and distribution and address specific employment and school enrollment data for SimMobility using Baltimore metropolitan city as an example. However, population at the individual level and its allocation does not exist due privacy concerns and must be synthesized. As such, I synthesize socio-demographic population using available samples and marginal totals. Then I allocate the population residential addresses based on land use data. Similarly, I allocate work employment and school enrollment using land use data. In the following sections, I describe population synthesis procedure, land use data nature and population and work/education allocations. The overview of the workflow is shown in Figure 5-1.

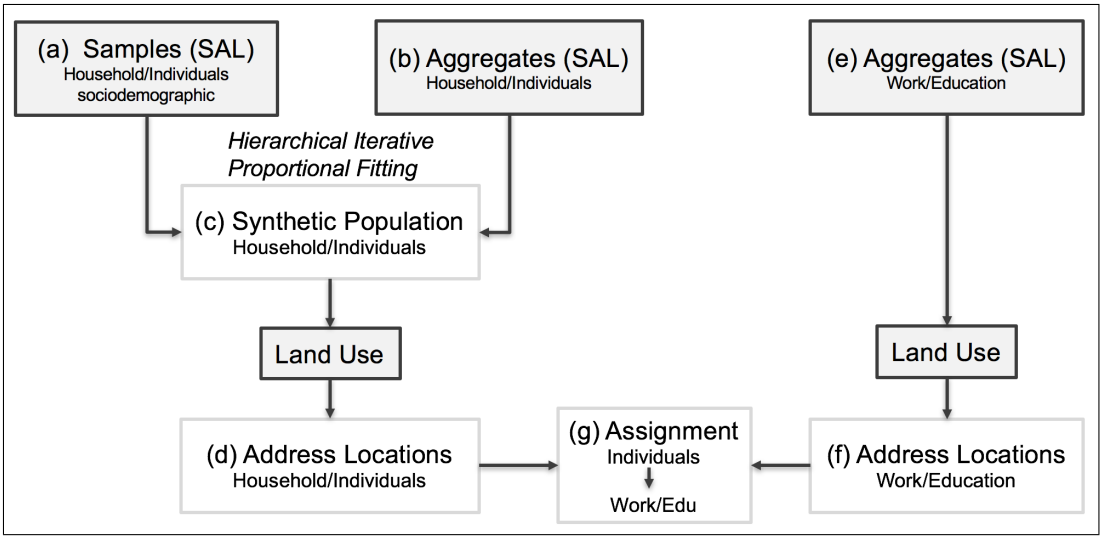


Figure 5-1: Population generation workflow

5.1 Population synthesis

As shown in Figure 5-1, I utilize available microsamples (a), and aggregated totals (b), on each attribute of interests, i.e. gender, age, household income and vehicle ownership, for population synthesis (c). I scale a given microsample to match aggregate totals, to represent whole population. To do this, I deploy Hierarchical Iterative Proportional Fitting (HIPF) procedure which iteratively and alternately fits samples at household and person levels [29], and is available under MultiLevelIPF package [28]. It estimates household-level weights that fit the control totals at both person and household levels. This eliminates the need to account for person-level control during the generaion of synthetic households. Based on the resulting weights, I draw synthetic population.

5.1.1 Population synthesis

I synthesize Baltimore metropolitan population covering seven counties: Anne Arundel County, Baltimore County, Baltimore City, Carroll County, Harford County, Howard County and Queen Anne's County. Since we could acquire county level totals, we synthesize seven county populations separately and then merge them.

Microdata sample preparation

I obtain Public Use Microdata Samples (PUMS) [1] which cover only five percent disaggregate population samples of the Baltimore metropolitan (Baltimore-Columbia-Towson, MD), and are collected as part of the ACS. The samples are compiled and anonymized for public use. This microsample of Baltimore metropolitan consists of approximately 6303 persons and 3002 households. For the MultiLevelIPF R script, we must format samples in the format shown in 5.3.

Aggregated data preparation

Under MultilevelIPF method, I control numerical total of records. These totals can be at either household or individual level. I collect marginal data from The American Community Survey (ACS) [11]. I create table files for each category totals in the format shown in Figures 5.1 and 5.2. Category should be coded into categories with their counts. If an attribute has numeric value, then we can create range bins, like {0: <10'000$, 1: 10'000-20'000$, .., 15: > 500'000$}. Total number of households and individuals must be consistent in these marginal total tables.

Table 5.1: Marginal total at household level

household income	**total**
0	3059
1	9820
3	4211
4	1830

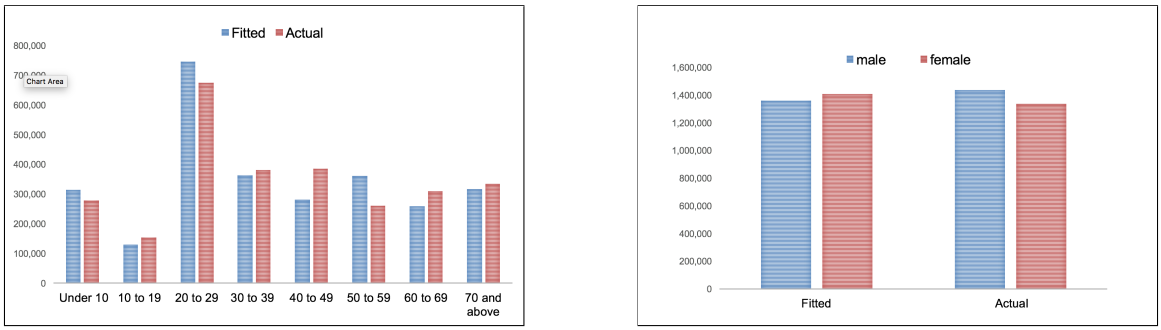
Table 5.2: Marginal total at individual level

Gender	**total**
0	35249
1	38300

Table 5.3: Microdata sample

Household_id	individual_id	APER	gender	age	edu	vehicle	household_income
10001	1	1	0	8	6	1	3
10002	2	2	0	8	10	1	5
10002	3	2	1	1	1	1	6
10003	4	1	1	0	0	1	8
10004	5	1	0	0	0	1	9

After generating the household weights, we duplicate households with their individuals proportionally to match the total population. Individuals and households can have extra attributes which are not controlled. Following this procedure, I synthesized Auto-sprawl urban typology cite and its basic statistics are listed in 5-2.



(a) Age distribution

(b) Gender distribution

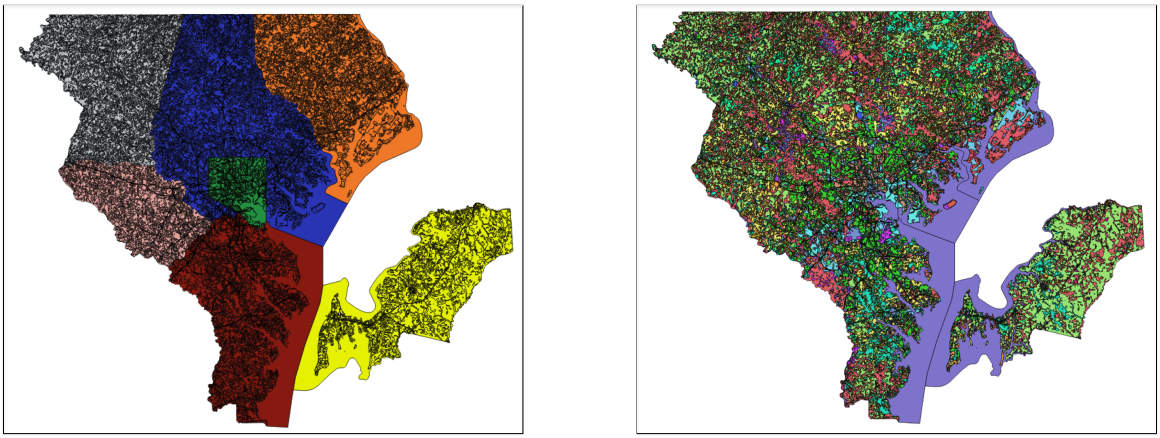
Figure 5-2: Auto-sprawl (i.e. Baltimore metropolitan area)

5.2 Population allocation

In this section, I determine allocation of number of people at specific locations, i.e. address points, in Baltimore metropolitan based on residence. I then perform similar allocations for school enrollment and employment in Baltimore separately. For performing these three tasks, I utilize land use data and grid point addresses, and use linear programming with certain constraints.

5.2.1 Custom Land Use Categorization

Land Use data refers to division of a certain area (i.e. Baltimore metropolitan) into various categories based on how the landscape has been altered by human activities. The categories that we use have been derived from "Maryland Department of Planning Land Use/Land Cover Classification Definitions" which consist of 13 main category types with 23 total subtypes [12] . We then map these category types into our custom twelve categories, which are: agriculture (A), commercial (C), high residential (HR), medium residential (MR), low residential (LR), sparse residential (SR), education (E), industrial (I), forest (F), openland (O), water (W), transportation (T) as listed in Appendix A.25. Based on Baltimore's land use data handout, an estimate for the number of residents, employees and students in each category was calculated.



(a) Baltimore Counties

(b) Baltimore land use classification

Figure 5-3: GTFS process flow and result

Table 5.4: County level statistics

County	County code	Map color	Total population	Employment
Anne Arundel	24003	red	323655	267123
Baltimore city	24510	green	457735	337533
Baltimore county	24005	blue	513941	373120
Carroll	24013	grey	95423	57783
Howard	24027	pink	177327	167909
Queens Anne	24035	yellow	26988	14788

Harford	24025	orange	146580	91376

5.2.2 Grid point address

In order to find reasonable address point allocations, I make a Grid of the map of Baltimore metropolitan with equidistant points at a distance of 350 meters, as shown in Figure 5-4. These address points are travel origin and destination points in SimMobility. This distance was calculated assuming that the neighborhood of an address point within 350 meters does not dramatically change the travel path to that point and that most of the land use divisions have at least one grid point.

(a) Grid address points of Baltimore

(b) Grid address points (zoomed in)

Figure 5-4: GTFS process flow and result

After allocating grid points, we overlap them over land use divisions, assign a land use category to each grid point and calculate the number of grid points in each land use category of a specific county of Baltimore.

5.2.3 Residential population and employment allocation

From the previous section, I find the number of grid points in each land use category. I then distribute the total population of Baltimore over each of those grid points given their category using Formula 5.1, assuming that each grid point on a particular category has the same number of people. This gives a particular number of people

Landuse	**Anne Arundel**	**Baltimore county**	**Carroll**	**Harford**	**Howard**	**Queens Anne**	**Baltimore city**
water	4817	2402	168	2447	43	3896	301
openLand	104	116	174	341	70	122	26
indust	185	453	60	89	216	9	374
highR	364	755	58	180	210	11	621
forest	4146	4845	3027	3709	1823	2659	158
lowR	1756	2170	1823	1637	1245	479	24
commer	499	509	180	235	171	59	175
edu	437	382	134	720	147	59	225
mediumR	1488	1743	307	545	705	173	406
urbanOL	251	394	82	164	128	60	155
sparseR	542	1195	1044	931	585	381	0
trans	164	144	8	50	106	32	82
agricul	1392	3747	5532	3546	1533	6103	0
total	16145	18855	12597	14594	6982	14043	2547

Table 5.5: County level statistics

living at a address point of each category. We define:

N_{total}^{HH} - total population

M_{total}^{EM} - total number of employment

LU - land use categories

N_j - number of address points with j^{th} land use category

n_j - number of population at a point address with j^{th} land use category

m_j - number of employement at a point address with j^{th} land use category

$$\begin{aligned} \text{solve} \quad & \sum_{j \in LU} n_j * N_j = N_{total}^{HH} \\ \text{subject to} \quad & 0 \leq n_A \leq 10 \\ & 0 \leq n_C \leq 10 \\ & 240 \leq n_{HR} \leq 600 \\ & 60 \leq n_{MR} \leq 240 \\ & 6 \leq n_{LR} \leq 60 \\ & 0 \leq n_{SR} \leq 6 \end{aligned} \tag{5.1}$$

Employment allocation is done using a similar method with Formula 5.2.

$$\begin{aligned} \text{solve} \quad & \sum_{j \in LU} m_j * N_j = M_{total}^{EM} \\ \text{subject to} \quad & 0 \leq m_A \leq 600 \\ & 0 \leq m_C \leq 600 \\ & 0 \leq m_{HR} \leq 600 \\ & 0 \leq m_{MR} \leq 600 \\ & 6 \leq m_{LR} \leq 600 \\ & 0 \leq m_{SR} \leq 600 \end{aligned} \tag{5.2}$$

I solve these equations using linear programming and obtain the resulting weights for counties in Tables 5.6 and 5.7.

Land use	**24003**	**24005**	**24013**	**24025**	**24027**	**24035**	**24510**
A	10	0	9.42607	10	10	0	0
C	10	0	0	10	10	0	10
HR	293.552	240	240	368.044	526.271	240	600
LR	60	6	6	6	6	29.1608	60
MR	60	183.431	60	60	60	60	201.835
SR	6	0	0	0	0	0	0
N_{total}^{HH}	323655	513941	95423	146580	177327	26988	457735
$Found\ N_{total}^{HH}$	323654.928	513940.233	95423.01924	146579.92	177326.91	26988.0232	457735.01
Error	-0.072	-0.767	0.01924	-0.08	-0.09	0.0232	0.01

Table 5.6: Population weights

Land use	**24003**	**24005**	**24013**	**24025**	**24027**	**24035**	**24510**
A	161.496	0	7.90365	17.2239	99.5492	0	0
C	60	60	60	60	60	60	393.274
HR	10	10	10	10	10	10	10
MR	0	187.831	0	0	0	0	600
LR	0	0	0	0	0	20.7891	600
SR	0	0	0	0	0	0	0
I	0	0	0	0	0	0	0
E	20	20	20	20	20	20	20
M_{total}^{EM}	267123	373120	57783	91376	167909	14788	337533
$Found\ M_{total}^{EM}$	267122.432	373119.433	57782.9918	91375.9494	167908.9236	14787.9789	337532.95
Error	-0.568	-0.567	-0.0082	-0.0506	-0.0764	-0.0211	-0.05

Table 5.7: Employment weights

5.2.4 Education allocation

I collected address of all educational institutions in Baltimore metropolitan with number of enrollment and type of institution from Maryland's Mapping and GIS Data Portal [16]. Since educational institution areas and enrollments can be huge, I allocate each institution's enrollment to a neighborhood of radius R (R = 2 km square for schools, 4 km for universities) around point address of each institution, generating a map with all educational institutions encircled with their radii.

I take the grid point addresses and overlap with the encircled area of the educational institutions and note only the number of grid points in education land use category for each educational institution, as shown in Figure 5-5. I then divide the total number of enrollment equally into the grid points for each institution.

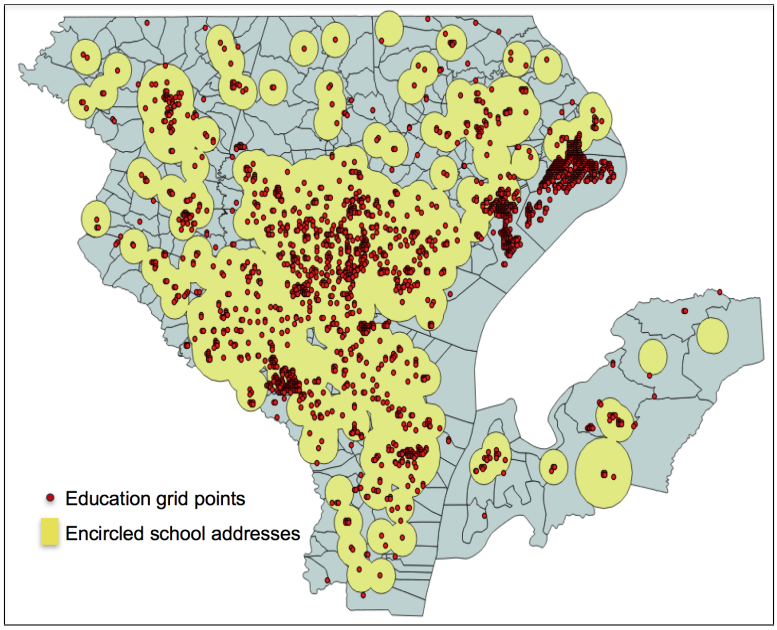


Figure 5-5: Education Establishments, Baltimore Metro Area

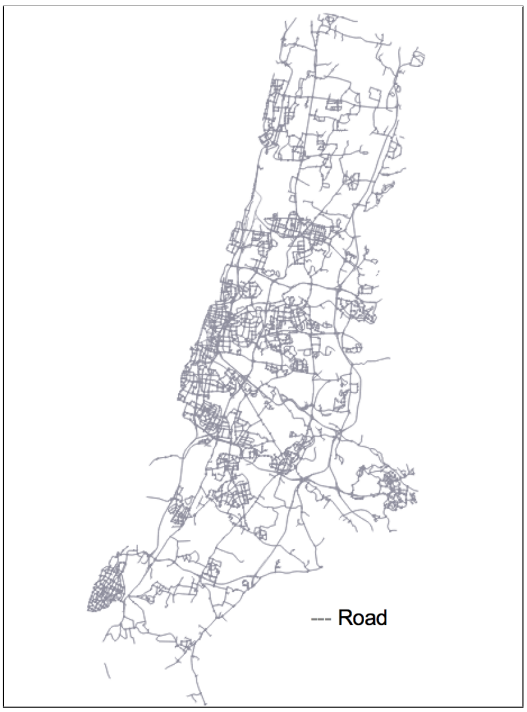
THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

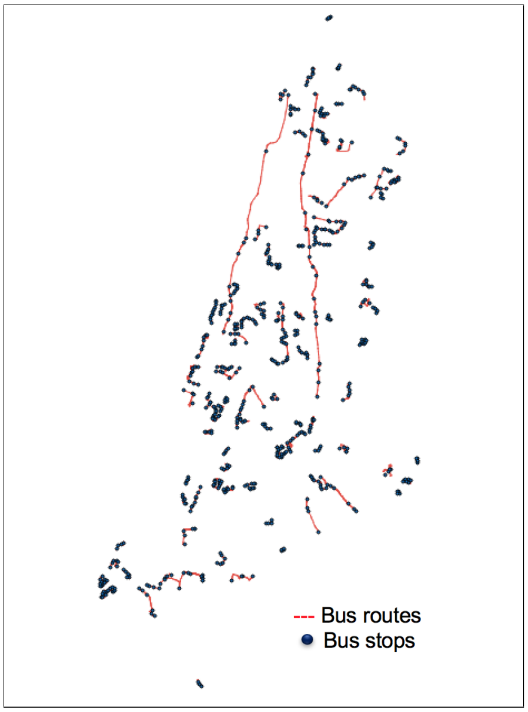
Results and Ongoing Development

I deployed ubiquitously available OSM, population microdata samples and aggregated totals, educational institution locations and land use data to create a prototype city generation pipeline. I used this pipeline to generate prototype city for Auto-sprawl typology (i.e. Baltimore metropolitan) and tested with the real data. I found that the Auto-sprawl prototype city fit the real city population under control aggregated data.

My pipeline is being applied to generate prototype cities for other urban typologies such as Sustainable Anchor (i.e. Tel Aviv, Isreal). Its road network, bus lanes and socio-demographic population and its allocation are created, as shown in Table 6-1 and 6-2.

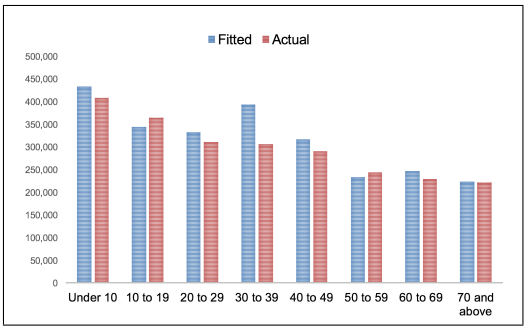


(a) Generated road network

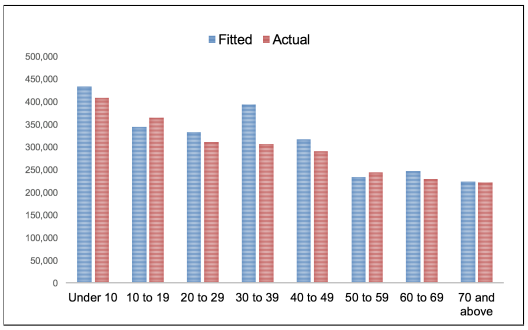


(b) Processed bus lanes

Figure 6-1: Synthesized Sustainable Anchor typology (i.e. Tel Aviv, Isreal)



(a) Age distributionl



(b) Gender distribution

Figure 6-2: Sustainable Anchor typology (i.e. Tel Aviv, Isreal)

Auto-sprawl and Sustainable Anchor basic statistics are listed in Table 6.1. Furthermore, synthesized Auto-sprawl has been fed to SimMobility simulation platform and tested for basic statistics. For example, in SimMobility, individuals' daily activities schedules (DAS), which is a sequence of activities and DAS of Auto-sprawl has been found as shown in Table 6.2. Simulation tests on Auto-sprawl are still ongoing.

Attributes	**Auto-sprawl**	**Sustainable Anchor**
Representative city	Baltimore metropolitan	Tel Aviv metropolitan
Total population	2779819	2523900
Road network intersections (node)	11431	15746
Road network (link)	24133	32424
Number of bus stops	870	670
Number of bus routes	150	211
Number of train lines	10	9

Table 6.1: Synthesized prototype cities (ongoing)

Travel mode	**Number of trips**
walk	649351
taxi	60610
private car	7395654
car sharing	2330280
private bus	952578
public bus	37834
train	199894

Table 6.2: Daily Activity Schedule Trips, Auto-sprawl

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Conclusion

The contribution of this project lies in the following three aspects: First, I created an automated road network query and feature synthesis, and developed population synthesis and allocation framework using existing methods. Second, I synthesized Auto sprawl urban typology prototype and provided an exemplary workflow. Third, I made my documented workflow framework available to my group which could be modified and extended in future and applied to other urban typologies.

These tools and pipeline rely only on open data. As such, I use high resolution land use data that is publicly available. But, this kind of land use data may not be available for some cities. However, this issue can be tackled by replacing land use data with OpenStreetMap land use or road network density inference.

The created prototype city generation framework is generalizable and tested with an example. This framework will facilitate an integrated generation process for simulatable urban typology or cities for agent-based studies using SimMobility.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Tables

A.1 SimMobility bus tables

column	data type	explanation
frequency_id	string	Identify bus line
route_id	string	Identify bus line
start_time	string	Define starting time, format like: HH:MM:SS
end_time	string	Define ending time, format like: HH:MM:SS
headway_sec	integer	Define how long time to dispatch next bus. The unit is second

Table A.1: SimMobility pt_bus_dispatch_freq table

column	data type	explanation
route_id	string	Identify bus line
section_id	integer	Identify road segment on the map
sequence_no	integer	The index in the list

Table A.2: SimMobility pt_bus_routes table

column	data type	explanation
route_id	string	Identify bus line
stop_code	string	Identify bus stop
sequence_no	integer	The index in the list

Table A.3: SimMobility pt_bus_stops table

A.2 SimMobility train tables

column	data type	explanation
shape_id	integer	train stop id
x	double	x coordinate
y	double	y coordinate
z	double	z coordinate (currently we do not have this data)
platform_name	string	the name of platform, eg: EW24/NS1, EW23
station_name	string	the station name, one station may include multiple platforms, eg: Jurong East, Clementi
type	string	the type of train service, eg: MRT, LRT
op_year	string	operating year

Table A.4: SimMobility train_stop table

column	data type	explanation
mrt_stop_id	integer	train stop id
segment_id	integer	segment id

Table A.5: SimMobility train_access_segment table

column	data type	explanation
platform_no	string	platform id, eg: EW23_1, EW23_2
station_no	string	train stop, eg: EW23
line_id	string	train service line, eg: EW_1 (East West Line towards Pasir Ris), EW_2 (East West Line towards Tuas)
capacity	integer	capacity of train, eg: MRT(1920 persons), LRT(105 persons)
type	integer	terminal - 1 ; not terminal - 0
block_id	integer	the block id of the platform
pos_offset	double	default value: 0
length	double	the length of the platform, in meters. Default value: MRT(142m), LRT(56m)

Table A.6: SimMobility train_platform table

column	data type	explanation
block_id	integer	the train block id
default_speed_limit	double	default value: 70
accelerate_rate	double	default value: 1.1
decelerate_rate	double	default value: 1.1
length	double	the length of train block, in meters

Table A.7: SimMobility pt_train_block table

column	data type	explanation
polyline_id	integer	the train block id
x	double	x coordinate
y	double	y coordinate
z	double	z coordinate (currently we do not have this data)
sequence_no	integer	order of point in the train block

Table A.8: SimMobility pt_train_block_polyline table

column	data type	explanation
frequency_id	string	frequency id
line_id	string	train service line id, eg: EW_1, EW_2
start_time	string	time of first train
end_time	string	time of last train
headway_sec	integer	time between trains, in seconds

Table A.9: SimMobility pt_train_dispatch_freq table

column	data type	explanation
station_no	string	train stop id, eg: EW23
platform_first	string	from platfrom, eg: EW23_1
platform_second	string	to platfrom, eg: EW23_2
transfered_time_sec	integer	the transfer time between platforms, in seconds. Default value: In the same train stop/station: (Different platfrom, same train line service - 0 second); (Different platform, different train line service - 60 seconds)

Table A.10: SimMobility pt_train_platform_transfer_time table

column	data type	explanation
line_id	string	train service line id
platform_no	integer	train platform id
sequence_no	integer	order of platform of train routes

Table A.11: SimMobility pt_train_route_platform table

column	data type	explanation
line_id	string	train service line id
block_id	integer	train block id
sequence_no	integer	order of block of train routes

Table A.12: SimMobility pt_train_route table

column	data type	explanation
line	string	train service line id
opp_line	string	the opposite travel direction train service line id

Table A.13: SimMobility pt_opposite_lines table

A.3 SimMobility road network tables

column	description
id	segment id
link_id	every segment is part of the link
sequence_num	sequence number of segment in the link
num_lanes	number of lanes
capacity	capacity of the segment (vehicle per hour per lane * num_lanes)
max_speed	speed limit of the segment (km/hr)
tags	to store any additional info
link_category	category of the link foreign key to the link category table

Table A.14: SimMobility segment table

column	description
id	segment id
x	x position
y	y position
z	z position (currently we do not have any data for z position)
seq_id	order of the coordinate in the segment

Table A.15: SimMobility segment_polyline table

column	description
id	node id
x	x position
y	y position
z	z position (currently we do not have this data)
traffic_light_id	traffic light id from the traffic data
tags	if any information we need to store
node_type	0 is default node, 1 is SINK or SOURCE node, 2 is INTERSECTION NODE, 3 is MERGING/DIVERGING NODE

Table A.16: SimMobility node table

column	description
id	link id
road_type	Road type EXPRESSWAY = 1, Road type URBAN = 2, Road type SLIPROAD = 3, Road type ROUNDABOUT = 4, Road type ACCESS = 5
category	Road type EXPRESSWAY(Linkcat A) = 1, Road type URBAN(Linkcat B) = 2, Road type URBAN(Linkcat C) = 3, Road type ACCESS(Linkcat D) = 4, Road type ACCESS(Linkcat E) = 5, Road type SLIPROAD (Linkcat SLIPROAD) = 6, Road type ROUNDABOUT(Linkcat ROUNDABOUT) = 7
from_node	Start node of the link
to_node	end node of the link
road_name	Name of the road

Table A.17: SimMobility link table

column	description
id	link id
x	x position
y	y position
z	z position (currently we do not have any data for z position)
seq_id	order of the coordinate in the segment

Table A.18: SimMobility link_polyline table

column	description
id	lane id
width	width of the lane by default is 3.5 m
vehicle_mode	currently not used
bus_lane	whether a bus lane?
can_stop	whether vehicle can stop in this lane
can_park	whether vehicle can park in this lane
high_occ_veh	whether high occ vehicle is allowed in this lane
has_road_shoulder	whether has road shoulder
segment_id	segment id the lane belongs to

Table A.19: SimMobility lane table

column	description
id	link id
x	x position
y	y position
z	z position (currently we do not have any data for z position)
seq_id	order of the coordinate in the segment

Table A.20: SimMobility lane_polyline table

column	description
id	connector id
from_segment	starting segment for this connection
to_segment	end segment for this connection
from_lane	starting lane
to_lane	ending lane
is_true_connector	indicates whether the from and to lanes are physically connected. 0 - not physically connected, 1 - physically connected

Table A.21: SimMobility connector table

column	description
id	turning path id
from_lane	id of the incoming lane to the turning path
to_lane	id of the outgoing lane of the turning path
group_id	id of the turning group to which the turning path belongs
max_speed	speed limit in the turning (km/hr)

Table A.22: SimMobility turning_path table

column	description
id	
node_id	Intersection node
from_link	id of the incoming link to the turning
to_link	id of the outgoing link of the turning
phases	traffic light phase for this intersection
rules	NO_STOP_SIGN = 0, STOP_SIGN = 1 (currently we do not have any data)
visibility	distance from where this intersection is visible

Table A.23: SimMobility turning_group table

column	description
id	turning path id
x	x positon
y	y position
z	z position (we do not have data yet)

Table A.24: SimMobility turning_path_polyline table

A.4 Miscellaneous tables

Table A.25: Maryland Department of Planning 2010 Land Use/Land Cover Classification Definitions [12]

LU code	Proposed categories	Cover Classification Definition
11	low_residential	**Low-density residential** - Detached single-family/duplex dwelling units, yards and associated areas. Areas of more than 90 percent single-family/duplex dwelling units, with lot sizes of less than five acres but at least one-half acre (.2 dwelling units/acre to 2 dwelling units/acre).
12	medium_residential	**Medium-density residential** - Detached single-family/duplex, attached single-unit row housing, yards, and associated areas. Areas of more than 90 percent single-family/duplex units and attached single-unit row housing, with lot sizes of less than one-half acre but at least one-eighth acre (2 dwelling units/acre to 8 dwelling units/acre).
13	hight_residential	**High-density residential** - Attached single-unit row housing, garden apartments, high-rise apartments/condominiums, mobile home and trailer parks; areas of more than 90 percent high-density residential units, with more than 8 dwelling units per acre. subsidized housing
14	commercial	**Commercial** - Retail and wholesale services. Areas used primarily for the sale of products and services, including associated yards and parking areas. This category includes: Airports, Welcome houses, Telecommunication towers and Boat Marinas.
15	industrial	Industrial - Manufacturing and industrial parks, including associated warehouses, storage yards, research laboratories, and parking areas. Warehouses that are returned by a commercial query should be categorized as industrial. Also included are power plants.

16	education	**Institutional** - Elementary and secondary schools, middle schools, junior and senior high schools, public and private colleges and universities, military installations (built-up areas only, including buildings and storage, training, and similar areas), churches, medical and health facilities, correctional facilities, and government offices and facilities that are clearly separable from the surrounding land cover. This category includes: campgrounds owned by groups/community groups (ie girl scouts) and sports venues.
17	agriculture	**Extractive** - Surface mining operations, including sand and gravel pits, quarries, coal surface mines, and deep coal mines. Status of activity (active vs. abandoned) is not distinguished.
18	open_land	**Open urban land** - Urban areas whose use does not require structures, or urban areas where non-conforming uses characterized by open land have become isolated. Included are golf courses, parks, recreation areas (except areas associated with schools or other institutions), cemeteries, and entrapped agricultural and undeveloped land within urban areas. When addressing parks, buildings are classified as 18 and ground cover is classified according to imagery.
191	sparse_residential	**Large lot subdivision (agriculture)** - Residential subdivisions with lot sizes of less than 20 acres but at least 5 acres, with a dominant land cover of open fields or pasture.
192	sparse_residential	**Large lot subdivision (forest)** - Residential subdivisions with lot sizes of less than 20 acres but at least 5 acres, with a dominant land cover of deciduous, evergreen or mixed forest.
21	agriculture	**Cropland** - Field crops and forage crops.
22	agriculture	**Pasture** - Land used for pasture, both permanent and rotated; grass.

23	agriculture	**Orchards/vineyards/horticulture** - Areas of intensively managed commercial bush and tree crops, including areas used for fruit production, vineyards, sod and seed farms, nurseries, and green houses.
24	agriculture	**Feeding operations** - Cattle feed lots, holding lots for animals, hog feeding lots, poultry houses, and commercial fishing areas (including oyster beds).
241	agriculture	**Feeding operations** - Cattle feed lots, holding lots for animals, hog feeding lots, poultry houses.
242	agriculture	**Agricultural** building breeding and training facilities, storage facilities, built-up areas associated with a farmstead, small farm ponds, and commercial fishing areas.
25	agriculture	**Row and garden crops** - Intensively managed truck and vegetable farms and associated areas.
41	forest	**Deciduous forest** - Forested areas in which the trees characteristically lose their leaves at the end of the growing season. Included are such species as oak, hickory, aspen, sycamore, birch, yellow poplar, elm, maple, and cypress. Note that forest classifications may not be reliable as to type (decidiouous versus evergreen).
42	forest	**Evergreen forest** - Forested areas in which the trees are characterized by persistent foliage throughout the year. Included are such species as white pine, pond pine, hemlock, southern white cedar, and red pine. Note that forest classifications may not be reliable as to type (decidiouous versus evergreen).
43	forest	**Mixed forest** - Forested areas in which neither deciduous nor evergreen species dominate, but in which there is a combination of both types.

44	forest	**Brush** - Areas which do not produce timber or other wood products but may have cut-over timber stands, abandoned agriculture fields, or pasture. These areas are characterized by vegetation types such as sumac, vines, rose, brambles, and tree seedlings.
50	water	**Water** - Rivers, waterways, reservoirs, ponds, bays, estuaries, and ocean.
60	water	**Wetlands** - Forested or non-forested wetlands, including tidal flats, tidal and non-tidal marshes, and upland swamps and wet areas.
70	open_land	**Barren land**
71	open_land	**Beaches** - Extensive shoreline areas of sand and gravel accumulation, with no vegetative cover or other land use.
72	open_land	**Bare exposed rock** - Areas of bedrock exposure, scarps, and other natural accumulations of rock without vegetative cover.
73	open_land	**Bare ground** - Areas of exposed ground caused naturally, by construction, or by other cultural processes. Landfills (cultural process) are included in this category
80	transportation	**Transportation** - Transportation features include major highways, light rail or metro stations and large "Park N Ride" lots, generally over ten acres in size. Major highways were defined a those appearing on the State Highway maps as Controlled Access Highways or Primary Highways

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

Scripts

B.1 Population Synthesis

Listing B.1: MultiLevelIPF R library

```
library(MultiLevelIPF)
library(stringr)

samples <- read.table("samples.dat", header=TRUE)

age <- read.table("totals/age.dat", header=TRUE)
gender <- read.table("totals/gender.dat", header=TRUE)
edu <- read.table("totals/edu.dat", header=TRUE)

household_income <- read.table("totals/household_income.dat", header=TRUE)
vehicles <- read.table("totals/vehicles.dat", header=TRUE)

fit_problem <- fitting_problem(ref_sample = samples,
           field_names = special_field_names(groupId = "household_id", individualId↩
                = "individual_id", count = "count"),
           group_controls = list(household_income, vehicles),
           individual_controls = list(age, gender, edu))

result <- ml_fit_ipu(fit_problem)
write.csv(result["weights"], "multilevel_sample_weights.csv", append=False, sep = "↩
    ")
```

Listing B.2: Duplicate households with their individuals based on household weights

```
import pandas as pd
import math
import numpy as np

def create_county_population(sampleFile, total_population, weightFile):
    weights = pd.read_csv(weightFile)
    samples = pd.read_csv(sampleFile)

    w_scaling = total_population / weights['weights'].sum()
    samples['weight'] = weights['weights'] * w_scaling

    ############################ Integerize households ############################
    unique_hh = samples.drop_duplicates('hhid', inplace=False)
    counts = samples['hhid'].value_counts()

    error_weight = 0
    num_in_population = {}
    syn_pop = 0

    SMALL_HHID = {}
    for index, row in unique_hh.iterrows():
        if error_weight > 0:
            int_num = math.floor(row.weight)
        else:
            int_num = math.ceil(row.weight)
        error_weight += (int_num - row.weight) * row.APER # ind weight
        syn_pop +=  int_num * row.APER
        num_in_population[row.hhid] = int_num
        SMALL_HHID[row.hhid] = index
    print('synthetic total population', syn_pop, ' number of households', sum(↩
        num_in_population.values())) # (unique_hh['weight']*unique_hh['APER']).sum↩
        ())

    ######################### Write new population ##############################
    max_num = max(num_in_population.values())
    scale = 10**(math.ceil(math.log(max_num, 10)))
    hhsample = 0

    new_population = []
    new_columns = ['hhid', 'indid', 'APER', 'gender' , 'age', 'educ', 'vehicles', '↩
        income', 'school', 'employment']

    for index, sample in samples.iterrows():
        for i in range(num_in_population[sample.hhid]):
```

```
            # hhid = sample.hhid * scale + i # using old hhid
            hhid = SMALL_HHID[sample.hhid] * scale + i
            indid = hhid * 100 + (sample.indid % 100)
            new_population.append((int(hhid), int(indid),
                                   int(sample.APER), int(sample.gender),
                                   int(sample.age), int(sample.educ),
                                   int(sample.vehicles), int(sample.income_earn),
                                   int(sample.school), int(sample.employment)))

    synthetic_population = pd.DataFrame.from_records(new_population, columns=↩
        new_columns)
    synthetic_population.to_csv('synthesized_population.csv', index=False)
```

B.2 Road Network Generation

Listing B.3: Interface for Generating Road Network

```
##############################################################################
# Module: simplify.py
# Description: Query OpenStreetMap and prepare SimMobility road network files.
# Written by: Iveel Tsogsuren, Arun Akkinepally
##############################################################################
from collections import OrderedDict
import process_osm as posm
import osmnx as ox
import query_osm as qr
import networkx as nx
import os
from network import*
# from processing import*

LAT_LONG_CRS = {'init': 'epsg:4326'}
TEL_AVIV_CRS = {'init': 'epsg:2039'}

# Get default SimMobility attribute values
inputFolder = "metadata/"
typeToWidthFname = os.path.join(inputFolder,"LinkCat_Roadtype_LaneWidth.csv")
ffsFname = os.path.join(inputFolder,"HCM2000.csv")

# Set this directory where you optionally pass boundary.shp file and get all ↩
    outputs.
directory = '/Outputs/Tel_Aviv'
directory = '/Outputs/Tel_Aviv_drive_all'
directory = '/Outputs/Tel_Aviv_drive_main'
# boundary = os.getcwd() + '/Outputs/Tel_Aviv/Tel_Aviv_Metro_Area/buffer_wgs84.shp'↩
     # Can be either bounding box coordinates or
directory = '/Outputs/Test'
boundary = [42.3645,42.3635,-71.1046,-71.108]



# Prepare subfolders for outputs
outDir = os.getcwd() + directory
simmobility_crs_projected = outDir + "/simmobility_crs_projected"  # For later use.
simmobility_dir = outDir + "/simmobility_wgs84"
shapefile_dir = outDir + "/shapefiles"
sumo_dir = outDir + "/sumo"
```

```
graph_pickle_file = outDir + '/osm_graph.pkl'
for d in [outDir, simmobility_dir, simmobility_crs_projected, shapefile_dir, ↩
    sumo_dir]:
    if not os.path.exists(d):
        print(d)
        os.makedirs(d)

def query_OSM(directory, boundary, graph_pickle_file, query='drive_main'):
    """
    Query OpenStreetMap and save it as a pickle file.
    Parameters
    ----------
    directory : directory path where inputs and outputs should be
    boundary : Boundary for road network which can be either polygon boundary file
               or bounding box coordinates [north, south, west, east]
    network_type : string
        {'walk', 'bike', 'drive_all', 'drive_main', 'drive_main_links_included', '↩
            drive_service', 'all', 'all_private', 'none'}
        what type of street or other network to get
    """
    if type(boundary) == str: # boundary .shp file path
        polygon = gpd.read_file(boundary)
        polygon = polygon.get_value(0, 'geometry')
        mainG = ox.graph_from_polygon(polygon, network_type=query,  simplify=False,↩
             retain_all=True, truncate_by_edge=False)
    else: # bounding box []
        mainG = qr.graph_from_bbox(*boundary)
    nx.write_gpickle(mainG, graph_pickle_file)

def main(graph_pickle_file=graph_pickle_file, simmobility_dir=simmobility_dir):
    """
    Create all SimMobility road network tables and their shapes given a OSM graph ↩
        file.
    Parameters
    ----------
    graph_pickle_file : file path of OSM graph in pickle format.
    simmobility_dir : output directory
    """
    query_OSM(directory, boundary, graph_pickle_file)
    # all SimMobility road network module files.
    mainG = nx.read_gpickle(graph_pickle_file)
    roadnetwork = Network(mainG)
    roadnetwork.process_segments_links_nodes(clean_intersections=False)
    roadnetwork.lanes = posm.constructLanes(roadnetwork.segments, typeToWidthFname)
    roadnetwork.constructSegmentConnections()
```

```
    roadnetwork.construct_default_turning_path()

    # Files for creating smart turnning path connections using SUMO. TODO: use SUMO↩
        .
    # roadnetwork.writeSumoShapefile(sumo_dir)
    # roadnetwork.construct_turning_paths_from_SUMO(sumo_dir)

    roadnetwork.write_wgs84(foldername=simmobility_dir+inputFolder.split('/')[1])
    roadnetwork.writeShapeFiles(foldername=shapefile_dir+inputFolder.split('/')[1])
```

B.3 Public Transit GTFS Process

Listing B.4: Bus GTFS Process

```
import pandas as pd
import geopandas as gpd
from shapely.ops import nearest_points
from shapely.geometry import Point
from shapely.geometry import MultiPoint
import math
from collections import defaultdict
from process_helper2 import *


# Use consistent xy coordinate GTFS. Adjust right CRS conversions.
LAT_LONG_CRS = {'init': 'epsg:4326'}
BALTIMORE_CRS = {'init': 'epsg:6487'}
TELAVIV_CRS = {'init': 'epsg:2039'}


# Baltimore
simFolder = 'Auto_sprawl_drive_main/simmobility/'
gtfsFolder = 'clean-gtfs-baltimore/MergedBus/'
processFolder = 'process_baltimore/'
CURRENT_CRS =  BALTIMORE_CRS

# Tel Aviv
# simFolder = '../../network-from-OSM/Outputs/tel_aviv/simmobility_wgs84/'
# gtfsFolder = 'gtfs_clean_israel/bus/'
# processFolder = 'process_telaviv/'
# CURRENT_CRS =  TELAVIV_CRS

# # Small example
# simFolder = 'Baltimore_small/simmobility/'
# gtfsFolder = 'gtfs_source_small_example/gtfs-QueenAnne/'
# processFolder = 'process_small_example/'
# CURRENT_CRS =  BALTIMORE_CRS


def findCandidateSegments(bufferSize=400):
    gtfs_trips_df, gtfs_stoptime_df, gtfs_shape_df, gtfs_stop_df = readGTFS(↩
        gtfsFolder,
```

```
        trip='trips.txt', stoptime='stop_times.txt', shape='shapes.txt', stop='↩
            stops.txt')
    segmentGeo = getRoadNetworkGeos(simFolder, fromCRS=LAT_LONG_CRS, toCRS=↩
        CURRENT_CRS)
    segmentGeo.to_file(processFolder + 'SegmentGeo')
    busStops, busShapes = getBusRouteGeo(gtfs_stop_df, gtfs_shape_df, fromCRS=↩
        LAT_LONG_CRS, toCRS=CURRENT_CRS)
    busStops.to_file(processFolder + 'busStops')
    busShapes.to_file(processFolder + 'busShapes')
    candidateShapeSegments = findRouteCandidateSegments(segmentGeo, busShapes, ↩
        processFolder, bufferSize=bufferSize)
    candidateShapeSegments.to_file(processFolder + 'candidateSegments')

# Step 1: Convert segments into graph with end vertices.
# SimMobility segments do not have start and end vertices (nodes).
def getSegment(simFolder=simFolder):
    # Simmobility segment-nodes in lat and long
    segmentGeo = gpd.read_file(processFolder + 'candidateSegments/candidateSegments↩
        .shp')
    segmentsWithNodes = getSegmentsWithEndNodes(simFolder, segmentGeo, CURRENT_CRS)
    # clean out the same direction (from node, to node)
    # There could be segments whose ends are the same but have different shapes.
    # For understanding, you can comment out the following code and check its ↩
        shapefile.
    # segmentsWithNodes_duplicated = segmentsWithNodes[segmentsWithNodes.duplicated↩
        (['from_node', 'to_node'], keep=False)]
    # print('segmentsWithNodes duplicated (from node, to node) ', len(↩
        segmentsWithNodes_duplicated.id.unique()))
    # segmentsWithNodes_duplicated.to_file(processFolder + '↩
        segmentsWithNodes_duplicated')
    segmentsWithNodes.drop_duplicates(subset=['ends'], inplace=True)
    segmentsWithNodes.to_file(processFolder + 'segmentsWithNodes')


# Step 3: Find a start point for each segment. For each bus stop, we find
# the nearest start point because it is computationally faster.
def getSegment_startEndPoint():
    segmentsWithNodes = gpd.read_file(processFolder + 'segmentsWithNodes/↩
        segmentsWithNodes.shp')
    segmentsWithNodes['start_point'] =  segmentsWithNodes.apply(lambda row: Point(↩
        list(row.geometry.coords)[0]), axis=1)
    segmentEnds = gpd.GeoDataFrame(segmentsWithNodes[['id', 'from_node', 'to_node'↩
        ]], crs=CURRENT_CRS, geometry=segmentsWithNodes['start_point'])
    segmentEnds.to_file(processFolder + 'segmentStartPoint')
    # Present segments by their start point
```

```
# Step 4: Find the nearest segment end for each bus stop. (QGIS NNjoin tool)
    # input files: busStops and segmentStartPoint (prefex S)
    # output: stop_to_segmentEnd or busStops_segmentStartPoint
    # all in processFolder.


# Step 5: Find a representive segment start for each stop.
# Later we assign a segment based on bus trip connection.
def stop_to_Segment(maxDistance=400):
    # stopSegment = gpd.read_file(processFolder + 'busStops/↩
        busStops_segmentStartPoint.shp')
    stopSegment = gpd.read_file(processFolder + 'busStops/stop_to_segmentStart.shp'↩
        )

    stopSegment = stopSegment[['stop_id', 'Sid', 'Sfrom_node', 'Sto_node', '↩
        distance']]
    stopSegment['distance'] = stopSegment['distance'].astype('float')
    stopSegment = stopSegment[stopSegment['distance'] < maxDistance]

    # Keep the nearest segment
    stopSegment.sort_values(by='distance', inplace=True)
    stopSegment = stopSegment.drop_duplicates(subset=['stop_id'], keep='first')
    stopSegment.rename(columns={'Sid':'segment_id', 'Sfrom_node':'from_node', '↩
        Sto_node':'to_node'}, inplace=True)
    stopSegment['str_end_nodes'] = stopSegment.apply(lambda row: str(int(row.↩
        from_node)) + '_' + str(int(row.to_node)), axis=1)
    stopSegment.to_csv(processFolder + 'cleaned_stop_segmentEnd.csv')
    print(stopSegment.columns)


# Step 6: Find a unique trips which we will construct.
def mainUniqueBusRoutes():
    gtfs_trips_df, gtfs_stoptime_df, gtfs_shape_df, gtfs_stop_df = readGTFS(↩
        gtfsFolder,
        trip='trips.txt', stoptime='stop_times.txt', shape='shapes.txt', stop='↩
            stops.txt')
    trip_stops = getUniqueBusRoutes(gtfs_stoptime_df, gtfs_trips_df)
    trip_stops.to_pickle(processFolder + 'trip_stop_sequence.pkl')


# Step 7: Find connected subsequent bus stops
def getConnectedConsequentStops():
    trip_stops = pd.read_pickle(processFolder + 'trip_stop_sequence.pkl')
```

```
    candidateShapeSegments = gpd.read_file(processFolder + 'segmentsWithNodes/↩
        segmentsWithNodes.shp')
    stopSegmentEnd = pd.read_csv(processFolder + 'cleaned_stop_segmentEnd.csv')
    stopConnection_df = getShortestPath(trip_stops, candidateShapeSegments, ↩
        stopSegmentEnd)
    stopConnection_df.to_pickle(processFolder + 'connectedStops.pkl')


# Step 8: Construct connected trips
def processConnectedStops():
    stopConnection = pd.read_pickle(processFolder + 'connectedStops.pkl')
    stopConnection.index = stopConnection.apply(lambda row: str(int(row.startStop))↩
         + '_' + str(int(row.endStop)), axis=1)
    connectedStops = stopConnection.index.tolist()
    trip_stops = pd.read_pickle(processFolder + 'trip_stop_sequence.pkl')
    subtrips = []
    stop_to_segment = {}
    for indx, row in trip_stops.iterrows():
        disconnected=False
        conseqStops = list(zip(row.stop_id[:-1], row.stop_id[1:]))
        stop_segmentEnds = []
        path_segmentEnds = []
        path_stops = []
        connected=True
        for s1, s2 in conseqStops:
            string_code = str(s1) + '_' + str(s2)
            if string_code in connectedStops:
                node_sequence =  stopConnection.loc[string_code, 'nodePath']
                if len(node_sequence) > 3: # length must be at least 3
                    stopSeg1 = (node_sequence[0], node_sequence[1])
                    stopSeg2 = (node_sequence[-2], node_sequence[-1])
                    if len(path_stops) == 0:
                        path_stops = [s1,s2]
                        path_segmentEnds += node_sequence
                        stop_segmentEnds += [stopSeg1, stopSeg2]
                    # check if (... from1 -->to) + (from2-->D); fact (from2-->to) ↩
                        exists
                    elif path_segmentEnds[-2] == node_sequence[0]:
                        path_stops.append(s2) # s1 already appended  (... A->B) + (↩
                            A->C ...) ==> ... A --> C ...
                        path_segmentEnds = path_segmentEnds[:-1] + node_sequence↩
                            [1:]
                        stop_segmentEnds.append(stopSeg2) # leave the very first ↩
                            node
```

```
                    # Untwist # check if (... from1 -->to) + (from2-->D); fact (↩
                        from2-->to) exists
                    # TODO: check if to --> from2, then simply add to --> from2
                    else:
                        disconnected=True
                else:
                    disconnected=True
            else:
                disconnected=True

            if disconnected:
                if len(path_stops) > 0: # Not connect; start a new trip .
                    subtrips.append((row.trip_id, row.shape_id, path_stops, ↩
                        path_segmentEnds, stop_segmentEnds))
                path_stops = []
                path_segmentEnds = []
                stop_segmentEnds = []
                disconnected = False

    subtrips = pd.DataFrame.from_records(subtrips, columns=['trip_id', 'shape_id', ↩
        'stops', 'path_segmentEnds', 'stop_segmentEnds'])
    subtrips['number_stops'] = subtrips.apply(lambda row: len(row.stops), axis=1)
    print('Number of subtrips ', len(subtrips))
    print('Max number of stops ', subtrips.number_stops.max())
    print('Avg number of stops ', subtrips.number_stops.mean())
    subtrips.to_pickle(processFolder + 'subtrips_found.pkl')


# Step 9: Express trips in segments as SimMobility requires.
def segmentizeTrips():
    subtrips = pd.read_pickle(processFolder + 'subtrips_found.pkl')
    segmentsWithNodes = gpd.read_file(processFolder + 'segmentsWithNodes/↩
        segmentsWithNodes.shp')
    segmentsWithNodes.drop_duplicates(subset=['ends'], inplace=True)
    segmentsWithNodes.to_csv(processFolder + 'candidateShapeSegments.csv')
    segmentsWithNodes.index = segmentsWithNodes.ends
    def getSegment(ends, row, segmentsWithNodes=segmentsWithNodes):
        ends_string = str(int(ends[0])) + '_' + str(int(ends[1]))
        return segmentsWithNodes.loc[ends_string, 'id']
    subtrips['path_segmentEnds'] = subtrips.apply(lambda row: list(zip(row.↩
        path_segmentEnds[:-1], row.path_segmentEnds[1:])), axis=1)
    # print(subtrips['path_segmentEnds'])
    def segmentize(segmentEnds):
        segments = []
        for ends in segmentEnds:
```

```
            ends_string = str(int(ends[0])) + '_' + str(int(ends[1]))
            s = segmentsWithNodes.loc[ends_string, 'id']
            segments.append(s)
        return segments

    subtrips['path_segments'] = subtrips.apply(lambda row:segmentize(row.↩
        path_segmentEnds), axis=1)
    subtrips['stop_segments'] = subtrips.apply(lambda row: [getSegment(end, row) ↩
        for end in row.stop_segmentEnds], axis=1)
    print(subtrips[['path_segments', 'path_segmentEnds']])
    subtrips.to_csv(processFolder + 'subtrips_wSegments_full.csv', index=False)
    subtrips = subtrips[['trip_id', 'shape_id', 'stop_segments', 'path_segments', '↩
        stops']]
    subtrips.to_pickle(processFolder + 'subtrips_wSegments.pkl')

def test():
    subtrips = pd.read_pickle(processFolder + 'subtrips_found.pkl')
    print(subtrips.head(10))
# test()


print('Step 1: Find segments which are in buffered bus routes.')
findCandidateSegments()

print('Step 2: Convert segments into graph with end vertices.')
getSegment()

print('Step 3: Find a start point for each segment.')
getSegment_startEndPoint()

# Step 4: Find the nearest segment end for each bus stop. (QGIS NNjoin tool)
    # input files: busStops and segmentStartPoint in processFolder ( add prefex 'S↩
        ')
    # output: stop_to_segmentEnd
    # all in processFolder.

print('Step 5: Find a representive segment start for each stop.')
stop_to_Segment()

print('Step 6: Find a unique trips which we will construct.')
mainUniqueBusRoutes()

print('Step 7: Find connected subsequent bus stops')
getConnectedConsequentStops()
```

```
print('Step 8: Construct connected trips')
processConnectedStops()

print('Step 9: Express trips in segments as SimMobility requires.')
segmentizeTrips()
```

Listing B.5: Bus GTFS Process Helper Function

```
import pandas as pd
import geopandas as gpd
import networkx as nx
from shapely.geometry import LineString, Point
from shapely.ops import transform
import matplotlib.pyplot as plt
from shapely.ops import nearest_points


def constructSeqLine(df, idColumn='id', seqColumn='seq_id', x='x', y='y'):
    df = df.sort_values([seqColumn])
    grouped = df.groupby(idColumn)
    rows = []
    for name,group in grouped:
        line_coords = group[[x, y]]
        line = LineString(tuple(x) for x in line_coords.to_records(index=False))
        rows.append( (name, line) )
    return pd.DataFrame.from_records(rows, columns=[idColumn, 'geometry'])

def readGTFS(gtfsFolder, trip='trip.csv', stoptime='stoptime.csv', shape='shape.csv↩
    ', stop='stops.csv'):
    # read
    gtfs_trips_df = pd.read_csv(gtfsFolder  + trip)
    gtfs_stoptime_df = pd.read_csv(gtfsFolder  + stoptime)
    gtfs_shape_df = pd.read_csv(gtfsFolder + shape)
    gtfs_stop_df = pd.read_csv(gtfsFolder + stop)
    # gtfs_stop_df = pd.read_csv(gtfsFolder + 'stops.txt')
    return gtfs_trips_df, gtfs_stoptime_df, gtfs_shape_df, gtfs_stop_df

def getRoadNetworkGeos(simFolder, fromCRS, toCRS):
    # segmentCoords = pd.read_csv(simFolder + 'segment-nodes.csv') #id,x,y,z,seq_id
    segmentCoords = pd.read_csv(simFolder + 'segment_polyline.csv')
    segmentGeo = constructSeqLine(segmentCoords)
    segmentGeo = gpd.GeoDataFrame(segmentGeo, crs=fromCRS)
    segmentGeo = segmentGeo.to_crs(toCRS)
    return segmentGeo

def getBusRouteGeo(busStops, busShapes, fromCRS, toCRS):
    busStopGeo = [Point(xy) for xy in zip(busStops.stop_lon, busStops.stop_lat)]
    busStopGeo = gpd.GeoDataFrame(busStops, crs=fromCRS, geometry=busStopGeo)
    busStopGeo = busStopGeo.to_crs(toCRS)
    busShapeGeo = constructSeqLine(busShapes, idColumn= 'shape_id', seqColumn='↩
        shape_pt_sequence', x='shape_pt_lon', y='shape_pt_lat')
    busGeo = gpd.GeoDataFrame(busShapeGeo, crs=fromCRS)
```

```
    busGeo = busGeo.to_crs(toCRS)
    return busStopGeo, busGeo

def getSegmentsWithEndNodes(simFolder, candidateSegment, CRS, base = 10000000):
    # turningPaths = pd.read_csv(simFolder + 'turning-attributes.csv')  #id,↩
        from_lane,to_lane,group_id,max_speed,tags
    turningPaths = pd.read_csv(simFolder + 'turning_path.csv')
    turningPaths["from_segment"] = turningPaths["from_lane"] // 100
    turningPaths["to_segment"] = turningPaths["to_lane"] // 100
    connector = pd.read_csv(simFolder + 'connector.csv') #id,from_segment,↩
        to_segment,from_lane,to_lane,is_true_connector,tags

    connectSegments = list(zip(turningPaths.from_segment, turningPaths.to_segment))
    connectSegments += list(zip(connector.from_segment, connector.to_segment))

    # Not necessary to include the following code since turning paths and ↩
        connectors are enough.
    # segments = segments.sort_values(['id'])
    # segSequence = segments.groupby('link_id')['id'].apply(list).tolist()
    # for seq in segSequence:
    #     if len(seq) > 1:
    #         connectSegments += list(zip(seq[:-1],seq[1:]))

    fromPoint = {} # seg --> node
    toPoint = {}
    NODE_ID = 1
    candidateSegIDs = set(candidateSegment.id.tolist())
    # Create nodes between segments    fromPoint --> seg1 --> (toPoint) = (↩
        fromPoint) --> seg2
    for seg1, seg2 in  connectSegments:
        # We are interested in only candidate segments
        if seg1 in candidateSegIDs and seg2 in candidateSegIDs:
            if seg1 in toPoint and seg2 in fromPoint:
                assert(toPoint[seg1] == fromPoint[seg2])
            elif seg1 in toPoint:
                fromPoint[seg2] = toPoint[seg1]
            elif seg2 in fromPoint:
                toPoint[seg1] = fromPoint[seg2]
            else:
                toPoint[seg1] = fromPoint[seg2] = NODE_ID
                NODE_ID += 1
    # Have to give nodes for dead ends
    for seg in candidateSegIDs:
        if seg not in toPoint: # seg --> toPoint --> None
            toPoint[seg] = NODE_ID
```

```
            NODE_ID += 1
        if seg not in fromPoint: # None --> fromPoint --> seg
            fromPoint[seg] = NODE_ID
            NODE_ID += 1

    print('MAX Node id ', NODE_ID)

    candidateSegment['from_node'] = candidateSegment.apply(lambda row: fromPoint[↩
        int(row.id)] if int(row.id) in fromPoint else row.link_id * base + row.↩
        sequence - 1, axis=1)
    candidateSegment['to_node'] = candidateSegment.apply(lambda row: toPoint[int(↩
        row.id)] if int(row.id) in toPoint else row.link_id * base + row.sequence, ↩
        axis=1)
    candidateSegment['from_node'] = candidateSegment['from_node'].astype('int')
    candidateSegment['to_node'] = candidateSegment['to_node'].astype('int')
    candidateSegment['ends'] = candidateSegment.apply(lambda row: str(int(row.↩
        from_node)) + '_' + str(int(row.to_node)), axis=1)
    print("Number of segments and unique ends: ", len(candidateSegment), len(↩
        candidateSegment.ends.unique()))
    return candidateSegment

def  findRouteCandidateSegments(segmentsWithNodes, busShapes, processFolder, ↩
    bufferSize=50):
    busShapes['geometry'] = busShapes['geometry'].buffer(distance=bufferSize)
    candidateShapeSegments = gpd.sjoin(segmentsWithNodes,busShapes, op='intersects'↩
        )
    # candidateShapeSegments.to_file(processFolder + 'busShape')
    return candidateShapeSegments

def getUniqueBusRoutes(stoptime_df, trips_df):
    trips_df.drop_duplicates(subset=['shape_id'], inplace=True)
    unique_trips = trips_df.trip_id.unique()
    stoptime_df = stoptime_df[stoptime_df.trip_id.isin(unique_trips)]
    stoptime_df.sort_values(by=['trip_id', 'stop_sequence'], inplace=True)
    trip_stops = stoptime_df.groupby('trip_id')['stop_id'].apply(list).to_frame()
    trip_stops = trip_stops.reset_index()
    trip_stops = trip_stops.merge(trips_df[['trip_id', 'shape_id']], on='trip_id', ↩
        how='left')
    print('Unique trips ', len(unique_trips))
    return trip_stops

def getShortestPath(trip_stops, shapeSegments, stopSegmentEnd):
    stopSegmentEnd.index = stopSegmentEnd.stop_id
    valid_stops = stopSegmentEnd[stopSegmentEnd['distance'] < 500].stop_id.tolist()
```

```
    stopConnectionSet = set()
    stopConnection_df = []
    num_no_connection = 0
    for indx, trip in trip_stops.iterrows():
        print('trip being.. ', indx)
        # Filter trip route segments
        routeSegments = shapeSegments[shapeSegments['shape_id'] == trip.shape_id]
        # print('routeSegments', len(routeSegments))
        edgeList = list(zip(routeSegments['from_node'], routeSegments['to_node']))
        G = nx.MultiDiGraph()
        G.add_edges_from(edgeList)

        # Search connection between consequent stops
        consequentStops = zip(trip.stop_id[:-1], trip.stop_id[1:])
        for startStop, endStop in consequentStops:
            # print(startStop, endStop)
            if ((startStop, endStop) not in stopConnectionSet) and (startStop in ↩
                valid_stops) and (endStop in valid_stops):
                startNode = stopSegmentEnd.loc[startStop, 'from_node']
                endNode = stopSegmentEnd.loc[endStop ,'to_node']
                if startNode in G and endNode in G:
                    try:
                        nodePath = nx.shortest_path(G, source=startNode, target=↩
                            endNode)
                        stopConnectionSet.add((startStop, endStop))
                        stopConnection_df.append((startStop, endStop, nodePath))
                    except nx.exception.NetworkXNoPath:
                        # print('No connection in between ', startStop, endStop)
                        num_no_connection += 1
                        pass
    print('Number of no connection ', num_no_connection)
    stopConnection_df = pd.DataFrame.from_records(stopConnection_df, columns=['↩
        startStop', 'endStop', 'nodePath'])
    return stopConnection_df
```

Listing B.6: Train GTFS Process

```
##############################################################################################
# Description: create SimMobility tables.
# Written by: Iveel Tsogsuren
##############################################################################################
import pandas as pd
from shapely.geometry import Point, LineString
import geopandas as gpd
from collections import defaultdict
from shapely.ops import transform
from functools import partial
import pyproj
from shapely.ops import split
from process_helper import*

processFolder='Outputs/Process/'
outputFolder='Outputs/to_db_may15/'

train_stop_cols = ['shape_id', 'x', 'y', 'z', 'platform_name', 'station_name', '↩
    type','op_year']
train_access_segment_cols = ['mrt_stop_id', 'segment_id']
train_platform_cols = ['platform_no', 'station_no', 'line_id', 'capacity', 'type', ↩
    'block_id', 'pos_offset', 'length']
pt_train_block_cols = ['block_id', 'default_speed_limit', 'accelerate_rate', '↩
    decelerate_rate', 'length']
pt_train_block_polyline_cols = ['polyline_id', 'x', 'y', 'z', 'sequence_no']
pt_train_dispatch_freq_cols = ['frequency_id', 'line_id', 'start_time', 'end_time',↩
     'headway_sec']
pt_train_route_platform_cols = ['line_id', 'platform_no', 'sequence_no']
pt_train_route_cols = ['line_id', 'block_id', 'sequence_no']
pt_train_platform_transfer_time_cols = ['station_no', 'platform_first', '↩
    platform_second', 'transfered_time_sec']
pt_opposite_lines_cols = ['line', 'opp_line']
dispatch_detialed_cols = ['trip_id','arrival_time','departure_time','stop_id','↩
    stop_sequence','service','service_id','stop_lat','stop_long','C_type']

# Selected shapes
# Manually select routes: opposite shapes
Route_to_shapes = {
10196: ['L5', 'L1'],
10210: ['L16', 'L23'],
10222: ['L51', 'L40'],
10223: ['L89', 'L75'],
10224: ['L153', 'L143'],
}
```

```
# line (name up to you), direction (opposite pairs), opp_direction (2,1)
Line_pair = namedtuple('Line_pair', 'line direction opp_direction')
LinePairs = {
'L5': Line_pair(line='A', direction=1, opp_direction=2),
'L1': Line_pair(line='A', direction=2, opp_direction=1),
'L16': Line_pair(line='B', direction=1, opp_direction=2),
'L23': Line_pair(line='B', direction=2, opp_direction=1),
'L51': Line_pair(line='C', direction=1, opp_direction=2),
'L40': Line_pair(line='C', direction=2, opp_direction=1),
'L89': Line_pair(line='D', direction=1, opp_direction=2),
'L75': Line_pair(line='D', direction=2, opp_direction=1),
'L153': Line_pair(line='F', direction=1, opp_direction=2),
'L143': Line_pair(line='F', direction=2, opp_direction=1),
}

# Expect each shape is a unique line (one direcion) and must pair up with other ↩
    shape.
route_df, shape, stops_df, stoptime_df, trip_df = get_gtfs(inFolder='Outputs/↩
    clean_gtfs/')
trip_wLine, stoptime_df, line_stoptime, line_toStartTimes = createLinesPlatformName↩
    (stoptime_df, stops_df, trip_df, route_df, shape, LinePairs)
train_route_platform, train_route, train_block, train_block_polyline, line_stoptime↩
     = createBlocks(line_stoptime, shape)
train_stop, train_platform, line_stoptime = platform_stations(line_stoptime, ↩
    stops_df)
dispatch_freq_table, dispatch_detialed = dispatch_freq(line_stoptime, ↩
    line_toStartTimes)

transfer_time = transfer_time(train_platform)
opposite_lines = line_stoptime.drop_duplicates(subset=['line_id'])
opposite_lines = opposite_lines.rename(columns={'line_id':'line'})[['line', '↩
    opp_line']]
train_uturn_platforms = create_uturn(train_route, train_platform)
mrt_line_properties = mrt_line_properties(train_route)
train_fleet = train_fleet(train_route)
train_transit_edge = transit_edge(dispatch_detialed)

train_uturn_platforms.to_csv(outputFolder + 'train_uturn_platforms.csv', index=↩
    False)
mrt_line_properties.to_csv(outputFolder + 'mrt_line_properties.csv', index=False)
train_fleet.to_csv(outputFolder + 'train_fleet.csv', index=False)
train_transit_edge.to_csv(outputFolder + 'rail_transit_edge.csv', index=False)

line_stoptime.to_csv(outputFolder + 'line_stoptime.csv', index=False)
```

```
transfer_time.to_csv(outputFolder + 'pt_train_platform_transfer_time.csv', index=↩
    False)
train_stop.to_csv(outputFolder + 'mrt_stop.csv', index=False)
train_stop.to_csv(outputFolder + 'train_stop.csv', index=False)
train_platform[train_platform_cols].to_csv(outputFolder + 'train_platform.csv', ↩
    index=False)
train_block.to_csv(outputFolder + 'pt_train_block.csv', index=False)
train_block_polyline.to_csv(outputFolder + 'pt_train_block_polyline.csv', index=↩
    False)
train_route_platform[pt_train_route_platform_cols].to_csv(outputFolder + '↩
    pt_train_route_platform.csv', index=False)
train_route[pt_train_route_cols].to_csv(outputFolder + 'pt_train_route.csv', index=↩
    False)
opposite_lines.to_csv(outputFolder + 'pt_opposite_lines.csv', index=False)
dispatch_freq_table.to_csv(outputFolder + 'pt_train_dispatch_freq.csv', index=False↩
    )
dispatch_detialed[dispatch_detialed_cols].to_csv(outputFolder + '↩
    weekday_train_seq_31Mar18.csv', index=False)


# print('Convert segments ----------------------------------------')
convertSegment(simFolder='Auto_sprawl_drive_main/SimMobility/')
access_segment = find_access_segment(train_stop, simFolder='Auto_sprawl_drive_main/↩
    simmobility/')
access_segment.to_csv(outputFolder + 'train_access_segment.csv', index=False)
# #
# We need train stops in lat and long for public transit graph generation.
convertCoordinates(outputFolder)
```

Listing B.7: Train GTFS Process Helper Function

```
##############################################################################################
# Description: helper functions for creating SimMobility tables.
# Written by: Iveel Tsogsuren
##############################################################################################
import pandas as pd
import geopandas as gpd
import datetime
from collections import defaultdict, namedtuple
from shapely.geometry import Point, LineString, MultiPoint, MultiLineString, ↩
    mapping, Polygon
from shapely.ops import transform, split, nearest_points
import fiona
from functools import partial
import pyproj
import numpy as np


LAT_LONG_CRS = {'init': 'epsg:4326'}
BALTIMORE_CRS = {'init': 'epsg:6487'}
CURRENT_CRS = BALTIMORE_CRS

WEEKDAY_SERVICES = ['1']
HEADWAY_SEC = 60

DEFAULT_SPEED_LIMIT = 70
ACCELERATE_RATE = 1.1
DECELERATE_RATE = 1.1
Route_type = namedtuple('Route_type', 'type_name type_int platform_length capacity'↩
    )
ROUTE_TYPE = {
0: Route_type(type_name='LR', type_int=0, platform_length=56, capacity=105),
1: Route_type(type_name='CR', type_int=1, platform_length=60, capacity=170),
2: Route_type(type_name='HR', type_int=1, platform_length=142, capacity=1920),
}

# 0 - Tram, Light Rail, Streetcar - 900
# 1 - Subway, Metro - 400
# 2 - Rail - 100
# 3 - Bus - 700
# 4 - Ferry - 1000
# 5 - Cable Car - ?
# 6 - Gondola, Suspended cable car - 1300
# 7 - Funicular - 1400
```

```
def convertCoordinates(inputFolder, toCRS=CURRENT_CRS):
    """ Convert train stops crs from xy to lat and long. """
    stops = pd.read_csv(inputFolder + 'mrt_stop.csv')
    stops_cols = stops.columns
    coords = [Point(xy) for xy in zip(stops.x, stops.y)]
    stops = gpd.GeoDataFrame(stops, crs=toCRS, geometry=coords)
    stops = stops.to_crs(LAT_LONG_CRS)
    stops['x'] = stops['geometry'].x
    stops['y'] = stops['geometry'].y
    stops = stops[stops_cols]
    stops.to_csv(inputFolder + 'mrt_stop_wgs84.csv', index=False)


def get_gtfs(inFolder='Merged_clean/'):
    """ Read GTFS files into panda dataFrames. """
    route = pd.read_csv(inFolder + 'route.csv')
    shape = pd.read_csv(inFolder + 'shapes.csv')
    stops = pd.read_csv(inFolder + 'stops.csv')
    stoptime = pd.read_csv(inFolder + 'stop_times.csv')
    trip = pd.read_csv(inFolder + 'trips.csv')
    # Set shape_id as numbers
    # shape_ids = shape.shape_id.unique()
    # shapeIDmap = dict(zip(shape_ids, ['L' + str(i) for i in range(1, len(↩
        shape_ids)+1)]))
    # shape['shape_id'] = shape.apply(lambda row: shapeIDmap[row.shape_id], axis=1)
    # trip['shape_id'] = trip.apply(lambda row: shapeIDmap[row.shape_id], axis=1)
    return route, shape, stops, stoptime, trip

def merge_two_dicts(x, y):
    z = x.copy()   # start with x's keys and values
    z.update(y)    # modifies z with y's keys and values & returns None
    return z

def createLinesPlatformName(stoptime_df, stops_df, trip_df, route, shape, LineInfo)↩
    :
    """
    Route and shape_id(s) are consistent in input files at this point.
    Map trips to lines based on trip shapes.
    Extend stoptime by line_id and shape_id
    For each shape, find all trips and choose a representive trip.
    Parameters
    ----------
    stoptime_df, stops_df, trip_df, route, shape : standard GTFS files in pandas ↩
        dataFrames.
    LineInfo : Specification for which an opposite pair of shapes each route has.
```

```
    Returns
    -------
    stoptime_df : added extra columns line_id, direction, shape_id, etc.
    trip : lines' only representive trips
    linetime : only representive trips' stoptime
    line_toStartTimes : line's start times of all trips
    """
    # keep a representive trip (stop sequence) for each shape.
    # We will assume that other trips with the same line_id have the same stop ↩
        sequences.
    trip = trip_df.copy()
    trip['direction'] = trip.apply(lambda row: LineInfo[row.shape_id].direction, ↩
        axis=1)
    trip['generic_line'] = trip.apply(lambda row: LineInfo[row.shape_id].line, axis↩
        =1)
    trip['line_id'] = trip.apply(lambda row: row.generic_line + '_' + str(row.↩
        direction), axis=1)
    trip['opp_line'] = trip.apply(lambda row: row.generic_line + '_' + str(LineInfo↩
        [row.shape_id].opp_direction), axis=1)

    stoptime_df = stoptime_df.merge(trip[['trip_id', 'line_id', 'opp_line', '↩
        route_id', 'shape_id', 'service_id', 'generic_line', 'direction']], on='↩
        trip_id', how='left')
    # Choose a representive trip for each shape.
    repTrips = trip_df.groupby('shape_id')['trip_id'].apply(lambda row: list(row)↩
        [0])
    trip = trip_df[trip_df.trip_id.isin(repTrips.values)]
    line_toStartTimes = stoptime_df[stoptime_df.stop_sequence == 1].groupby('↩
        line_id')['arrival_time'].apply(set)

    linetime = stoptime_df[stoptime_df.trip_id.isin(repTrips.values)]
    linetime = linetime.merge(stops_df, on='stop_id', how='left')
    linetime = linetime.merge(route[['route_id', 'route_type']], on='route_id', how↩
        ='left')
    linetime = linetime[linetime.service_id.isin(WEEKDAY_SERVICES)]

    linetime.sort_values(by=['trip_id', 'stop_sequence'], inplace=True) # get a ↩
        unique trip
    linetime['type'] = linetime.apply(lambda row: ROUTE_TYPE[row.route_type].↩
        type_name, axis=1)
    linetime['type_int'] = linetime.apply(lambda row: ROUTE_TYPE[row.route_type].↩
        type_int, axis=1)
    linetime['capacity'] = linetime.apply(lambda row: ROUTE_TYPE[row.route_type].↩
        capacity, axis=1)
    linetime['length'] = linetime.apply(lambda row: ROUTE_TYPE[row.route_type].↩
```

```
        platform_length, axis=1)
    # Name platforms. Stops in different lines have different platfrom names.
    def setPlatformNo(row, df):
        if row.direction == 1:
            return row.generic_line + str(row.stop_sequence) + '_' + str(row.↩
                direction)
        else:
            num_stops = df[df.line_id == row.line_id].stop_sequence.max()
            return row.generic_line + str(num_stops + 1 - row.stop_sequence) + '_' ↩
                + str(row.direction)

    linetime['platform_no'] = linetime.apply(lambda row: setPlatformNo(row, ↩
        linetime), axis=1)
    # print(linetime[['stop_sequence', 'line_id', 'opp_line', 'platform_no']])
    print('Number of stops in lines', len(linetime.stop_id.unique()))
    print('Number of trips in lines', len(linetime.trip_id.unique()))
    return trip, stoptime_df, linetime, line_toStartTimes


def platform_stations(linetime, stops):
    """
    Construct train stations. Train platforms are allocated to the single train
    station if they are at the same location.
    Parameters
    _________
    linetime : representive trips' stoptime
    stops : a standard stops GTFS (cleaned such that oppostie directional two
                                   stops are the same location)
    Returns
    _______
    train_stop : SimMobility train_stop table (unique station_no). This table is
                 used for public transit graph generation and has a different
                 naming convention.
    train_platform : SimMobility train_platform table (all platforms).
    line_stoptime : added station_no column.
    """
    # Find stations
    coords_to_lines = defaultdict(list)
    coords_to_platforms = defaultdict(list)
    # Find platforms (without direction) of lines which share the same stop ↩
        coordinates.
    # These platform share the same station.
    for indx, row in linetime.iterrows():
        # Check if there is already representive stop from line
        if row.generic_line not in coords_to_lines[(row.stop_lon, row.stop_lat)]:
```

```
            coords_to_lines[(row.stop_lon, row.stop_lat)].append(row.generic_line)
            coords_to_platforms[(row.stop_lon, row.stop_lat)].append(row.↩
                platform_no.split('_')[0]) # row.generic_line))
    print('###### Number of stations ', len(coords_to_lines))
    coords_to_stationName = {}
    for coords, line_list in coords_to_platforms.items():
        coords_to_stationName[coords] = '/'.join(sorted(set(line_list)))
    print('platform and station coords_to_platforms')
    print(coords_to_platforms)
    print('platform and station coords_to_stationName')
    print(coords_to_stationName)
    print('platform and station coords_to_lines')
    print(coords_to_lines)

    linetime['station_no'] = linetime.apply(lambda row: coords_to_stationName[(row.↩
        stop_lon, row.stop_lat)], axis=1)
    # Create train_stop table (Be careful with absurd name convention)
    train_stop = linetime.drop_duplicates(subset=['station_no'])
    train_stop.rename(columns={'stop_y':'y', 'stop_x':'x', 'station_no':'↩
        platform_name', 'stop_name':'station_name'}, inplace=True)
    train_stop['shape_id'] = train_stop.reset_index().index + 1
    train_stop['id'] = train_stop['shape_id']
    train_stop['op_year'] = 1997
    train_stop['z'] = 0
    train_stop = train_stop[['shape_id', 'x', 'y', 'z', 'id', 'platform_name', '↩
        station_name', 'type', 'op_year']]

    # Create train_platform table
    train_platform = linetime.copy()
    train_platform['pos_offset'] = 0
    train_platform['name_len'] = train_platform.apply(lambda row: len(row.↩
        station_no), axis=1)
    train_platform.rename(columns={'type': 'string_type', 'type_int':'type'}, ↩
        inplace=True)
    train_platform = train_platform[['platform_no', 'station_no', 'line_id', '↩
        capacity', 'type', 'block_id', 'pos_offset', 'length']]
    print('Number of stations in platform table', len(train_platform.station_no.↩
        unique()))
    print('Number of platforms in platform table', len(train_platform.platform_no.↩
        unique()))
    # print(linetime)
    return train_stop, train_platform, linetime

def convertSegment(simFolder=None, fromCRS=LAT_LONG_CRS, toCRS=CURRENT_CRS):
    segment_points = pd.read_csv(simFolder + 'segment-nodes.csv')
```

```
    pointGeo = [Point(xy) for xy in zip(segment_points.x, segment_points.y)]
    segment_points = gpd.GeoDataFrame(segment_points, crs=fromCRS, geometry=↩
        pointGeo)
    segment_points = segment_points.to_crs(toCRS)
    segment_points['x'] = segment_points['geometry'].x
    segment_points['y'] = segment_points['geometry'].y
    segment_points['coords'] = segment_points.apply(lambda row: (row.x, row.y), ↩
        axis=1)
    segment_points = segment_points[['id', 'x', 'y', 'coords']]
    segment_points.to_csv(simFolder + 'segment-baltimore-crs.csv')

def find_access_segment(train_stop, simFolder=None, fromCRS=LAT_LONG_CRS, toCRS=↩
    CURRENT_CRS):
    segment_points = pd.read_csv(simFolder + 'segment-baltimore-crs.csv')
    segment_points['coords'] = segment_points.apply(lambda row: (row.x, row.y), ↩
        axis=1)
    points = MultiPoint(segment_points.coords.tolist())
    print('unique stops ', len(train_stop))
    access_segment = []
    for indx, row in train_stop.iterrows():
        nearest = nearest_points(points, Point(row.x, row.y))[0]
        access_segment.append(((nearest.x, nearest.y), row.shape_id))
    access_segment = pd.DataFrame.from_records(access_segment, columns = ['↩
        nearest_coord', 'mrt_stop_id'])
    access_segment = access_segment.merge(segment_points, left_on='nearest_coord', ↩
        right_on='coords', how='left')
    access_segment = access_segment[['mrt_stop_id', 'id']].rename(columns={'id':'↩
        segment_id'})
    access_segment.drop_duplicates(subset=['mrt_stop_id'], inplace=True)
    return access_segment

def constructSeqLine(df, idColumn='id', seqColumn='seq_id', x='x', y='y'):
    df = df.sort_values([seqColumn])
    grouped = df.groupby(idColumn)
    rows = []
    for name,group in grouped:
        line_coords = group[[x, y]]
        line = LineString(tuple(x) for x in line_coords.to_records(index=False))
        rows.append( (name, line) )
    return pd.DataFrame.from_records(rows, columns=[idColumn, 'geometry'])

def addRouteGeo(stops_df, shapes_df, fromCRS=LAT_LONG_CRS, toCRS=CURRENT_CRS):
    # print('stops ', stops.columns)
    # print('shapes ', shapes.columns)
    stopGeo = [Point(xy) for xy in zip(stops_df.stop_lon, stops_df.stop_lat)]
```

```
    # print('stopGeo ', stopGeo)
    stops = gpd.GeoDataFrame(stops_df, crs=fromCRS, geometry=stopGeo)
    stops = stops.to_crs(toCRS)
    shapeGeo = constructSeqLine(shapes_df, idColumn= 'shape_id', seqColumn='↩
        shape_pt_sequence', x='shape_pt_lon', y='shape_pt_lat')
    shapeGeo = gpd.GeoDataFrame(shapeGeo, crs=fromCRS)
    shapeGeo = shapeGeo.to_crs(toCRS)
    return stops, shapeGeo


def write(a_shape, shape_type, toFile):
    # Define a polygon feature geometry with one attribute
    schema = {
        'geometry': shape_type,
        'properties': {'id': 'int'},
    }

    # Write a new Shapefile
    with fiona.open(toFile, 'w', 'ESRI Shapefile', schema) as c:
        ## If there are multiple geometries, put the "for" loop here
        c.write({
            'geometry': mapping(a_shape),
            'properties': {'id': 123},
        })

def createBlocks(stoptime, shape):
    """
    Project stops onto the corresponding shapes
    Split shapes into blocks such that each stop is a middle or
    end point (for two ends) of a block.
    Index (ID) blocks and their polylines -> train_block, block_polyline
    Express routes in stop sequence and block sequence -> train_route_platform, ↩
        train_route
    Express each stop sequence by a block sequence -> train_route
    stoptime
    Parameters
    __________
    stoptime : representive trips' stoptimes
    shape : a standard GTFS shape
    Returns
    _______
    train_route_platform : route expressed in a stop (platform) sequence
    train_route : route expressed in a block sequence
    train_block : blocks
    block_polyline : block polylines
```

```
    stoptime : added by block_id based on stop_id
    """
    stoptime, shape = addRouteGeo(stoptime, shape)
    stoptime.sort_values(by=['line_id', 'stop_sequence'], inplace=True)
    stoptime['x'] = stoptime.geometry.x
    stoptime['y'] = stoptime.geometry.y
    shape.index = shape.shape_id

    def neartestPoint(row, shapes_df):
        # print('row ', row)
        line = MultiPoint(shapes_df.at[row.shape_id, 'geometry'].coords)
        point = row.geometry
        projected_point = nearest_points(line, point)
        return (projected_point[0].x, projected_point[0].y) # make a tuple
    stoptime['proj_on_line'] = stoptime.apply(lambda row: neartestPoint(row, shape)↩
        , axis=1)

    shape_stops = stoptime.groupby('shape_id')['proj_on_line'].apply(list).to_frame↩
        ().reset_index()
    shape_stops.index = shape_stops.shape_id

    shape_to_blocks = {}
    for shape_id, row in shape_stops.iterrows():
        proj_stops = row['proj_on_line']
        if len(proj_stops) > 2 and len(set(proj_stops)) == len(proj_stops):
            proj_stops_cutter = MultiPoint(proj_stops[1:-1]) # exclude end stops
            line = shape.at[shape_id, 'geometry']
            splitted = split(line, proj_stops_cutter)
            splitted_lines = MultiLineString(splitted)
            # Cut into blocks
            cutter_points = [s.interpolate(0.5, normalized=True) for s in splitted]
            lineMultipoint = MultiPoint(line.coords)
            cut_on = MultiPoint([nearest_points(lineMultipoint, P)[0] for P in ↩
                cutter_points])
            line_blocks = split(line, cut_on)
            if len(proj_stops) == len(line_blocks):
                shape_to_blocks[shape_id] = list(line_blocks)

    # Create block related tables
    print('Number of lines splitted into blocks ', len(shape_to_blocks))
    shape_stops = shape_stops[shape_stops.shape_id.isin(shape_to_blocks.keys())]
    shape_stopseq = stoptime[stoptime.shape_id.isin(shape_to_blocks.keys())]

    shape_stopseq = shape_stopseq.groupby('shape_id')['platform_no'].apply(list)
    shape_stopseq = shape_stopseq.to_frame().reset_index()
```

```
    shape_stopseq.index = shape_stopseq.shape_id
    platform_to_block = {}
    train_route_platform = []
    BLOCK_ID = 1
    block_to_poly = {}
    for shape_id, row in shape_stopseq.iterrows():
        sequence_no = 0
        for index, platform in enumerate(row.platform_no):
            stop_b = (shape_id, platform, sequence_no)
            platform_to_block[platform] = BLOCK_ID
            # print(len(shape_to_blocks[line_id]), len(row.platform_no))
            # print(shape_to_blocks[line_id])
            block_to_poly[BLOCK_ID] = shape_to_blocks[shape_id][index]
            train_route_platform.append(stop_b)
            BLOCK_ID += 1
            sequence_no +=1

    train_route_platform = pd.DataFrame.from_records(train_route_platform, columns↩
        =['shape_id', 'platform_no', 'sequence_no'])
    print('train_route_platform ', train_route_platform.head(10))
    train_route_platform = train_route_platform.merge(stoptime[['shape_id', '↩
        line_id']], on='shape_id', how='left')
    train_route_platform = train_route_platform.drop_duplicates(subset=['line_id','↩
        platform_no'])
    train_route = train_route_platform.copy()
    train_route['block_id'] = train_route.apply(lambda row: platform_to_block[row.↩
        platform_no], axis=1)
    stoptime['block_id'] = stoptime.apply(lambda row: platform_to_block[row.↩
        platform_no], axis=1)
    train_route = train_route[['line_id', 'block_id', 'sequence_no']]

    # Create blocks
    block_polyline = []
    train_block = []
    for blockID, lineString in block_to_poly.items():
        train_block.append((blockID, lineString.length))
        for indx, coord in enumerate(list(lineString.coords), 1):
            poly = (blockID, coord[0], coord[1], 0, indx)
            block_polyline.append(poly)

    train_block = pd.DataFrame.from_records(train_block, columns=['block_id', '↩
        length'])
    block_polyline = pd.DataFrame.from_records(block_polyline, columns=['↩
        polyline_id', 'x', 'y', 'z', 'sequence_no'])
    train_block['default_speed_limit'] = DEFAULT_SPEED_LIMIT
```

```
    train_block['accelerate_rate'] = ACCELERATE_RATE
    train_block['decelerate_rate'] = DECELERATE_RATE
    print('Number of platforms ', len(train_route_platform.platform_no.unique()))
    print('Number of blocks ', len(train_block.block_id.unique()))
    print('Number of lines ', len(train_route.line_id.unique()))
    return train_route_platform, train_route, train_block, block_polyline, stoptime


# if a train starts before midnight but ends after midnight, we drop this
def overMidnight(row):
    # if row.start
    time_units = time.split(':')
    if int(time_units[0]) >=24:
        return True
    return False

# if hour is larger than 24, convert it to the today's morning as 00:xx:xx
def StartTime(time):
    time_units = time.split(':')
    hours = int(time_units[0])
    hours = hours%24
    minutes = time_units[1]
    seconds = time_units[2]
    correct_hours ='foo'
    if hours == 0:
        correct_hours = '00'
    elif hours < 10:
        correct_hours = '0' + str(hours)
    else:
        correct_hours = str(hours)
    return ':'.join([correct_hours,minutes, seconds])


def EndTime(time):
    # time = row["Start_Time"]
    start_time = datetime.datetime.strptime(time, "%H:%M:%S")
    end_time = start_time + datetime.timedelta(minutes=1)
    return end_time.strftime("%H:%M:%S")


def InSeconds(time):
    time_units = time.split(':')
    seconds = int(time_units[0])*3600 + int(time_units[1])*60 + int(time_units[2])
    return seconds
```

```
def formatSecond(deltaSeconds):
    if deltaSeconds > 23*3600 + 59*60 + 59: #23:59:59
        return '23:59:59'
    totalSec = deltaSeconds %60
    totalMin = (deltaSeconds // 60) % 60
    totalHour = (deltaSeconds //3600) % 24
    total_time = datetime.time(int(totalHour), int(totalMin), int(totalSec))
    total_time =  total_time.strftime("%H:%M:%S")
    return total_time

def dispatch_freq(line_stoptime, line_toStartTimes, headway_sec=300):
    """
    Find line dispatch time tables.
    Parameters
    ----------
    line_stoptime : lines' representive trips' stoptimes
    stoptime : an original standard GTFS stoptime
    line_toStartTimes :  line's start times of all trips
    Returns
    -------
    dispatch : SimMobility dispatch_freq table which is head based. Thus,
               start_time is time the first train starts a trip and
               end_time is time the last train starts a trip
    dispatch_detialed : Dispatch frequency table for generating a public tranist
               graph. One trip for each lane must be speficied with all stop times.
               It is used for computing travel times between stops.
    """
    # SimMobility dispatch_freq table [frequency_id, line_id, start_time, end_time,↩
        headway_sec]
    dispatch_freq = []
    for line, startTimes in line_toStartTimes.items():
        startTimeInSeconds = [InSeconds(t) for t in startTimes]
        start_time = min(startTimeInSeconds)
        end_time = max(startTimeInSeconds)
        if start_time == end_time:
            end_time += 2*3600
        dispatch_freq.append((line, start_time, end_time, headway_sec))
    dispatch_freq = pd.DataFrame.from_records(dispatch_freq, columns=['line_id', '↩
        start_time', 'end_time', 'headway_sec'])
    print(dispatch_freq)
    dispatch_freq['start_time'] = dispatch_freq.apply(lambda row: formatSecond(row.↩
        start_time), axis=1)
    dispatch_freq['end_time'] = dispatch_freq.apply(lambda row: formatSecond(row.↩
        end_time), axis=1)
```

```
    dispatch_freq['frequency_id'] = dispatch_freq.index
    dispatch_freq = dispatch_freq[['frequency_id', 'line_id', 'start_time', '↩
        end_time', 'headway_sec']]

    # Public transit generation dispatch table
    # ['trip_id','arrival_time','departure_time','stop_id','stop_sequence','service↩
        ','service_id','stop_lat','stop_long','C_type']
    print('line_stops------- ', line_stoptime.columns)
    dispatch_detialed = line_stoptime[['trip_id', 'stop_sequence', 'arrival_time','↩
        line_id',
                        'departure_time', 'station_no', 'type', 'stop_lon', '↩
                            stop_lat']]
    print('Before rename ', dispatch_detialed.columns)
    dispatch_detialed.rename(columns={'station_no':'stop_id', 'type':'C_type',
                            'stop_lon':'stop_long', 'line_id': 'service_id'}, ↩
                                inplace=True)
    # dispatch_detialed = line_stoptime.rename(columns={'start_time':'arrival_time↩
        ', 'arrival_time':'arrival_time_old', 'end_time':'departure_time',
    #         'station_no':'stop_id','type':'C_type', 'stop_lon':'stop_long', '↩
        service_id':'service_id_gtfs', 'line_id':'service_id'})
    print(dispatch_detialed.columns)
    dispatch_detialed['service'] =  dispatch_detialed['service_id']
    dispatch_detialed.sort_values(by=['service_id', 'trip_id', 'stop_sequence'], ↩
        inplace=True)
    print('number of frequencey ', len(dispatch_freq.frequency_id.unique()))
    print('number of lines ', len(dispatch_detialed.service_id.unique()))
    return dispatch_freq, dispatch_detialed


def transfer_time(platforms):
    train_platform = ['platform_no', 'station_no', 'line_id', 'capacity', 'type', '↩
        block_id', 'pos_offset', 'length']
    pt_train_platform_transfer_time = ['station_no', 'platform_first', '↩
        platform_second', 'transfered_time_sec']
    transfer_time = []
    for station, platforms in platforms.groupby('station_no'):
        # print(type(platforms))
        # return
        for i1, p1 in platforms.iterrows():
            for i2, p2 in platforms.iterrows():
                if p1.platform_no != p2.platform_no:
                    if p1.platform_no.split('_')[0] == p2.platform_no.split('_')↩
                        [0]:
                        t1 = (station, p1.platform_no, p2.platform_no, 0)
                        t2 = (station, p2.platform_no, p1.platform_no, 0)
```

```
                    else:
                        t1 = (station, p1.platform_no, p2.platform_no, 60)
                        t2 = (station, p2.platform_no, p1.platform_no, 60)
                    transfer_time += [t1, t2]
    transfer_time = pd.DataFrame.from_records(transfer_time, columns = ['station_no↩
        ', 'platform_first', 'platform_second', 'transfered_time_sec'])
    transfer_time.drop_duplicates(subset=['station_no', 'platform_first', '↩
        platform_second'], inplace=True)
    return transfer_time


def create_uturn(pt_train_route, pt_train_platform):
    # Step 18: Create train_uturn_platforms table
    pt_train_route_copy = pt_train_route.copy()
    pt_train_platform_copy = pt_train_platform.copy()
    pt_route_platform_merge = pd.merge(pt_train_route_copy, pt_train_platform_copy,↩
         on='block_id')

    train_uturn_platforms = []
    route_platform =  pd.merge(pt_train_route_copy, pt_train_platform_copy, on='↩
        block_id')
    for lineID, line_route_platform in route_platform.groupby('line_id_x'):
        line_route_platform.sort_values(by=['sequence_no'], inplace=True)
        p = line_route_platform['platform_no'].values.tolist()[0]
        train_uturn_platforms.append((p, lineID))
    train_uturn_platforms = pd.DataFrame.from_records(train_uturn_platforms, ↩
        columns=['platformno', 'lineid'])
    return train_uturn_platforms


def mrt_line_properties(pt_train_route):
    # Step 19: create mrt_line_properties table
    mrt_line_properties = pd.DataFrame()
    print(pt_train_route.columns)
    mrt_line_properties['line_id'] = pd.unique(pt_train_route['line_id'])
    mrt_line_properties['min_dwell_time_normal'] = [20] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['min_dwell_time_terminal'] = [60] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['min_dwell_time_interchange'] = [50] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['dwell_time_formula_first_coeff'] = [12.22] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['dwell_time_formula_second_coeff'] = [2.27] *len(pd.unique(↩
        pt_train_route['line_id']))
```

```
    mrt_line_properties['dwell_time_formula_third_coeff'] = [1.82] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['dwell_time_formula_fourth_coeff'] = [0.00062] *len(pd.↩
        unique(pt_train_route['line_id']))

    mrt_line_properties['safe_operation_distance_meter'] = [50] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['safe_operation_headway_sec'] = [90] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['min_dis_behind_unscheduled_train'] = [200] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['train_length'] = [138] *len(pd.unique(pt_train_route['↩
        line_id']))
    mrt_line_properties['train_capacity'] = [1600] *len(pd.unique(pt_train_route['↩
        line_id']))
    mrt_line_properties['distance_arriving_at_platform'] = [0.001] *len(pd.unique(↩
        pt_train_route['line_id']))
    mrt_line_properties['max_dwell_time'] = [120] *len(pd.unique(pt_train_route['↩
        line_id']))
    return mrt_line_properties


def train_fleet(pt_train_route):
    # Step 25: create train_fleet table
    train_fleet = pd.DataFrame()
    train_fleet['line'] = pd.unique(pt_train_route['line_id'])
    train_fleet['min_initial_id'] = [1] * len(pd.unique(pt_train_route['line_id']))
    train_fleet['max_initial_id'] = [5] * len(pd.unique(pt_train_route['line_id']))
    return train_fleet


def deltaTime(start, end):
    sStart = start.split(':')
    sEnd = end.split(':')
    sEnd = int(sEnd[0])*3600 + int(sEnd[1])*60 + int(sEnd[2])
    sStart = int(sStart[0])*3600 + int(sStart[1])*60 + int(sStart[2])
    deltaSeconds = sStart - sEnd
    return deltaSeconds

# Neighbor station travel times
def transit_edge(weekday_train_seq): #TODO
    # columns = ['from_stn', 'to_stn', 'travel_time', 'type'] 'RTS', 'TRS'
    # train_platform_cols = ['platform_no', 'station_no', 'line_id', 'capacity', '↩
        type', 'block_id', 'pos_offset', 'length']
    # dispatch_detialed_cols = ['trip_id','arrival_time','departure_time','stop_id↩
```

```
        ','stop_sequence','service','service_id','stop_lat','stop_long','C_type']
    # dispatch = pd.merge(dispatch_detialed[['trip_id','arrival_time','↩
        departure_time', 'stop_id', 'platform_no']],
    #                     train_platform[['platform_no', 'station_no']], on='↩
        platform', how='left')
    # transit_df = []
    # for tripID, group in dispatch.groupby('trip_id'):
    #     from_stn = group.stop_id.tolist()[:-1]
    #     to_stn = from_to = group.stop_id.tolist()[:-1]
    # transit_df = pd.DataFrame.from_records(transit_df, columns=['from_stn', '↩
        to_stn', 'travel_time'])

    # Step 23: create rail_transit_edge table
    # print(weekday_train_seq.head(3))
    test = weekday_train_seq.copy()
    fro_stn = []
    to = []
    dur = []
    print(weekday_train_seq.columns)
    for name, group in weekday_train_seq.groupby('trip_id'):
        fro_stn += group['stop_id'][0:len(group)-1].tolist()
        to += group['stop_id'][1:len(group)].tolist()
        dur +=[ deltaTime(start, end) for start, end in list(zip(group['↩
            arrival_time'][1:], group['departure_time'][:-1]))]
        # dur += (group['arrival_time'][1:len(group)].as_matrix()- group['↩
            departure_time'][0:len(group)-1].as_matrix()).astype('timedelta64[s]').↩
            astype(int).tolist()

    d = {'from_stn': fro_stn,
         'to_stn': to,
         'travel_time': dur}
    rail_transit_edge = pd.DataFrame(d)
    rail_transit_edge = rail_transit_edge.groupby(['from_stn','to_stn']).mean().↩
        reset_index(level=[0,1])
    # rail_transit_edge['travel_time_format'] = rail_transit_edge.apply(lambda row:↩
         formatSecond(row.travel_time), axis=1)
    rail_transit_edge['type'] = ['RTS'] * len(rail_transit_edge)
    cols = ['from_stn','to_stn','travel_time','type']
    rail_transit_edge = rail_transit_edge[cols]
    return rail_transit_edge
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] *ACS public use microdata sample (PUMS).* Washington, D.C. : U.S. Census Bureau, 2015.

[2] Muhammad Adnan, Francisco C Pereira, Carlos Miguel Lima Azevedo, Kakali Basak, Milan Lovric, Sebastián Raveau, Yi Zhu, Joseph Ferreira, Christopher Zegras, and M Ben-Akiva. Simmobility: A multi-scale integrated agent-based simulation platform. In *95th Annual Meeting of the Transportation Research Board Forthcoming in Transportation Research Record*, 2016.

[3] Derek Azar, Ryan Engstrom, Jordan Graesser, and Joshua Comenetz. Generation of fine-scale population layers using multi-resolution satellite imagery and geospatial data. *Remote Sensing of Environment*, 130:219–232, 2013.

[4] Michael Balmer, Marcel Rieser, Konrad Meister, David Charypar, Nicolas Lefebvre, and Kai Nagel. Matsim-t: Architecture and simulation times. In *Multi-agent systems for traffic and transportation engineering*, pages 57–78. IGI Global, 2009.

[5] Johan Barthelemy and Philippe L Toint. Synthetic population generation without a sample. *Transportation Science*, 47(2):266–279, 2013.

[6] Filipe Batista e Silva, Javier Gallego, and Carlo Lavalle. A high-resolution population grid map for europe. *Journal of Maps*, 9(1):16–28, 2013.

[7] Catherine Baumont, Cem Ertur, and Julie Gallo. Spatial analysis of employment and population density: the case of the agglomeration of dijon 1999. *Geographical analysis*, 36(2):146–176, 2004.

[8] Carolien Beckx, Luc Int Panis, Jean Vankerkom, Davy Janssens, Geert Wets, and Theo Arentze. An integrated activity-based modelling framework to assess vehicle emissions: approach and application. *Environment and Planning B: Planning and Design*, 36(6):1086–1102, 2009.

[9] Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Browser Download This Paper*, 2016.

[10] Mark Bradley, John L Bowman, and Bruce Griesenbeck. Sacsim: An applied activity-based model system with fine-level spatial and temporal resolution. *Journal of Choice Modelling*, 3(1):5–31, 2010.

[11] Census.U.S. Census Bureau. United states census bureau / american factfinder. http://factfinder2.census.gov, 2015.

[12] Maryland Department of Planning. Maryland department of planning 2010 land use/land cover, 2010.

[13] Jerry M Faris, Lisa B Beever, and Mike Brown. Geography information system (gis) and urban land use allocation model (ulam) techniques for existing and projected land use data. In *Seventh National Conference on Transportation Planning for Small and Medium-Sized CommunitiesTransportation Research BoardFederal Highway AdministrationMack-Blackwell Transportation Center*, 2000.

[14] Bilal Farooq, Michel Bierlaire, Ricardo Hurtubia, and Gunnar Flötteröd. Simulation based population synthesis. *Transportation Research Part B: Methodological*, 58:243–263, 2013.

[15] Singapore-MIT Alliance for Research and Technology. Simmobility, 2017.

[16] government of Maryland. Maryland gis data catalog. http://data.imap.maryland.gov/datasets?q=school, 2018.

[17] Mordechai Haklay. How good is volunteered geographical information? a comparative study of openstreetmap and ordnance survey datasets. *Environment and planning B: Planning and design*, 37(4):682–703, 2010.

[18] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[19] Yafei Han, Jimi Oke, Sean Hua, Jin Zhou, Carlos Miguel Lima Azevedo, Christopher Zegras, Joseph Ferreira, and Moshe Ben-Akiva. Global urban typology discovery with a latent class choice.

[20] Kerstin Hermes and Michael Poulsen. A review of current methods to generate synthetic spatial microdata using reweighting and future directions. *Computers, Environment and Urban Systems*, 36(4):281–290, 2012.

[21] https://www.census.gov/programs-surveys/acs/guidance/training-presentations/acs-intro pums.html. Introduction to the public use microdata sample (pums), February 2017.

[22] Mitchel Langford and David J Unwin. Generating and mapping population density surfaces within a geographical information system. *The Cartographic Journal*, 31(1):21–26, 1994.

[23] Yilan Liao, Jinfeng Wang, Bin Meng, and Xinhu Li. Integration of gp and ga for mapping population distribution. *International Journal of Geographical Information Science*, 24(1):47–67, 2010.

[24] Ying Long and Zhenjiang Shen. Disaggregating heterogeneous agent attributes and location. *Computers, Environment and Urban Systems*, 42:14–25, 2013.

[25] Xiaoming Lyu, Qi Han, and Bauke de Vries. Procedural modeling of urban layout: population, land use, and road network. *Transportation Research Procedia*, 25:3337–3346, 2017.

[26] ITS Lab MIT. Mobility of the future.

[27] Rolf Moekel, Klaus Spiekermann, Carsten Schürmann, and Michael Wegener. Microsimulation of land use. *International Journal of Urban Sciences*, 7(1):14–31, 2003.

[28] Kirill Müller. Multilevelipf. https://github.com/krlmlr/MultiLevelIPF, 2017.

[29] Kirill Müller and Kay W Axhausen. Hierarchical ipf: Generating a synthetic population for switzerland. 2011.

[30] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[31] Matthew J Roorda, Eric J Miller, and Khandker MN Habib. Validation of tasha: A 24-h activity scheduling microsimulation model. *Transportation Research Part A: Policy and Practice*, 42(2):360–375, 2008.

[32] Patrick Schirmer, Christof Zöllig, Kirill Müller, Balz Bodenmann, and Kay Axhausen. The zurich case study of urbansim. 2011.

[33] Jose L Silvan-Cardenas, Le Wang, Peter Rogerson, Changshan Wu, Tiantian Feng, and Benjamin D Kamphaus. Assessing fine-spatial-resolution remote sensing for small-area population estimation. *International Journal of Remote Sensing*, 31(21):5605–5634, 2010.

[34] Han Yalie, Youssef Medhat Aboutaleb, Iveel Tsogsuren, Jimi Oke, Carlos Lima Azevedo, Christopher Zegras, Joseph Ferreira, and Moshe Ben-Akiva. Future urban mobility: Urban typologies. Poster presented at Mobility of the Future Sponsors Meeting, MIT, April 2018.

[35] Changping Zhang. An analysis of urban spatial structure using comprehensive prominence of irregular areas. *International Journal of Geographical Information Science*, 22(6):675–686, 2008.