Oregon Statewide Integrated Model

SWIM2 Users Guide v.13

|  |  |
| --- | --- |
| Draft Report  December 2014 | C:\Users\stabler\Desktop\siteLogo.jpg  new_pb_logo.jpg |

Table of Contents

[1. Introduction 4](#_Toc432069333)

[2. Requirements 4](#_Toc432069334)

[3. Installation 4](#_Toc432069335)

[4. File Structure 5](#_Toc432069336)

[5. Creating a New Scenario 6](#_Toc432069337)

[6. Run Documentation 9](#_Toc432069338)

[7. Tracing 10](#_Toc432069339)

[8. Reference Scenario 10](#_Toc432069340)

[9. User Modified Input Files 11](#_Toc432069341)

[10. Visualization Dashboard 13](#_Toc432069342)

[11. Select Link Analysis Module 15](#_Toc432069343)

[Appendix Module-Specific Run Configurations 16](#_Toc432069344)

[Appendix The build\_run.bat Program Specification 21](#_Toc432069345)

[Appendix Model Setup Redesign Description 27](#_Toc432069346)

[Appendix VISUM Network Management Process 30](#_Toc432069347)

[Appendix Zip Matrix Format 35](#_Toc432069348)

[Appendix Select Link Module 36](#_Toc432069349)

[Appendix Master VISUM Output File 37](#_Toc432069350)

# Introduction

The Oregon Department of Transportation’s Statewide Integrated Model (SWIM2) is an integrated land use transport model covering the state of Oregon. The development of the SWIM2 was commissioned by ODOT as part of its Transportation and Land Use Model Improvement Program (TLUMIP) within the larger Oregon Model Improvement Program (OMIP). This development has been undertaken by a series of teams led by Parsons Brinckerhoff, with Hunt Analytics, HBA Specto, and EcoNorthwest.

This guide is intended to support a user in running SWIM2. The model is open source and can be installed on any single computer with the required hardware. This guide is a summary of pertinent information as it applies to this project, as well as other project-specific context. The reader is also advised to consult the SWIM2 Model Description document[[1]](#footnote-1), which included model design, software implementation, development of data parameters/inputs and initial calibration/validation. Additional documentation can be found on the SWIM2 Networks Management process, the Zoning and Land Use inputs, the SWIM2 Visualization Database Tool, and the TS to VISUM migration documentation.

This guide provides an overview of the SWIM2 model, including a summary of input and output files and the directory structure in which they reside. Instructions for using the model, ranging from the initial installation, to scenario creation, and using the visualization tool are also included. The Appendices contain a complete listing of input and output files, module-specific run instructions, and detailed information about network management, running the select link module, and using the zip matrix file format.

# Requirements

The current version of the model requires a computer with a 64-bit Windows operating system. The computer should have 48 gigabytes or more of RAM, and at least a few hundred gigabytes of free space on the model hard drive (a given model run may take up to 100 GB or more of hard drive space when finished). The model also required VISUM 14 or later.

# Installation

To install the model, unzip the provided model template into any location on the computer. This location will be used to hold the model results as well, so this location should be on the hard-disk which has been selected for this purpose. Although it should not matter in theory, it is generally a good idea to place the model in a directory whose full path does not contain any spaces (*e.g.* c:/swim\_model/ instead of c:/swim model/). The model installation contains all necessary executable files for programs associated with running the model, and dynamically adds them as needed to the system path, so the user does not need to do a separate installation for programs[[2]](#footnote-2).

# File Structure

The installed model has a general file structure as shown below (all paths relative to the installation location):

/model/

/census/

/lib/

/java7/

/Python27/

/R-3.1.2/

/[scenario\_name]/

/model/

/config/

/code/

/code/aa/

/code/model\_runner/

/code/sl/

/code/viz/

/code/ct

/inputs/

/parameters/

/t[year]/

/outputs/

/t[year]/

The main /model/ directory contains large files which are too onerous to copy for every scenario. These include the Census ACS PUMS files and the external provided program executables for Java, Python, etc. Though outside of the main scenario directories, it is an essential part of the model distribution and should be included when transferring the model.

Every scenario is contained within its own folder, with a unique name. The folder name is the same as the scenario name. Within the scenario, the /inputs/ folder contains the input files for a model run (*i.e.* those files that are an input to the model *before it has started running*). This includes parameters, template configuration properties, and bootstrap files (these latter files are held in the /t[year]/ directories). The /output/ folder holds the outputs of the model in the appropriate /t[year]/ directories.

The /model/ directory holds the model files used to control the actually running and runtime configuration of the model. The /config/ directory contains the logging and DAF configuration files, and the runtime specification file.

The /code/ directory holds all of the Java, Python, R, *etc.* model code. The /code/aa/ directory holds the code specific to the AA (PECAS) module; /code/ct/ contains that for the commercial transport module; /code/sl/ contains that for the select link module; /code/viz/ contains that for the VIZ database functionality. The /code/model\_runner/ directory holds the code and files used to specify and execute model runs.

# Creating a New Scenario

To create a new model scenario:

1. make a folder in the model installation folder and give it a unique name (the folder’s name will become the scenario’s name); spaces in the name are not recommended.
2. Choose the scenario you want to “base” the new scenario on, and copy the /model/, /inputs/, outputs/t18 and outputs/t19 directories from the source scenario to the new one
3. Copy the build\_run.bat file from the base scenario’s main folder to the new one

You could create a new scenario by copying another scenario’s entire directory and renaming it, then going through and deleting t20+ year output folders, as they will be re-created in the new scenario.

## Scenario-Specific Modifications

If there are code updates or other structural changes to the model, then the files in the scenario’s /model/... directories may need to be changed. In general, the files which are changed to make a scenario unique are contained in the scenario’s /inputs/... directories. The /inputs/parameters/ and /inputs/t[year]/ directories contain various files that can be used to customize a scenario, and the descriptions of these files can be found in SWIM\_file\_descriptions.xlsx.

Also in the scenario’s /inputs/ directory are properties files which are used to specify various things about a model run. The main model properties are specified in the /inputs/globalTemplate.properties file as described in SWIM\_properties.xlsx. This file is “templated” with keywords that are replaced during the model build process and the “detemplified” file is saved in various (module-specific) properties files in the scenario’s /outputs/t[year]/ folders. The available template keywords are as follows:

* **@ROOT.DIR@** - the model install directory
* **@CURRENT.INTERVAL@** - the current “t” year
* **@SCENARIO.NAME@** - the scenario name
* **@SCENARIO.INPUTS@** - the scenario’s inputs directory (relative to the scenario directory)
* **@SCENARIO.OUTPUTS@** - the scenario’s outputs directory (relative to the scenario directory)
* **@[MODULE].LAST.RUN@** - the “t” year that a module last ran (or was initialized for bootstrapping purposes). The available module names are:
  + **SI** – Swim Inputs
  + **NED** - New regional Economics and Demographics
  + **ALD** - Aggregate Land Development
  + **SPG1** - First phase of the Synthetic Population Generator
  + **AA** – Activity Allocation (aka PECAS)
  + **SPG2** - Second phase of the Synthetic Population Generator
  + **PT** - Personal Transport
  + **CT** - Commercial Transport
  + **ET** - External Truck
  + **TA** – Traffic (highway) Assignment
  + **TR** – Transit Assignment
* **@AA.PRIOR.RUN@** - the second-to-last “t” year that the AA module last ran (or was initialized for bootstrapping purposes).
* **@SL.MODE@** - this will detemplify to the name of the SL mode that will be run.
* **@PT.LOGSUMS@** - if PT logsums will be calculated, then this will detemplify to “true”; otherwise it will detemplify to “false”
* **@PT.LDT@** - if PT LDT will be calculated, then this will detemplify to “true”; otherwise it will detemplify to “false”
* **@PT.SDT@** - if PT SDT will be calculated, then this will detemplify to “true”; otherwise it will detemplify to “false”

Additionally, it is sometimes necessary to update a property file at certain points in a model run’s multi-year sequence. To allow for this, property file “updates” may be specified in /inputs/t[year]/globalTemplateUpdate.properties files. In any model run year starting at t[year], the properties in this update file supersede all of those that came before it.

## Running the Model

This section describes the running the SWIM2 model, including run setup/initiation, run monitoring, run analysis, and understanding the run history.

## Initiating a Run

Initiating a run consists of two steps: 1) building the run and 2) starting it. To build a run, the user specifies all of the model steps in the **/[scenario name]/model/config/tsteps.csv** file, and then runs the **/[scenario name]/build\_run.bat** script. This script will create all of the necessary output folders and configuration files, as well as the batch files /[scenario name]/run\_model.bat and /[scenario name]/run\_model\_python.bat. Running the **run\_model.bat** or **run\_model\_python.bat** file will start the model run. Both batch files have identical functionality, only one is purely in batch form and the other runs through a Python layer. The reason both exist is that the former is simpler, but the latter may be needed if certain use cases arise in the future. Further details about the outputs of the build\_run.bat program can be found in build\_run.bat Program Specification Appendix.

The **tsteps.csv** file used to specify a model run is formatted as follows:

The **Year** column specifies the year for the run. The order of the years in this column signify the order that they will be run in the model (so if a year is repeated, or if years are out of order, then they will be repeated or run out of order in the model). The value in the column should be numeric (it should not include the “t” prefix).

All the other columns specify the various modules that are available to run. These modules are (columns must be named the names in bold):

* **SI** – Swim Inputs
* **NED** - New regional Economics and Demographics
* **ALD** - Aggregate Land Development
* **SPG1** - First phase of the Synthetic Population Generator
* **AA** – Activity Allocation (aka PECAS)
* **SPG2** - Second phase of the Synthetic Population Generator
* **PT** - Personal Transport
* **CT** - Commercial Transport (also included External Truck component)
* **TA** – Traffic (highway) Assignment
* **TR** – Transit Assignment
* **PT - Logsums Only** - Personal Transport model running in logsums-only mode
* **SL - Generate Select Link Data** - Select Link generate link data
* **SL - Append Select Link To Trips** - Select Link append select link data to trip lists
* **VIZ** - Create visualization database
* **MICROVIZ** - Create micro-visualization database

In a given year-module cell, one of three values can be specified:

* **[empty]** - (no-op) nothing happens with that module in this year
* **1** - the module will be run in this year
* **i** - the module will not be run in this year, but the model state will be updated such that it will act like the module was run in this year. This is necessary to bootstrap the model. For example, if a model run will start in t20, and bootstrap files will be used for the prior two years, then rows for years 18 and 19 need to be created and populated with “i” in the appropriate module cells.

Below is an example of the file for a run which will use bootstrapped files in year t0, run NED and ALD in t1, and run only NED in t2.

Table 1: Example of tsteps.csv Configuration File

|  |  |  |
| --- | --- | --- |
| **Year** | **NED** | **ALD** |
| 0 | i | i |
| 1 | 1 | 1 |
| 2 | 1 |  |

## Monitoring and Managing a Run

There are multiple ways to monitor a run that has started. The simplest way is to open a Task Manager and see if any Java, Python, and/or R processes are running. Along the same lines, to stop a run, run the /[scenario name]/model/code/stop\_run.bat program, which will end any Java, Python, and R processes. Note that this will end *all* processes running these programs, not just those started by a model run. If the user only wants to end Java processes (to avoid stopping R or Python processes started outside of a model run) the /[scenario name]/model/code/stop\_java.bat program can be run.

While a run is proceeding, the runner program writes out two files which can be used to monitor the run status. The first, **/[scenario name]/model\_report.txt**, writes out a time-stamped entry every time a module has started and finished. This can be used to both monitor the current status of the model as well as determine how long each module took to finish. The following is an example of the contents of a model report:

Mon 02/18/2013 11:15 AM - \*\*\*\*\*Model run started\*\*\*\*\*

Mon 02/18/2013 11:16 AM - Starting NED

Mon 02/18/2013 11:16 AM - Finished NED

Mon 02/18/2013 11:16 AM - Starting ALD

Mon 02/18/2013 11:16 AM - Finished ALD

Mon 02/18/2013 11:16 AM - \*\*\*\*\*Model run finished\*\*\*\*\*

The other file output by the runner program is **/[scenario name]/model\_run\_output.txt**, which captures all of the console output (stdout and stderr) from the programs run by the model runner. Since this will combine many output streams, this file can be difficult to parse; however, if there are any errors with program initialization (such as a Java memory error) this file will be the only place to locate these.

For more fine-grained monitoring of a run, the log files output by the actual module programs can be examined. These log files are all located in the /[scenario name]/ directory and are as follows:

* **main\_event.log** - the main (module) program log file.
* **node0\_event.log** - the log file for the DAF node. The PT module will write to this file.
* **bootstrap\_server\_node0.log** - the log file for the DAF bootstrap server.
* **fileMonitor\_event.log** - the log file for the DAF file monitor.
* **status.log** - a log holding various in-module status messages.

# Run Documentation

## Run History

Inside the scenario folder, there is a file which holds run information on model build setups and runs: **/[scenario name/run\_history.zip**. If this file doesn’t exist, it will be created as needed after a model run has been built or finished executing. This zip file contents are as follows:

21\_02\_2013\_\_04\_53\_47PM

* **/\_settings/[date/time]/** - the build settings for the model built at the date time:
  + **globalTemplate.properties** - the template properties file used when the run was built.
  + **tsteps.csv** - the tsteps file used to define the model run.
  + **model\_run\_batch.bat** - the (batch mode) model runner program.
  + **model\_run\_commands.txt** - the (Python model) model runner program command file.
* **/\_run/[date/time]/** - the run results for the model run which finished at the date time:
  + **model\_report.txt** - the model report file.
  + **model\_run\_output.txt** - the model run output file.

The [date/time] folders in the run\_history.zip file are formatted as follows:

* [day]\_[month]\_[year]\_\_[hour]\_[minute]\_[second][AM/PM]

The run\_history.zip file serves as a documentation of the history of the scenario; once created, it should never be deleted.

## Reproducing a Run

The other purpose that the run\_history.zip file serves is to allow for reproducing a past run. If a previous model run (for a given scenario) needs to be recreated, the globalTemplate.properties and tsteps.csv files can be pulled from the zip file and restored to their original location. Then, a new model run may be built and started to reproduce the previous results. Note that if other inputs (parameters, for example) have changed between the time of the original run and the “reproduction” run, then the resulting run *will not* be identical (because inputs have changed). If this is of concern, then the modeler should take care to version their input files in some manner.

# Tracing

Tracing is a useful diagnostic tool used to track the application of the individual components of the short-distance personal travel (SDT) model across zones, households, and persons. The following property keys are used to set up a trace:

* **trace** – if *true*, then tracing is enabled, if *false*, it is turned off
* **sdt.trace.itazs** – a comma delimited list of origin TAZs to trace
* **sdt.trace.jtazs** – a comma delimited list of destination TAZs to trace
* **sdt.trace.households** – a comma delimited list of integer household IDs (the IDs are from the households.csv output file)
* **sdt.trace.persons** – a comma delimited list of person IDs, where the ID is:

[household ID number]\_[person ID number]

If a trace is turned on, then the various calculations and resulting summaries for the SDT model components will be printed out in the main log file. Note that turning model tracing on will not only add a significant amount of log output, but will also increase the model’s runtime. For this reason, the functionality should generally be used only when attempting to sort out unclear or anomalous model behavior.

# Reference Scenario

SWIM2 policy scenario results are typically compared to a reference scenario, or “business as usual” future. This “reference” scenario is set up to correspond with official economic forecasts, transportation system investments and maintenance assumptions, transport costs, and land use controls.

It is intended that a SWIM2 20-30 year reference scenario should always be available for comparison with policy scenarios. The reference scenario may need to be updated periodically to reflect model upgrades (changes to software, parameters, or inputs), clean-up in the reference networks, a run with or without transit assignment, or other SWIM2 changes, The remainder of this section discusses the preparation of a Reference Scenario.

The SWIM2 model was calibrated for the year 1998 (transport) and 2009 (spatial). When used in production, SWIM2 scenarios begin with the year 2010, and typically run for 20-30 years in the future, including the Reference Scenario. The calibration of the AA model provides inputs and most bootstrapped files necessary to produce the first predicted year of 2010 in production runs. Additionally, bootstrapped travel skims (beta and alpha zone versions) are required in 2009. They are typically obtained after a full 2010 year production run, and copied back to 2009 folders, iterating until differences between 2009 and 2010 predicted values are minimal.

If calibration of the spatial model has produced new files, a process has been put in place to update SWIM2 files from the HBA AA folder structure. First a SVN account for the HBA system must be obtained to share tracked files. Once calibrated files are shared via SVN on the SWIM2 computer, the batch file **hba\_to\_swim2.bat** will copy calibrated 2009 files to create the required inputs and bootstrap files in 2009-2010 folders. Boostrap skims and mode choice logsums are still required by running the full model in 2010, as discussed above.

To run the hba\_to\_swim2.bat program, the user needs to specify the location of the local HBA repository (it is assumed that the scenario holding the batch file will be the recipient of the updates):

[scenario\_name]/model/code/hba\_to\_swim2.bat [path\_to\_hba\_repository]

An example HBA repository path is “C:\models\tlumip\hba\svn\C05”. The location of the HBA repository is: http://svn.hbaspecto.com/svn/pecas/PECASOregon/C05

# User Modified Input Files

It is anticipated that the user will create and run scenarios by modifying selected input files. For this to occur, the new files must be located in the appropriate [scenario name]/inputs/ or [scenario name]/inputs/tn/ directory for the given scenario, where tn represents the first year this file will be used (tn, n years beyond base year). If the filename is changed, then the appropriate property in the [scenario name]/inputs/globalTemplate.propertes file must be edited accordingly.

The following files can be modified and placed in a future year directory for use by the model:

**Macro economic growth** – The NED module is influenced by national forecasts and historic Oregon economic trends that affect the economic growth in the model area. The user can affect modelwide growth projections in future years of a model run, by modifying these ED inputs with absolute.csv or marginal.csv files in the scenario /t0/ directory. These file alter the modelwide ED forecasts, by overriding (absolute.csv) or changing (marginal.csv) the reference NED inputs (found in the scenario /t0/model\_data file).[[3]](#footnote-3)

**Household Attribute Distributions** – SWIM2 produces a synthetic population meeting NED employment by industry as well as user input constraints on workers per household and person age (workersPerHouseholdMarginalxYEAR.csv, oregonPersonsByAgeMarginalxYear.csv). These inputs can be adjusted by the user. Note that significant adjustments to the baseline conditions may lead to incompatibility among the constraints which cannot all be satisfied and/or lead to long runtimes of the SPG module in attempting to meet these conflicting constraints.

**Zoning coverage** – A GIS-based coverage file has been developed that represents the study area zoning coverage. [3] Applying the ‘Tabulate Area’ function in ArcGIS Spatial Analyst to this coverage and the Alpha zone shape file, results in a tabulation of land area in each zoning category in each Alpha zone. The resulting matrix contains a row for each Alpha zone and a column for each zoning category and cell values in land square feet. This forms the ALD input LandSQFTxZoning.csv, which is output by VISUM at the start of the model run. The user can provide different zoning inputs in future years of a model run, by modifying the corresponding zone variables in the master Visum version file in /t0/ or for a specific year, by adding a version file to /tn/ with the different values.

**Vehicle operating parameters** – Vehicle operating parameters, such as vehicle occupancies and capacities and operating costs are inputs to the PT module. The parameter values in the globalTemplate.propertes for these modules can be modified and the file placed in the appropriate /tn/ directory.

**Transport Networks** –Each year’s network is housed in a VISUM version file and link SPEED, CAPACITY, NUMLANES, and COST variables are stored in native VISUM attributes. On the transit side, each line route is tagged with a NET attribute that identifies if the line route is in service in that year’s version file. Running the SI module in each year will trigger the following logic when creating new version files:

SI checks if version file exists in /inputs/t@CURRENT.INTERVAL@/

If exists copy to /outputs/t@CURRENT.INTERVAL@/ and use

If not exists,

For i between 1 and [t@CURRENT.INTERVAL@/](mailto:t@CURRENT.INTERVAL@/)

Check [/inputs/t@CURRENT.INTERVAL@/-i](mailto:/inputs/t@CURRENT.INTERVAL@/-i)

If exists copy to /outputs/t@CURRENT.INTERVAL@/ and use

So, if the user inputs a t0 and a t30 version file in the corresponding inputs/t0 and inputs/t30 directories, and starts a run from t20 to t40, the scenario manager will use the t0 version file from years t20 to t29 and use the t30 version file for runs from t30 to t40. See the VISUM Network Management Appendix for more information on the VISUM network management process.

**Activity Location –** To force activity (industry$ or households) in a particular zone, the user first updates the PI zonal activity targets in VISUM, which are then export to the ActivityConstraintsI.csv file. After export, the user *must run PI in constrained mode* to get new zonal constants (see SWIM\_properties.xlsx). These location constants must be manually updated in the /parameters/ file ActivityZonalValuesI.csv to be used for that and subsequent years.

**Other –** Because some SWIM2 components are micro-simulation models (SPG, CT, and PT), their output varies with each run. This can be controlled to some extent with the specification of random number seed in the [globalTemplate.propertes].

# Visualization Dashboard

ODOT needed a platform for dynamic visualization and inspection of core multi-year SWIM2 model results in order to better understand SWIM2 operations. As a result, the SWIMVIZ database and SWIMVIZ tool was developed.

The SWIMVIZ DB conveniently organizes the core output data for all years of a scenario. Data is organized in a few tables at the Beta zone level. This standardized format facilitates further output processing using the SWIMVIZ tool or other scripting, such as using SQLite and/or R software. The key tables are listed below, with more detail available in the SWIMVIZ DB documentation located in the model/code/viz/documentation folder.

* **ACTIVITYLOCATIONS** – the quantity of activity generated by BZONE, such as industry dollars, household, employment.
* **BUYSELLMATRIX** – the AA module dollar flows between BZONE, by COMMODITY (flow of labor, goods, and services). It also stores the quantity of labor used by activity.
* **EXCHANGERESULTS** – AA BZONE information on the exchange of COMMODITY (goods, services, labor, and floorspace), such as the quantity of demand and supply, prices, etc. It also stores the transport component of the AA utility
* **FLR\_INVENTORY** – ALD BZONE floorspace inventory of space, increment of space change between years, and zoning capacity by COMMODITY (floorspace type).
* **AZONE** – SPG AZONE population, employment and other attributes from SynPop\_Taz\_Summary.csv produced in years the transport models are run.
* **DC\_LOGSUM** – average destination choice logsums from PT-SDT by BZONE, trip PURPOSE, and market SEGMENT.
* **TRIPS\_SDT, Trips\_LDT, Trips\_CT, Trips\_ET** – aggregated trips and trip distances by *trip* origin BZONE from the various transport modules. SDT and LDT are further classified by HHINC ([0-30,30-70,70+]), TRIPPURPOSE, TRIP MODE; CT by TRUCKCLASS and COMMODITY (and includes a column of commodity weight in addition to trips); ET by TRUCKCLASS (EE and import/export within 75 miles), It also stores . There is a column of trips for each time period.
* **TRIPS\_SDT\_Home, TRIPS\_LDT\_Home, TRIPS\_CT\_Home** – same as above, except trips are aggregated by person trip household *home* BZONE origin and *truck tour* origin, rather than *trip* origin. LDT includes non-commute trips over 50 miles as well.
* **TRIPMATRIX** – combined trip matrices for each BZONE OD pair, time period ([“am”,”md”,”pm”,”nt”]), and mode from SDT, LDT, CT, and ET aggregated to common modes/truck classes. Note that the SR2 and SR3P modes are vehicle trips not person trips.
* **LINK\_DATA** – VISUM link assignment results by TSysSet (volumes, travel time, cost, etc.) and key link attributes (free flow speed, capacities, length, type, lanes, azone) for each time period.
* **SKIM** – Travel distance, time, tolls for each BZONE OD pair for peak and offpeak, auto and truck. Collapsed from BZONEs by a weighted average of drive alone or truck trips by OD for the same time period.
* **BOARDINGS** –Transit assignment boardings by line route, access mode, time period and year.
* **MODELWIDE** – various modelwide data typically associated with the NED module (AA composite utilities, annual construction dollar budget).
* Other tables: **ALLZONES**, **BZONE**, **TSTEP**

To automatically create a SWIMVIZ DB from a completed SWIM2 scenario run, the user will edit the configuration file to include a ‘1’ in the column for Viz, in the row of the final year of the model run. This runs two R/SQLite scripts: **build\_Viz\_DB.R** (creates DB for each year), and **all\_Viz\_DBs.R** (runs the prior script each year, merging into one database with a table for each year). A zipped version of the multi-year SWIMVIZ DB is copied to the ODOT FTP site facilitating remote users in obtaining this data.

The SWIMVIZ Micro database houses the synthetic population (persons and households from SPG) as well as detailed tour and trip information for person (from PT) and internal trucks (from CT). The key tables are listed below with more detail available in the SWIMVIZ micro DB documentation.

* **HH** – represents all the households in the model and their key attributes, including description of any long distance tours.
* **PER** – represents all the persons in the model and their key attributes, including industry, work status, and descriptions of the modeled long distance and short distance daily tour pattern by purpose.
* **TOUR\_LDT\_MICRO** – represents all the long distance (LDT) tours in the model, including their purpose, mode, origin and destination zones, times, and party size.
* **TRIP\_LDT\_MICRO** – represents all the LDT vehicle trips in the model, including their purpose, mode, origin and destination zones, and times.
* **TOUR\_SDT\_MICRO** – represents all the short distance (SDT) tours in the model, including their purpose, mode, origin and destination zones, and times.
* **TRIP\_SDT\_MICRO** – represents all the SDT person trips in the model, including their purpose, mode, origin and destination zones, and times.
* **TRIP\_CT\_MICRO** – represents all but the through truck trips (CT, not ET) in the model, including commodity, carrier type, weight, origin and destination zones, and times.

To use the SWIMVIZ, the user either:

* installs the Adobe AIR Runtime (<http://get.adobe.com/air>) program and the SWIMVIZ AIR application on their local machine and has a local copy of a SWIMVIZ DB or
* using remote desktop to access the SWIMVIZ tool on one of the ODOT machines, which have the SWIMVIZ databases.

SWIMVIZ reads the SWIMVIZ database and allows the user to dynamically query, plot, and map data from the DB. On startup, it reads the Settings.xml, and DefaultQueries.xm (or PostgresDefaultQueries.xml for the PostgreSQL setup) files. See the SWIMVIZ User’s Guide (/model/code/viz/documentation/swim\_viz\_userguide.doc) for more information.

Once a SWIMVIZ SQLite database is created, it can be exported to PostgreSQL (v. 8.3) database format. The SWIMVIZtoPOSTGRES.R script (in model/code/viz) generates a SQL script and CSV data files, with customizable variables at the top of the script. The R-generated bulk insert SQL script may be run on the PostgreSQL database to load the data from the associated CSV files.

# Select Link Analysis Module

The Select Link (SL) Module was created for appending route information to microsimulated SWIM2 trips in order to create subarea external matrices for MPO models. Complete step-by-step instructions for creating the SL inputs and processing SL outputs are contained in Appendix IX. The only input to the Select Link module is located in the [scenario name]\inputs\t0 directory and is **selectLinks.csv**. This file is a mapping of SWIM network links to subarea network zones. For each link in the file, a select link analysis is run on the saved VISUM assignment paths. The outputs from the process are stored in the [scenario name]\outputs\tn directory and are:

* sl\_subarea\_demand\_matrix.csv - Volumes by time period and purpose for all origins and destinations (SWIM2 zone numbers) in the subarea
* selectLinkResults.csv - Full mapping of select links to model station numbers.

The assignment paths required for Select Link are generated during the SWIM TA module. If TA was not run, then the user will need to re-run the TA module in the desired year, and edit the tsteps.csv configuration file to include a ‘1’ in the SL column for the years that Select Link should run.

# Appendix Module-Specific Run Configurations

In order to consolidate and clarify the core tasks that are run for each module, a new Java class has been created to replace the original ApplicationOrchestrator class ModelEntry. This class has instance members for every SWIM module (SI, NED, ALD, etc). The ModelEntry class has a main method through which a module is actually run; the syntax for the main method call is:

java [...] com.pb.tlumip.ao.ModelEntry [key1=parameter1] [key2=parameter2] …

All modules require the property file specific to the module be specified using the key “property\_file”; additionally, the VIZ and MICROVIZ modules require the years to build the database for as a comma-separated lists using the key “viz\_years”.

Specific details for each module are discussed in the remainder of this appendix.

## SI

The SI module is run from the same Python program used to run traffic and transit assignment - SWIM\_VISUM\_Main.py, which is located in /[scenario\_name]/model/code/Visum folder. This module simply copies the base VISUM version file to a local copy in /[scenario\_name]/outputs/tXX/, and then extracts year-specific model inputs from the version file into files used by other modules.

## NED

The NED module is a Python program located in /[scenario\_name]/model/code/NED.py. The ModelEntry NED instance calls the com.pb.tlumip.ed.NEDModel class, which initiates the Python program as a process, redirects its console output to the appropriate loggers, and indicates if the module finished normally or not. The specifics of how the process commands are formed come from the following property file keys:

* **python.executable** - the path to the Python program
* **ned.python.command** - the path to the NED.py program file
* **ned.property** - the path to the NED-specific property file

## ALD

The ALD module is an R program located in /[scenario\_name]/model/code/ and consists of the following programs:

* ALD.R
* ALD\_Inputs.R
* ALD\_Functions.R

The ModelEntry ALD instance calls the com.pb.tlumip.ald.ALDModel class, which initiates the R program as a process, redirects its console output to the appropriate loggers, and indicates if the module finished normally or not. The specifics of how the process commands are formed come from the following property file keys:

* **r.executable** - the path to the R program
* **ald.codePath** - the path to the directory holding the ALD R program files
* **ald.property** - the path to the ALD-specific property file
* **ald.filePath** - the path to the base output directory (not including the “t” year directory)
* **nameOfRCode** - the name of the main ALD program (ALD.R)

## SPG1

The SPG1 module is called via the com.pb.tlumip.spg.SPGnew class’s spg1 method. This method is actually called *twice*, once without the person-age constraint enabled to get an estimate for the total regional population (the module requires households be forecast, but not the actual population), and then a second time with the person-age constraint using the estimated population from the first run.

## AA

The AA module is unique in that even though it is Java, the ModelEntry class actually creates a new process to run it, rather than calling it directly. This is because the AA code uses classes derived from those used by the ModelEntry (and other module) programs, but which have been modified by HBA to suit the AA module. Because these modifications may have altered the functionality of those classes, but the namespaces remain the same, the ModelEntry program and AA program must be kept completely separate.

Thus, the ModelEntry AA instance calls the com.pb.tlumip.aa.AAModel class, which initiates the AA Java program as a process, specifying the appropriate logging configuration to use, and indicates if the module finished normally or not. The specifics of how the process commands are formed come from the following property file keys:

* **aa.command.java** - the location of the Java program
* **aa.command.max.heap.size** - the maximum Java memory size for the AA program
* **aa.command.log4j.config.file** - the location of the log4j configuration used by the AA module
* **aa.command.classpath** - the classpath for the AA module
* **aa.command.class** - the qualified classname of the entry class for the AA module

As a note, the jar files holding all of the Java code used by AA are held in the /[scenario\_name]/model/code/aa/ directory.

## SPG2

The SPG2 module is called via the com.pb.tlumip.spg.SPGnew class’s spg2 method.

## PT

The PT module is a distributed application framework (DAF) application which must be started indirectly by writing a series of commands into the DAF file monitor’s command file. This is achieved via the com.pb.tlumip.ao.StartDafApplication class, which writes a series of commands to the command file to start the DAF node (only one is used in the current setup), start the DAF cluster, and then start the PT application. Then the program waits for an appropriate “done” file to be created which indicates the module has finished. After each DAF initialization step, the program must wait in order to allow for the DAF framework to fully initialize; additionally, the program will pause for a certain length of time between checking to see if the “done” file has been created or not. These wait times can be adjusted via the following properties:

* daf.start.node.sleep.time
* daf.start.cluster.sleep.time
* daf.done.file.check.sleep.time

## CT

The CT module is called via the com.pb.tlumip.ct.CTModel class’s startModel method, which calls an R program specified by the following property:

* ct.rcode

The code then executes the remainder of the R scripts in the /[scenario\_name]/model/code/ct/ directory. The CT code also creates the External Truck trip tables (currently as a placeholder only).

## TA

The TA module is used to extract demand information (by time of day and mode) and to run the traffic (highway) assignment and skimming procedures. The actual module is the SWIM\_VISUM\_Main.py script located in the [scenario\_name]/model/code/Visum folder. The program opens the VISUM version file stored in the year output folder, prepares the assignments, runs the assignment, and writes out assignment results and skims. The multi-class assignment is run for auto and truck classes and is either run twice – once for peak and once for offpeak, or for the entire day (FULL) – this includes peak (PK), off-peak (OP), afternoon (PM), night (NT) and DAILY assignments.

## TR

The TR module is used to run the transit assignment and skimming procedures. The actual module is the SWIM\_VISUM\_Main.py script located in the [scenario\_name]/model/code/Visum folder. The program opens the VISUM version file stored in the year output folder, prepares the assignments, runs the assignment, and writes out assignment results and skims. The assignment is run for both short and long distance trips for both the peak and offpeak time periods.

## SL

The SL module is called via the com.pb.tlumip.sl.SelectLink class’s runStages method. This method takes the stage to run as its parameter, of which the following stages are available (the stage “key” used as a parameter to the main call are listed in parentheses; this key is specified using the sl.mode property):

* **generate select link data** (“a” or “d”)
* **append select link to trips** (“a” or “d”)

## VIZ

The VIZ module creates the visualization database and operates through the com.pb.tlumip.ao.VizDbBuilder class. The following R programs are used to build the visualization database and are located in the /[scenario\_name]/model/code/viz/ directory:

* build\_Viz\_DB.R - builds the visualization database for a single year
* all\_Viz\_DB.R - combines the individual databases into a single database

The build\_Viz\_DB.R program builds the visualization database for a single year, and the all\_Viz\_DB.R program combines the individual databases into a single database. Because of memory and processing constraints, all of a given model run’s individual year databases cannot be created in parallel; “full” model year database builds (those which include the transport modules) take up a large amount of memory, and overall the number of parallel database builds should not exceed the number of processors (though it may be lower due to memory limitations). There is no general algorithmic way to determine the best prescription for a given system, so the modeler may need to adjust this via the following properties:

* **viz.max.concurrent.ts** - the maximum number of concurrent transport year database builds
* **viz.max.concurrent.total** - the maximum number of *total* year database builds (transport plus non-transport)

The defaults for these (2 and 8) are a good place to start, but the modeler is encouraged to modify them to either increase performance or rectify any out-of-memory or memory/processing bottlenecks that may arise on a given computer setup.

The actual command used to build the databases is specified through the following properties:

* **t.year.prefix** - the prefix of the folders holding the outputs for a given year (“t”)
* **scenario.outputs** - the path to the base output directory (not including the “t” year directory)
* **viz.is.ts.year.indicator.file** - the name of the file whose presence is used to indicate whether or not a given year is a transport year or not
* **viz.build.command.list** - the command list for building the individual year database
* **viz.combine.command.list** - the command list for combining the years’s databases

The “command.list” properties use a special format:

[arg1,arg2,...]

where argN is the Nth argument in the command. If an argument is surrounded by pipes (*i.e.* |arg1|) then that argument is assumed to be a property key, and is filled in with that property key’s value.

Additionally, the following properties can be used to perform some post-build operations, if the database build process finished correctly:

* **viz.delete.subyear.dbs** - if “true”, then when the database building process is finished, the individual year databases will be deleted
* **viz.zip.final.db** - if “true”, then the final database will be compressed into a zip file when the process is finished
* **viz.subyear.dbs.wildcard** - the (wildcard) path to the individual year databases used to delete them when viz.delete.subyear.dbs is true
* **viz.final.db** - the path to the final viz database
* **viz.zip.file** - the path to the zipped database (if used)

## MICROVIZ

The MICROVIZ module runs identically to the VIZ module, only with different keys for some of the properties used to control the database creation and post-build operations:

* **microviz.build.command.list** - replaces viz.build.command.list
* **microviz.combine.command.list** - replacesviz.combine.command.list
* **viz.micro.delete.subyear.dbs** - replaces viz.delete.subyear.dbs
* **viz.micro.zip.final.db** - replaces viz.zip.final.db
* **viz.micro.subyear.dbs.wildcard** - replaces viz.subyear.dbs.wildcard
* **viz.micro.final.db** - replaces viz.final.db
* **viz.micro.zip.file** - replaces viz.zip.file

# Appendix The build\_run.bat Program Specification

The /[scenario\_name**]/build\_run.bat** program is used to build a model run for a scenario. It creates a number of configuration files specific to the scenario specifications, as well as programs that can be called to run the model. This appendix outlines the specifications of the build\_run.bat file, including its outputs and intentions.

The actual build\_run.bat program actually just calls a Python program /[scenario\_name]/model/code/model\_runner/**build\_run.py**. This program performs the following tasks:

1. Reads the **tsteps.csv** file and builds an ordered year-module mapping for the model run specification.
2. Using the model run specification, builds a key-value mapping for detokenizing the properties file for each year-module step.
3. Create output folders for the run, if necessary.
4. Detemplify the properties file for each year-module step and save it (with a module-specific name) into the appropriate output folder.
5. Build general logging and DAF configuration files specific to the run and computer configuration.
6. Build the batch file which can be used to directly run the model as well as the command file used to run the model through a Python layer.
7. Save the settings in the scenario history file.
8. Create shortcuts in the main scenario directory to run the model.

These steps are described in more detail in the rest of this appendix.

### Step 1: Build Year-Module Mapping

This step builds an internal year-module mapping based on the tsteps.csv file. Details of this file’s structure are presented in the Running a Model section earlier. A few points are worth noting, however:

* The year-module mapping is ordered according to the order of the rows and columns in the tsteps file. Specifically, years are run in the top-to-bottom ordering of the Year column in the file, and modules are run in the left-to-right ordering of the module columns. To create non-standard orderings (repeated years, out-of-order modules) the user can create rows with identical Year values to form the necessary structure. For example, the following table would run the following Year-module pairs, in this order: [20-NED, 21-ALD, 21-NED, 21-NED]:

Table 2: Tsteps File Setup

|  |  |  |
| --- | --- | --- |
| **Year** | **NED** | **ALD** |
| 20 | 1 |  |
| 21 |  | 1 |
| 21 | 1 |  |
| 21 | 1 |  |

* The initialization module identifier (“i”) is processed as such: act as though the module ran at this time in this year, but do not actually run the module. The practical application for this is for bootstrapping model runs, but it is also possible to “dry-run” a module by using this identifier mid-stream in a model run.

### Step 2: Build Detokenizing Map

This step builds a mapping for each year-module pair that is used to detokenize the property file to form a year-module specific property. This detokenizing mapping will contain an appropriate value for all keys listed in the Creating a New Scenario - Scenario-Specific Modifications section earlier.

### Step 3: Create Output Folders

This step will create any output folders needed by the run that don’t already exist. Note that if an output folder already exists and contains files (such as from a previous run), those files *will not* be deleted.

### Step 4: Detemplify the Properties File

This step creates year-module specific properties files for the model run; these files are saved in /[scenario\_name]/outputs/t[year]/[module\_name][cycle].properties, where [cycle] only exists if the module was run in this year in a previous step. The key-value mappings specific to the year-module pair determined in Step 2 are used to detokenize the template property files. The template property file is located in /[scenario\_name]/inputs/globalTemplate.properties, and, depending on the year run, will have update properties applied.

### Step 5: Build Configuration Files

This step builds the general model configuration files used for the model run. The logging configuration files specify the log4j settings for all of the Java programs and are stored in the /[scenario\_name]/model/config/ directory:

* **info\_log4j.xml** - for the main model stream
* **info\_log4j\_aa.xml** - for the AA module
* **info\_log4j\_fileMmonitor.xml** - for the DAF file monitor
* **info\_log4j\_node0.xml** - for the DAF node0 stream

All of these configuration files specify a console logger and a file logger, the latter of which is written out to the main scenario directory: /[scenario\_name]/. The main logging configuration also includes a status logger, which is used for temporary status logging written out to a status log file /[scenario\_name]/status.log.

The other configuration files written out by this step are the DAF configuration files, placed in the /[scenario\_name]/model/config/ directory:

* **daf.properties** - general DAF properties
* **startnode0.properties** - the starting command for DAF modules using a file monitor (PT)
* **ptdaf.properties** - the DAF program settings for the PT module

The following property keys from the /[scenario\_name]/inputs/global\_template.properties file are used when building these configuration files:

* **daf.memory** - the JVM memory size for DAF modules using startnode0.properties
* **daf.message.port** - the general DAF messaging port
* **daf.admin.port** - the port used for DAF administration messages
* **tsdaf.message.port** - the port used for DAF messages used with the TS module
* **daf.admin.server.port** - the port used for DAF server administration messages
* **cpu.factor** - used to determine the parallelism of the PT module

The memory settings may need to be modified from the defaults depending on the available resources of the modeler’s computer. If any of the ports listed in the properties file are in use or blocked (by, for example, a firewall or OS policy) then they may need to be changed.

The ptdaf.properties file specifies the manner in which the PT module is multi-threaded: how many concurrent threads are created to process the module’s tasks. In theory, more threads are “better” as they should provide increased processing throughput; however, too many threads may cause the program to run into memory limitations or other unforeseen processing bottlenecks. The cpu.factor property is used to control number of threads that are specified for the PT module. Specifically, the number of threads is roughly defined by the relationship:

threads = MAX(1,ROUND(CPU\_COUNT\*cpu.factor))

where CPU\_COUNT is the number of cores listed as available by the operating system. In other words, at least one thread is always created, but at most CPU\_COUNT\*cpu.factor will be created. The default cpu.factor is 0.8, though for a given system varying this number may provide improved performance of the PT module.

### Step 6: Build Program Files

This step builds the files that are used to actually run the model. There are two versions that are both built: one that is a batch program that is run directly, and the other which is a command specification which is run by a Python wrapper. As noted previously, the reason for both of these being created is to allow for future flexibility (via the slightly more complicated Python layer) if needed, but to prefer the simpler batch version for the moment.

Both versions specify the Java (and other) program calls used to run the various modules. The specification for what list of programs are created for each module are found in the /[scenario\_name]/model/code/model\_runner/module\_commands.py Python program file. In addition to a number of helper methods (*e.g.* to normalize paths that are safe to use in a batch file), there is a ModuleCommands class which has a number of methods, each of which return a list of batch commands which are to be run for the module/step indicated by the method.

According the specifications module\_command.py, most modules just call the main com.pb.tlumip.ao.ModelEntry entry class (see the Module Specific Run Configurations Appendix) with arguments specific to the year-module pair. However, in some cases, a module may perform some additional actions; for example, the PT module specification also, after the main module has finished, zips up the /[scenario\_name]/outputs/t[year]/debug/ folder (if it exists) that the PT module may have created.

Additionally, it is noted that a number of commands are available which are not specific to the modules, but serve to support the overall model run. The currently available commands are as follows:

* **conditional** - run a command only if some condition is true
* **conditionalIfExists** - run a command only if a file exists
* **conditionalIfNotExists** - run a command only if a file does not exist
* **deleteDirectory** - delete a directory (if it exists)
* **deleteFile** - delete a file (if it exists)
* **makeDirectory** - make a directory (if it doesn’t exist)
* **copy** - make a copy of a file
* **zip** - compress a file or directory into a zip file
* **runPause** - pause the run for a specified time period
* **runFileMonitor** - start the DAF file monitor
* **runStopFileMonitor** - stop the DAF file monitor
* **runBootstrapServer** - start the DAF bootstrap server
* **runStartBootstrapClient** - start the DAF bootstrap client
* **runStopBootstrapClient** - stop the DAF bootstrap client

##### Batch Model Runner

The batch program is saved in /[scenario\_name]/model/code/model\_runner/ **model\_run\_batch.bat**. This batch program is essentially a list of the commands specified by the module\_command.py file; there is some additional “wrapping” code which allows for cleanup if an error occurs. Also, there are some simple “logging” calls which note the general model progress in the /[scenario name]/model\_report.txt file. This logging is facilitated by the /[scenario\_name]/model/code/report.bat program, which simply inserts a date-time to a message.

##### Python Model Runner

The Python model runner uses a Python program, /[scenario\_name]/model/code/model\_runner/**run\_model.py**, to run the model. This program reads a command specification file (/[scenario\_name]/model/code/model\_runner/model\_run\_commands.txt) and runs the commands listed in it. This command specification file is similar to a batch file, except that all commands are in the form of a Python list (each argument is an element), and all commands are prepended with one or more control statements:

[error] [log/call/start] [command list]

If the model runs into an error, then only subsequent commands with the “error” statement will be executed; otherwise, only those commands without the “error” statement are executed. The other command statements do the following:

* **log** - prints a message to the /[scenario name]/model\_report.txt file
* **start** - starts a process but does not wait for it to finish before continuing
* **call** - starts a process and waits for it to finish; if it returns an error, then the model run is put into an “error” state

As an example of how the programs are related, the following two examples present identical model run specifications for running NED in a single year, the first for the batch program and the second for the Python wrapper program’s command file (paths have been truncated for brevity).

**model\_run\_batch.bat**

@ECHO OFF

echo. 2>"[scenario\_path]\model\_report.txt"

cmd /C ""[scenario\_path]\model\code\report.bat" \*\*\*\*\*Model run started\*\*\*\*\*>>"[scenario\_path]\model\_report.txt" 2>&1"

cmd /C ""[scenario\_path]\model\code\report.bat" Starting NED>>"[scenario\_path]\model\_report.txt" 2>&1"

[java\_program] [java\_parameters] com.pb.tlumip.ao.ModelEntry NED "property\_file=[scenario\_path]/outputs/t20/ned.properties"

IF %ERRORLEVEL% NEQ 0 GOTO MODEL\_ERROR

cmd /C ""[scenario\_path]\model\code\report.bat" Finished NED>>"[scenario\_path]\model\_report.txt" 2>&1"

cmd /C ""[scenario\_path]\model\code\report.bat" \*\*\*\*\*Model run finished\*\*\*\*\*>>"[scenario\_path]\model\_report.txt" 2>&1"

GOTO END

:MODEL\_ERROR

cmd /C ""[scenario\_path]\model\code\report.bat" \*\*\*\*\*Model run error (finished abnormally)\*\*\*\*\*>>"[scenario\_path]\model\_report.txt" 2>&1"

:END

CALL "[scenario\_path]\model\code\stop\_run.bat"

**model\_run\_commands.txt**

log ['\*\*\*\*\*Model run started\*\*\*\*\*']

log ['Starting NED']

call [[java\_program], [java\_parameters], 'com.pb.tlumip.ao.ModelEntry', 'NED', 'property\_file=[scenario\_path]/outputs/t20/ned.properties']

log ['Finished NED']

log ['\*\*\*\*\*Model run finished\*\*\*\*\*']

error log ['\*\*\*\*\*Model run error (finished abnormally)\*\*\*\*\*']

error call ['CALL', '[scenario\_path]\\model\\code\\stop\_run.bat']

call ['CALL', '[scenario\_path]\\model\\code\\stop\_run.bat

### Step 7: Save Settings

This step saves the configuration settings specific to this run of build\_run.bat in /[scenario name/**run\_history.zip**. The contents of this history file are described in more detail in the Running the Model section earlier in this document.

### Step 8: Create Shortcuts

This step just creates the shortcut files /[scenario\_name]/run\_model.bat and /[scenario\_name]/run\_model\_python.bat, which are used to run the model. The former program will first call /[scenario\_name]/model/code/model\_runner/model\_run\_batch.bat, and then call a program (discussed below) to save the model results in the scenario history file. The latter program first calls the Python runner program (/[scenario\_name]/model/code/model\_runner/run\_model.py) to run the commands in /[scenario\_name]/model/code/model\_runner/model\_run\_commands.txt, and then calls the model results saving program.

The model results saving program is /[scenario\_name]/model/code/model\_runner/ **save\_run\_history.bat**, and it saves model run results in /[scenario name/run\_history.zip.

# 

# Appendix Model Setup Redesign Description

This appendix describes the 2013 model setup redesign.

The old SWIM model implementation has evolved organically over many years and consists primarily of the following major components:

* Actual module components written primarily in Java, though R, Python, and EMME are used as well.
* Distributed Application Framework (DAF) Java code used to allow certain modules to run concurrently (multithreaded) within and across many computers in a networked cluster.
* Application Orchestrator Java code used as the entry point for the various model modules, as well as a configuration specifier given the current model state.
* Python Ant Substitute (PANTS) code used to specify and execute full model runs. This included not only running the various modules, but also (rudimentary) control over the DAF and logging specifications.
* Remote Application and Cluster Control (Artax) Python programs used to communicate with a cluster in order to stop, start, monitor model runs. The programs facilitated communication both between the computers in a cluster as well as between the cluster and an external controller.
* The Model Run GUI (MrsGUI) Python user interface used (locally or remotely) to communicate with and control the cluster and model runs (via communication with the Artax server).

The first two items contain the core of the actual model program, and the remaining items are concerned with controlling and monitoring the “running” of the model. Due to a host of factors, these latter items have become more complex and bloated over time, and it was felt that they could be streamlined greatly without losing any key functionality.

Additionally, with the availability of more powerful multi-core computers (which ODOT has recently acquired for model runs), it was decided that model runs would no longer be performed across a networked cluster, but rather on a single machine. This decision further simplified the requirements of the model “controller” in the following ways:

* The need to monitor (and maintain the uptime) of a set of cluster machines is no longer necessary. A single computer is the sole source of success (or failure) of a run.
* A consistently available network is no longer required. This requirement was difficult not only for interprocess communication, but also for accessing a shared drive for model inputs and outputs. Maintaining this requirement was especially difficult when cluster computers were not under the modeler’s control, and network downtimes or hiccups would occur due to various “IT management” issues.
* Balancing workload across non-homogenous systems is no longer an issue.

With this in mind, a complete rewrite of the model file structure and execution framework was developed. Using previous experience (good and bad) as well as the current situational requirements, the following goals were used to guide the effort:

1. The model will run on a single machine. Many previous problems and mis-steps would have been avoided had the “networked cluster” aspect of the model been removed from the runtime equation. While there may be a runtime penalty as the model work cannot be spread as widely as before, the operational runtime gains will probably be significant enough the more than make up for this.
2. The specifics of a given (multi-year/multi-module) run will be simply specified in an easy to understand external document. This will allow the modeler to (more easily) change a specific model run without having to understand the esoterica of various frameworks or model code. Furthermore, this will allow another person to more easily follow a given model scenario’s runtime history.
3. A scenario will be (within reason) completely self-contained. This means that each scenario will, in effect, contain an entire copy of what is needed to run the model. This serves three primary purposes:
   1. Encapsulation: changes in one scenario will not affect another.
   2. Version control: the specific “state” of the model (code, parameters, etc.) is captured and maintained within the scenario. Updates must be intentionally applied (on a scenario-by-scenario basis), which makes analysis and reproducibility much easier. (Note: The old model included a file version control system which, while done with the best intentions, wrecked havoc when any kind of debugging or multi-run situations arose; this mess was one of the primary motivating factors in this update.)
   3. Portability: the model can be transferred to another computer (or user) by, essentially, copying the scenario folder contents to the new machine.
4. Once a model run is specified, its configuration files will be completely specified. That is, a run will not need to dynamically change its configuration files (properties files, DAF specs, *etc.*) in order to specify a given model state. Instead, the model configurations will have the expected state (at each step) “baked into” the configurations themselves. While not only being more explicit than the old model, this will also avoid the specific issue that the old model had with repeatedly running modules (in the same year) during debugging efforts (specifically: AO would dynamically modify the properties file in such a way that the module would use the incorrect input files for repeated runs).
5. The model runtime would automatically configure itself (again, within reason) with respect to the computer it is running on. For the most part, this meant dynamic specification of the DAF configurations to scale the model to the number of available processors so as to maximize the resource utilization.

This new structure is somewhat different than that used in the old model. In order to clarify the relationship between the two model structures, the following matrix shows how the various file locations are related. An “X” in a cell indicates that files from the old model locations (rows) were moved to those in the new model location (columns):

Table : Mapping of Old Structure to New

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | /model/ | /[scenario\_name]/ | | | | |
| /census/ | /model/ | | /inputs/ | | /outputs/ |
| /config/ | /code/ | /parameters/ | /t[year]/ | /t[year]/ |
| /[scen\_name]/ | /t[year] | |  |  |  |  |  | X |
| /user\_inputs/ | /[scen\_name]/ | /t[year]/ |  |  |  |  | X |  |
| /model/ | /census/ | | X |  |  |  |  |  |
| /java\_files/ | |  | X | X |  |  |  |
| /parameters/ | |  |  |  | X |  |  |

In the effort to update the model, all attempts were made to ease the transition process between the new and old systems. This meant that rather than rewriting *every* component of the existing system, as few changes as possible were made to reach the goals outlined above. In the new structure, the following core elements were able to be retained:

* The module programs were left as-is (though the code used to actually call them - via AO and PANTS - was changed).
* The DAF framework was preserved, though now it is used purely to multi-thread the applications (its distributed functionality has effectively been removed by restricting it to a single node).

With respect to the (scenario) file structure, although there have been some significant changes to the folder structure, the actual contents of the folders have transitioned in a simple one-to-one fashion, which should allow moving files from the old model to the new one fairly straightforward.

Because the new model is both very easy to “install” and simple in its runtime specifications, transitional training should be minimal and (hopefully) the end users will find the new setup much easier to “wrap their head around.” Additionally, the new setup’s portability and static runtime configurations should make debugging and failure analysis (both by ODOT and PB) easier and more transparent.

# Appendix VISUM Network Management Process

This appendix describes the process an ODOT employee uses for editing the SWIM model network. VISUM 14, license size Large, is used to manage all highway and transit networks for all years. Some working knowledge of VISUM is assumed by the user who will manage the networks.

## Network Setup

The Nodes and links for the current year is maintained in the version file.

The VISUM link attributes that must be populated for use in SWIM are in the table below.

Table 5: Link Attributes

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| CapPrT | Capacity for this year’s version file |
| LANES | Number of lanes for this year’s version file |
| SPEED | Speed for this year’s version file |
| COST\_A | Link cost for auto |
| COST\_D | Link cost for truck |
| VDF | Link VDF code |
| TSysSet | Modes of transportation allowed on link (transport system set) |

There are a few important network definition conventions:

1. A link in VISUM is always defined for both directions (from A to B and B to A). However, that does not mean the link is available (or open) in both directions. Whether or not the link is open is a function of the TSYSET
2. The TSYSSET (modes) attribute defines the available modes on the link.
3. The attribute Sum:LineRoutes\NET identifies how many transit lines traverse the link. This attribute is useful for checking if the link is required by the transit network.
4. Centroid nodes and connector links are separate network objects in VISUM; they are ZONES and CONNECTORS respectively.
5. To specify if a transit line is available for a given year, set the LineRoute NET attribute to 1. Set it to 0 if the transit line is not available in the year.
6. Auto traffic assignment is done on a network with the capacity (CapPrT) calculated as follows: CapPRT = (1/VOL\_FACTOR) \* TEMP\_CAPACITY \* NUMLANES, where

VOL\_FACTOR = (max trip start VMT hour of the day) / (trip start VMT in time period)

VOL\_FACTOR values are calculated and output in outputs/t[year]/sdtTODTrips.csv.

## Checking the Highway Network

It is important to check the network connectivity for the year of interest before running the model. To check the network for a specific year, do the following:

1. Create and execute a procedure file to copy year specific attributes to VISUM standard attributes (capacity, speed, and number of lanes) in order to check the network
2. Calculate + Network Check to run network checks:

Table 6: Functions Used to Check Highway Network

|  |  |
| --- | --- |
| **Network Check** | **Solutions** |
| Isolated Nodes | Review and remove all isolated nodes |
| Zones w/o PrT connectors | Ensures that all zones have access to the network |
| Network consistency PrT | This is the most important check since it checks all OD pairs are connected. Only check modes a (auto) and d (truck) |
| Dead-end roads PrT | Likely roads that should not be dead ends; repair as needed |
| Links without succeeding link | Review each mode and repair as needed |
| Link with capacity=0 or v0=0 | Ignore rail & air (l, m, n, r) where capacity=0 (Only select modes “a” and “d” when checking capacity) |

1. Correct any network connectivity issues
2. Repeat this process until the network is clean and ready for use

## Highway Editing

There are various types of highway edits that can be done in VISUM. The most common are the following:

1. Add a node:
   1. Set Node as the current network type
   2. Switch the mode to insert mode
   3. Click on the map to add the node.
   4. The node number should be set at the time the node is created.
   5. All nodes for all years are managed in the master network file.
2. Move a node:
   1. Set Node as the current network type
   2. Switch the mode to the edit mode
   3. Single click a node, and then click and drag to move the node.
   4. There are various node editing helper functions in VISUM – such as merging and shifting nodes and splitting links by an existing node.
3. Add a link:
   1. Set Link as the current network type
   2. Switch the mode to insert mode
   3. Click the FROMNODE and TONODE to add the link
   4. Edit the required link attributes via the Edit Link window, Quick View panel, or the link listing.
   5. The link lengths are calculated automatically based on the link shape and the network projection (OGIC).
4. Modify a link:
   1. There are various link editing helper functions in VISUM – such as splitting links, deleting a node and merging links, reshaping links, copying attributes in the reverse direction, setting attributes of multiple links at once, etc.
   2. A useful function for selecting multiple links at once is the Link Sequence from Shortest Path Search option available by right clicking on “Links” in the network types. This function allows you to select links based on shortest path search. This function is especially useful for opening up all links for the TSYS of a new transit line. In addition, multiple links can be selected by holding down the Ctrl key to add an additional link to the already existing link selection set.
5. Check the network connectivity - Errors are written out to ‘ERROR.TXT’
   1. Calculate > Check Network

## Transit Edits

The definition of transit lines for all transit modes is split across multiple network objects in VISUM. The objects are stop points, lines, line routes, time profiles. These network objects are:

* Stop point – the actual physical point where the line stops
* Line – a simple name or label for a line such as the “#5”
* Line route – the node sequence for a line
* Time profile – the time it takes to traverse the line, including run time, dwell time, is boarding possible, is alighting possible, and headway.

The VISUM link TSYSSET (transport system set) field determines if the link is available for a transit line. The link must be open for the TSYSSET of the transit mode being coded. For example, in order to code a rail line, the TSYSSET must include an 'm' which is the inter-city rail mode code.

To add a transit line, do the following:

1. Add the stop points
   1. Set stop points as the current network type
   2. Switch the mode to insert mode
   3. Make sure “Generate stop automatically” is checked. There are actually three types of stops in VISUM – stop points, stop areas, and stops –that are used to model complex intermodal transit centers. In SWIM2, every transit stop node consists of all three. The ‘Generate stop automatically’ option does this automatically when coding the stop points.
   4. Click a node to add a stop – use the same number as the node number
2. Add the line
   1. Set Lines as the current network type
   2. Switch the mode to add mode
   3. Create “Line” and click on the map
   4. Enter the Line name
   5. Set the transport system (mode)
3. Add the line route
   1. Set Lines as the current network type
   2. Switch the mode to insert mode
   3. Create “Line Route” and click on the map
   4. Select the Line of interest and name the new Line Route as “1”
   5. Click and drag from the first stop to the next stop and VISUM will shortest path route across the network open for that line TSYSSET. Continue to click and drag from the last stop point to the next one until the line is complete.
   6. Click OK in the Editing dialog to get the standard VISUM line route dialog.
   7. Select the Profiles tab (for the Time Profile associated with this line route) and set the Boarding, Alighting, and Stop time attributes for each stop on the line.
   8. Select the User Profile tab (for user defined attributed for the Time Profile associated with this line route) and set the headway attribute (**Emme\_Headway**) in seconds.
   9. Click OK to close the edit Line Route dialog.
   10. Update other Line Route attributes via the Line Route listing at Lists + Line Route. These additional attributes include:
       1. SPEED
       2. NET
   11. Note the Emme\_LayoverTime field is not used because the import process split lines into two separate lines whenever there was a layover.

To modify a line route:

1. Set Lines as the current network type
2. Switch the mode to the edit mode
3. Switch to Line Routes and select the Line Route to edit
   1. Double click to enter the Line Route edit dialog
   2. Click the Edit Shape button to edit the node sequence
      1. Click a beginning and ending tie point to define the portion of the line route that you want to change, and the portions of the line routes that will not change.
      2. Then click and drag a stop within the tie points to a new stop and VISUM will route the line from the beginning tie point to the ending tie point and through the new stop.
   3. The line route can also be modified in the Listings
4. Note that is important to set “Create Profile Points Automatically” in the Edit Line Route dialog to ensure all components of the line route node sequence are created when editing existing line routes. This is especially important when a new node/stop is added in the middle of the line route node sequence.

To copy or add a similar line route:

1. Set Lines as the current network type
2. Switch to Lines in the Lines Dialog (as opposed to Line Routes)
3. Select the Line to copy
4. Right click and select Copy Line from the menu
5. Enter a new name for the copied line
6. Switch to Line Routes in the Lines Dialog
7. Select the new Line Route
8. Edit the shape of the Line Route
9. Run export when editing is complete and check log file for errors.

## Differencing Networks

The VISUM difference network feature – File + Difference Network – allows the user to compare **every** attribute of **every** network object in two version files. In addition, a new network attribute called “DIFFNET” is created for each network object with one of the following values: “NET1” for only being in network 1, “NET2” for only being in network 2, and “BOTH” for being in both networks. This tool is useful for auditing edits.

# Appendix Zip Matrix Format

SWIM uses an open source matrix format called – zipped matrix, or ZMX. The file is a zip archive that contains the following files:

1. \_version
2. \_description
3. \_name
4. \_external column numbers
5. \_external row numbers
6. \_columns
7. \_rows
8. row\_<i>; where <i> is a matrix row of data

Here is an example of the contents of each file:

1. 2
2. Distance matrix
3. DIST
4. 100 101 102
5. 100 101 102
6. 3
7. 3
8. row\_1: 10.21 45.34 23.35
9. row\_2: 9.45 4.56 2.34
10. row\_3: 1.23 2.45 2.34

Each row\_<i> file is a series of bytes stored as big-endian floats.

ZMX files can be read with the stand alone ZipMatrixReader program or read into R using the readZipMat function. To view the matrix using the ZipMatrixReader program, type the following:

[model\_install\_path]\model\lib\java7\bin\java.exe –Dlog4j.configuration=[scenario\_path]\model\code\info\_log4j.xml.for\_matrix\_viewer –cp [scenario\_path]\model\code\tlumip.jar com.pb.common.matrix.ui.MatrixViewer [path\_to\_zmx\_file]

*where:  
[scenario\_path] = [model\_install\_path]\[scenario\_name]  
[path\_to\_zmx\_file] = absolute file path to zmx file*

To read a ZMX file into R, load the readZipMat function defined in the build\_Viz\_DB.R script in the /[scenario\_name]/model/code/viz/ directory and type:

readZipMat([path\_to\_zmx\_file])

# Appendix Select Link Module

This appendix contains more detailed information about the Select Link input files and how to post-process the outputs. The steps to run the select link module are:

1. During scenario set-up, edit the tsteps.csv file to contain a ‘1’ in the SL column for each year where the user would like to run Select Link. The TA module must have been run in this year or a previous year for the assignment paths to be available.
2. Create the mapping of SWIM network links to the subarea external stations - selectLinks.csv

## Creating SelectLinks.csv

VISUM and/or ArcGIS can be used to map SWIM2 links to the external stations of the subarea. The file must be in the following format:

Table 7: Example SelectLinks.csv File

|  |  |  |  |
| --- | --- | --- | --- |
| FROMNODE | TONODE | DIRECTION | STATIONNUMBER |
| 15801 | 16091 | OUT | 1 |
| 16091 | 15801 | IN | 1 |
| 15980 | 16014 | IN | 4 |
| 16014 | 15980 | OUT | 4 |

FROMNODE and TONODE come from the SWIM links. The DIRECTION field means that the link is going into (IN) or out of (OUT) the selected area. The STATIONNUMBER is the external station of the subarea. This file needs to be located in the \[scenario name]\inputs\t0\ directory.

There may be subarea external stations that do not map to a SWIM zone. This may occur because SWIM2 statewide model has a coarser network and zone density than many urban models. Those links can be left out of the SWIM2 file. However, all SWIM links that enter/exit the subarea must be mapped to an external station or else trips will be lost.

## Outputs

Select link output is put in the scenario year-specific folder (\[scenario name]\outputs\tn). The module outputs the following files, with the key addition being the attributes in Table 12 below. The resulting files can then be used to inform urban area external models.

1. Trips\_CTTruck\_select\_link.csv – subarea CT trips
2. Trips\_ETTruck\_select\_link.csv – subarea ET trips
3. Trips\_LDTPerson\_select\_link.csv – subarea LDT person trips
4. Trips\_SDTPerson\_select\_link.csv – subarea SDT trips
5. SynPop\_Taz\_Summary.csv – SWIM zonal household and population data
6. Employment.csv – SWIM zonal employment data by type

Table 8: List of Attributes Added to the Trip List Files by Select Link

|  |  |
| --- | --- |
| Field | Description |
| HOME\_ZONE | Home location zone |
| EXTERNAL\_ZONE\_ORIGIN | External station if the trip is inbound to the subarea, else origin zone |
| EXTERNAL\_ZONE\_DESTINATION | External station if the trip is outbound to the subarea, else destination zone |
| SELECT\_LINK\_PERCENT | Percent of assigned traffic volume between the OD pair on the selected link (i.e. an external station) |

# Appendix Master VISUM Output File

A scenario run will produce a master VISUM output file (/inputs/t0/swimNetworkAttributes.ver) in each /outputs/t[year] directory. This version file contains all of the network and zonal data that is produced and updated during that year’s run, including the demand matrices and skim matrices used. Furthermore, up to four additional version files are created to save the assigned paths from four periods (PK, OP, PM and NI). The version file contains the world markets, and their attributes (as POIs) and all the data previously contained in ActivityConstraintsIWorldMarkets.csv. The version file also contains AgForestFloorspace.csv and AllZones.csv data and the latest calibrated floorspace data. Furthermore, the master VISUM output file contains the peak and off-peak period specific transit line route headway attributes and the link auto volume, truck volume, auto travel time, truck travel, transit ridership, transit and auto demand time from each time period, including daily.

When running traffic assignment (TA), there are several options that can be changed in the properties file. First, either standard (path based) or LUCE assignment can be performed. In addition, a “full” assignment or a standard assignment can be run, with the first including the PM and NI periods and the latter consisting of just the PK and OP periods. These are controlled by the following properties in the template:

'ta.assignment.type' -> “LUCE” or “PATHBASED”

'ta.assignment.periods' -> “ALL” or “PKOP”

The defaults are set to “PATHBASED” and “ALL”, producing path based assignments and corresponding outputs for four time periods.

See the SWIM\_versionFile.xlsx for a description of the attributes managed in the version file.

1. http://www.oregon.gov/ODOT/TD/TP/pages/statewide.aspx [↑](#footnote-ref-1)
2. This may be different depending on how VISUM and Python are setup on the machine. The model requires the numpy and VisumPy libraries included with VISUM in the PythonModules folder of the VISUM installation folder. These libraries need to either be copied into the Python install location referenced in the SWIM properties file or included in the Python25 or Python27 folders in the SWIM main/model folder described later in this document. [↑](#footnote-ref-2)
3. A knowledgeable user can also modify the national forecast and Oregon trends directly in the scenario’s /t0/model\_data file; but re-calibration of the NED module may be required for major changes. It should be noted that unlike the rest of the model, the units in the absolute.csv, marginal.csv and model\_data.csv are in constant dollar terms (except for output (OOuti) and sales to final demand (OFDi), which are in year 2000 dollars), not 1990 dollars as used in the rest of the SWIM2 model. [↑](#footnote-ref-3)