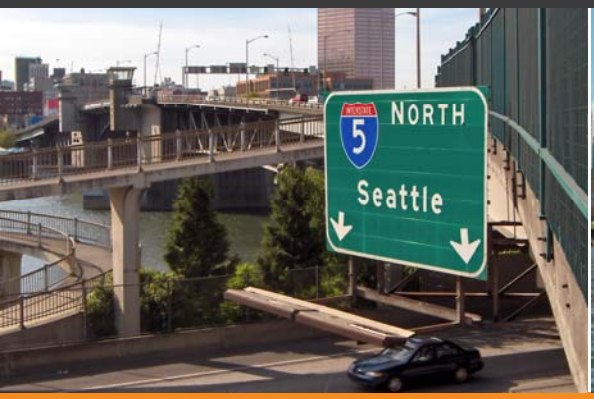
DRAFT TECHNICAL MEMORANDUM 1: POPULATION SYNTHESIZER SPECIFICATION

OREGON POPULATION SYNTHESIZER PROJECT

3.8.2017



PREPARED FOR:
OREGON DEPARTMENT OF TRANSPORTATION

SUBMITTED BY:
RSG

55 Railroad Row
White River Junction, VT 05001
802.295.4999
www.rsginc.com

IN COOPERATION WITH:
NAME OF SUBCONTRACTORS

CONTENTS

List of Figures

No table of figures entries found.

List of Tables

1.0 INTRODUCTION

The Oregon Department of Transportation (ODOT) and its partner agencies have a need for one standardized Population Synthesis tool. There are multiple tools currently used by ODOT and other planning agencies in Oregon for population synthesis, including PopSynIII, PopGen, the Oregon Statewide Model (SWIM) Synthetic Population Generator (SPG) I and II, and a population synthesizer implemented as part of the VisionEval tool. Each tool operates slightly differently and has advantages and disadvantages. This memorandum describes a new population synthesis tool that can replace the current tools used for urban models, SWIM, and strategic planning models. It will be capable of generating synthetic populations at the urban and regional level.

2.0 USE CASES AND REQUIREMENTS

2.1 | USE CASES

The tool must be suitable for the following use cases:

1) Urban area synthetic populations: Urban area synthetic populations are the primary input to activity-based travel demand models. Typically, synthetic populations generated for urban areas are controlled using 4-8 household and person level controls, at various levels of geography including county, transportation analysis zone (TAZ) and/or micro-analysis zone (MAZ) controls. The households generated by the population synthesizer must be attributed with a residence location at the TAZ and/or MAZ level, and accept both household and person controls at multiple geographic levels.

2) Statewide synthetic populations: The Oregon Statewide Integrated Model (SWIM) utilizes a synthetic population for the person travel components. SWIM generates a synthetic population for the entire state of Oregon plus a halo that includes Southwest Washington, part of Idaho including Boise, and part of Northern California and Nevada north of Reno. SWIM currently operates with 3,000 TAZs and utilizes household constraints. Because SWIM generates a synthetic population for every 3 years across a 30 or 40-year simulation, runtime is a very important issue for this use case.

3) Scenario Planning Models: There are a number of scenario planning and evaluation tools being developed or already in use in Oregon, including the Greenhouse Gas Strategic Transportation Energy Planning (GreenSTEP) and the Rapid Policy Assessment Tool (RPAT). These tools work at either the statewide or urban level of geography and typically utilize household constraints. A key difference between synthetic populations generated for scenario planning models versus urban or statewide models is that scenario planning models typically do not work at the TAZ level; scenario planning models may only utilize between one or a dozen district geographies for controls. However, scenario planning models may require multiple distributions with different controls to explore uncertainty associated with total households or control variables. While scenario planning model use case should be considered in the design of the Oregon population synthesizer, the many unknowns associated with this use case make it a lower priority than urban or statewide synthetic populations.

2.2 | POPULATION SYNTHESIZER REQUIREMENTS

Following is a list of population synthesizer requirements derived from the above use cases, as well as experience gained using the PopSyn III tool originally developed for Maricopa Association of Governments and used to generate synthetic populations at both the urban and statewide levels.

1) The population synthesizer should work with comma-separated value inputs and Public Use Microdata Sample (PUMS) household and person files downloaded

directly from the Census Bureau, rather than a SQL database. PopSyn III requires a SQL server database for storage of all inputs and outputs though it does not utilize any database technology in the actual creation of the synthetic population. The time investment that is required to set up SQL server in order to run the tool is unnecessary.

2) The population synthesizer should accept controls for a minimum of three levels of geography, as per the current PopSyn III tool. Further, the tool should not utilize the "MAZ" and "TAZ" terminology to describe the geographies since, although the word "meta" correctly implies that the geography is as large or larger than the PUMA, the actual definition of each lower level geography may change depending upon how the tool is used. Instead, we suggest geographic definitions that are independent of specific modeling methodologies, such as "Alpha", "Beta", and "Meta". Or if more than two sub-PUMA controls are possible in the algorithm (for maximum flexibility), then the controls could be named "Meta" for the PUMA or greater controls, then Geography1, Geography2, Geography3, and so on for subsequently smaller geographies.

3) The population synthesizer should accept both household and person level controls at any of the geographies specified above. Each control will be specified at one of the geographies with separate household and person inputs for each geography, and mapped to a field and attribute in the relevant input PUMS household or person data files. The user may optionally provide an integer weight that expresses the relative importance of the control compared to other controls across all geographies and across both households and persons.

4) The population synthesizer should provide a consistency checker that ensures that all household controls and person controls are consistent each level of geography. For example, all household controls for each alpha zone should add up to the same number of households. Similarly, all person controls for each alpha zone should add up to the same number of persons. A more complicated input checking procedure could be developed that ensures that all person controls add up to the persons implied by the household size distributions at whatever geography that control is specified at (should it be specified, and by using a user-input average household size for the largest household size category). Finally, household size compared to number of workers per household distributions can be checked for consistency. These last two checks are considered less important than ensuring total household and persons match across all controls. Failure to pass the test could result in a warning or an error message, at the user's discretion.

5) The population synthesizer should (optionally) provide output that is automatically linked with the full Public Use Microdata Sample (PUMS) household and person files.

6) The population synthesizer should (optionally) provide the aggregate multi-way distribution for any given geography of household attributes (minimum requirement) or household + person attributes, for any given synthetic population,

prior to integerization. This provides a useful dataset for debugging and validation of the tool, as well as a useful input to aggregate demand models such as trip generation models. Thus while not strictly a use case, the tool could be used as a replacement for household cross-classification models in the Oregon trip-based modeling frameworks including Jointly-estimated Models in R (JEMnR) and the Oregon Small Urban Model (OSUM).

7) The PopSyn III algorithm allocates households in order from smallest to largest zone according to the target number of households in each zone. This is done so that the error resulting from integerizing the household weights is allocated to the largest zone, in the expectation that the largest zone can 'absorb' the most error. However, this can result in cases where the last zone has significant error for certain attributes. For example, the application of PopSyn III for the Corvallis-Albany-Lebanon model resulted in a large suburban zone in Albany with a significant over-estimate of Oregon State University students. The Oregon population synthesizer should reduce or eliminate this 'garbage zone' problem.

8) The population synthesizer built for Atlanta Regional Commission had an attractive feature where the user could specify a list of PUMAs for any given PUMA geography, so that the tool could utilize households from other PUMA geographies in order to better fit control variables. The motivation for this feature is that as regions grow and change, PUMAs from other regions may be more appropriate sources of households than the PUMS sample from the last Census for the region. This should be a feature of the population synthesizer developed for this project.

9) One potential application of the population synthesis tool is to generate synthetic populations for traffic impact studies. In this type of application, a synthetic population would need to be generated for small geographies (one or a handful of MAZs or TAZs) without perturbing the synthetic population for the rest of the region. Typically only limited control variables are available for the new population (household type, perhaps stratified by cost). For example, San Diego Association of Governments regularly performs traffic impact analysis as a fee-based service for member agencies and the private sector.

10) There must be sufficient debugging in the code to ensure that it is working properly and to track errors in unknown future use cases. The debugging must be clear enough that the average user with a good working knowledge of population synthesis can interpret the outputs. For example, the tool might produce intermediate files of household weights at mid-level geographies to determine the extent of error in the balanced household weights. It could also produce the integerized weights so that error introduced in the integerization process can be evaluated for any given geographies.

3.0 POPULATION SYNTHESIZER ALGORITHM

3.1 | POPSYN III ALGORITHM

This section summarizes the current PopSyn III algorithm as per the details from Vovsha et al, 2014[1] and the source code from software implementation. PopSyn III operates at four geographic levels – lower, middle, seed and meta. For a typical implementation, these would translate to MAZ, TAZ, PUMA and County. Seed level is the geography at which the seed sample is available which normally is the PUMA geography. Controls are specified at lower, middle and meta geographic level. At present, PopSyn III requires at least one control to be specified at lower and meta geography as an algorithmic requirement. However, in cases where controls are not available for those geographies, data can be arranged in a smart way to make PopSyn III work with the available data.

PopSyn III is built around the traditional list balancing procedure which is applied at multiple geographic levels. While the balancing procedure can work independently for different geographic level, the challenge is to maintain correspondence between different geographic levels. This is achieved by following an aggregation approach for the controls where in the controls from the lower geographies are aggregated to the upper geographies. The list balancing implementation follows an allocation approach where in the results from an upper geography are distributed or allocated to a lower geography within the upper geography.

Two core and independent steps in the PopSyn III pseudocode can be mainly grouped as – core list balancing and integerizing. The following sub-sections first describe these components and then provide details of how these are stitched together in the full implementation.

CORE LIST BALANCING

The list balancing procedure in the context of population synthesis takes a list of households with an initial weight attached to each record. The objective is to update these weights in order to match marginal distributions of different variables, referred to as control variables. The output of the list balancing procedure is a list of weights which satisfies the marginal distribution of control variables for the given input list of households. The list balancing procedure is generally applied for a geographic zone for which the marginal controls are available. The very first step in implementation of list balancing procedure is formation of the incidence matrix for the given list of households. The incidence table determines which household contributes to which control. Table 1 presents an example of an incidence table where,

$i = 1,2...I$ = household and person controls,

A_i = values of controls to be met for the given zone,

[1] Vovsha, Peter, J Hicks, B Paul, V Livshits, K Jeon, P Maneva, New Features of Population Synthesis, Presented at the 2014 Annual Meeting, Transportation Research Board, Washington, D.C.

$n \in N$ = seed set of households in the PUMA (or any other sample),
w_n = a priori household weighs assigned to the PUMS household,
$a_{in} \geq 0$ = household attributes, i.e. coefficients of contribution to each control.

TABLE 1: INCIDENCE TABLE

HH ID	HH size				Person age				HH initial weight
	1	2	3	4+	0-15	16-35	36-64	65+	
	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	w_n
$n=1$	1							1	20
$n=2$		1			1	1			20
$n=3$			1			1	2		20
$n=4$				1		2	2		20
$n=5$				1	1	3	2		20
Current values	20	20	20	40	40	140	120	20	
Control (A_i)	100	200	250	300	400	400	650	250	

As such, there are multiple set of weights (x_n) that would satisfy the given marginal distributions. However, ideally these weights are desired to be as uniform as possible across all households. In order to achieve this, the list balancing procedure can be formulated as a convex entropy maximization problem as shown below:

$$\min_{\{x_n\}} \sum_n x_n \ln \frac{x_n}{w_n},$$

Subject to constraints:

$$\sum_n a_{in} \times x_n = A_i, (\alpha_i),$$

$$x_n \geq 0 ,$$

where α_i represents dual variables that give rise to the balancing factors.

The solution for the above problem is as follows:

$$x_n = k \times w_n \times exp(\sum_i a_{in}\, \alpha_i) = w_n \times \prod_i [exp(\alpha_i)]^{a_{in}} = w_n \times \prod_i (\hat{\alpha}_i)^{a_{in}},$$

where $\hat{\alpha}_i$ represents balancing factors that have to be calculated iteratively.

Control relaxations

The constrained optimization problem described above would yield a solution only if the controls allow for a solution to exist. In reality, the controls may not always be perfectly consistent, in which case the problem may not yield a solution. Relaxation factors can be introduced in the above problem to yield a solution even when the controls are not perfect. The modified specification is shown below:

$$\min_{\{x_n, z_i\}} \sum_n x_n \ln \frac{x_n}{w_n} + \sum_i z_i \ln z_i,$$

Subject to constraints:

$$\sum_n a_{in} \times x_n = A_i \times z_i, (\alpha_i),$$

$$x_n \geq 0, z_i \geq 0,$$

Where z_i represent relaxation factors

Importance weights

It may not be desirable to allow for all the controls to be relaxed. This important feature is incorporated by addition of importance factors in the objective function. In the specification below, μ_i represent importance weights for the controls. Since this is a minimization problem, a higher importance weight would result in a lower relaxation factor for the corresponding control.

$$\min_{\{x_n, z_i\}} \sum_n x_n \ln \frac{x_n}{w_n} + \sum_i \mu_i z_i \ln z_i,$$

Subject to constraints:

$$\sum_n a_{in} \times x_n = A_i \times z_i, (\alpha_i),$$

$$x_n \geq 0, z_i \geq 0,$$

Upper and lower bounds

Another important desired feature which becomes very useful in implementation is having upper and lower bounds on weights. This would prevent extreme solutions.

Maximum expansion factor

The upper and lower bounds on weights could be tagged to the initial weight of each household. This is achieved by introducing a maximum expansion factor (E) which bounds the households weights as follows:

$$0 = \underline{w}_n \leq x_n \leq \overline{w}_n = E \times w_n$$

Newton-Raphson Method

This convex optimization with linear constraints can be solved iteratively using the Newton-Raphson method. The pseudocode for solving this problem using the Newton-Raphson method is shown below:

```
# INPUTS
    A_I                                     #Control Set
    W_N                                     #Initial weights
    a[I][N]                                 #Incidence matrix
    U_I                                     #Importance factors
    E                                       #Maximum expansion factor/bounds
    MAX_ITERATION = 100000                  #Maximum number of iterations
    MAX_GAP = 1.0e-9                        #Termination criteria
    IMPORTANCE_ADJUST = 2                   #Importance Adjust
    IMPORTANCE_ADJUST_COUNT = 100           #Iterations after which adjustment is applied
    MINIMUM_IMPORTANCE = 1.0                #Minimum importance
    MAXIMUM_RELAXATION_FACTOR = 1000000     #Maximum relaxation factor
    MIN_CONTROL_VALUE = 0.1                 #Minimum control value

# INITIALIZATION
    Z[i]          := 1.0                    #initial relaxation factors
    X[n][0]       := W[n]                   #initial household weights
    lbWeights[n]  := 0                      #lower bound on weights
    ubWeights[n]  := MAX_VALUE              #initial upper bound if not an input
    adjust        := 1                      #adjustment factor
    oldAdjust     := 1


# ITERATIONS
    for(iter=0, iter<MAX_ITERATION, iter++)
    {
        #importance adjustment as number of iterations progress
        if ( iter > 0 && iter % IMPORTANCE_ADJUST_COUNT == 0 ) {
            adjust = oldAdjust / IMPORTANCE_ADJUST
            oldAdjust = adjust
        }
        #go through all controls
        for_each (A[i] in A_I)
        {
            X := SUM_n{X[n][iter-1]*a[i][n]}
            Y := SUM_n{X[n][iter-1]*(a[i][n])^2}

            if (isTotalHHsControl) {adjust = 1}  else {adjust = oldAdjust}

            if (X>0){
                if(A[i]>0){
                    alpha[i] := 1 - (X - A[i]*Z[i])/(Y + A[i]*Z[i]*(1/max(U[i]*adjust, MINIMUM_IMPORTANCE)))
                }else{
                    alpha[i] := 0.01
                }
            }else{
                alpha[i] := 1.0
            }
            #update HH weights
            if(a[i][n]>0){
                X[n][iter] := X[n][iter-1] * alpha[i]^a[i][n]
                X[n][iter] := max(X[n][iter], W[n]/E)
                X[n][iter] := min(X[n][iter], W[n]*E)
            }
            #update relaxation factors
            Z[i] := Z[i] * (1/alpha[i])^(1/max(U[i]*adjust, MINIMUM_IMPORTANCE))
            Z[i] := min(Z[i], MAXIMUM_RELAXATION_FACTOR)
        }
        maxGammaDiff := max(alpha[i]-1)
        delta        := SUM_n{ABS(X[n][iter] - X[n][iter-1])}/N
        if (delta <= MAX_GAP && maxGammaDiff < MAX_GAP) break
    }

# OUTPUTS
    X[n]        #Final fractional weights
    Z[i]        #Relaxation factors
    A[i]*Z[i]   #Relaxed controls
```

INTEGERIZING

The final output from the list balancing procedure are fractional weights corresponding to each household in the seed sample. Fractional weights cannot be used for expansion of the seed sample, therefore, weights need to be integerized, or converted to whole numbers. Simple rounding can result in violation of total household and/or person controls, so an optimization problem is formulated for the fractional part of the household weights. The residual controls are defined as:

$$\underline{A_i} = A_i - \sum_n a_{in} \times int\ (x_n)$$

The residual weights range from 0 to 1 and can be assumed to be binary (0 or 1). With these, a maximum entropy problem can be formulated for binary weights (y_n) as follows:

$$\min \sum_n y_n \times \begin{bmatrix} \ln\left(\frac{y_n}{x_n}\right), & if \quad y_n = 1 \\ 0 & if \quad y_n = 0 \end{bmatrix} \Rightarrow \max \sum_n y_n \times \ln x_n ,$$

Subject to constraints:

$$\sum_n a_{in} \times y_n = \underline{A_i} ,$$

$$y_n = 0,1$$

Here too, slack variables or relaxation factors can be introduced to handle cases when no solution exists as follows:

$$\max\{\textstyle\sum_n y_n \times \ln \underline{x_n} - 999 \times \sum_i U_i - 999 \times \sum_i V_i\},$$

Subject to constraints:

$$\textstyle\sum_n a_{in} \times y_n \leq \underline{A_i} + U_i,$$

$$\textstyle\sum_n a_{in} \times y_n \geq \underline{A_i} - V_i,$$

$$y_n = 0,1 ,$$

$$U_i \geq 0, V_i \geq 0.$$

POPSYN III PSEUDOCODE

As stated earlier, the main challenge in PopSyn III implementation is handling of multiple geographies. Controls from lower geography can be aggregated to an upper geography but it is not easy to disaggregate controls from an upper geography to a lower geography. Therefore, a two-step approach is followed under which the controls are aggregated upwards while the balancing is implemented from upper to lower geography. This ensures that at each step the controls are being satisfied for the current geographic level without violating the controls for an upper geography. However, list balancing at the meta level with all the

controls and for the entire seed sample can be quite complex. Therefore, a much simpler approach has been adopted for balancing at meta level. Step 2 and 3 in the pseudocode below comprise this approach termed as "***meta balancing***".

Step wise details of the entire PopSyn III algorithm are presented below:

STEP 1: PROCESS INPUTS

The inputs to PopSyn III include the following:

1. Geographic correspondence between lower, middle, seed (PUMA) and meta geographies.
2. Seed sample for each seed geography:

   ```
   HH[seed][n]
   ```

   Where **seed** is the seed geography index and **n** is the household index.

   Number of households in each seed zone in seed sample:

   ```
   numberOfHouseholds[seed]
   ```

   Initial sample weights in the seed sample. These are summed for each seed zone:

   ```
   totalSampleWeights[seed]=
   SumOverN(totalSampleWeights[seed][n])
   ```

3. Control set for each geography

   ```
   lowControls[low][i_low],
   medControls[med][i_med],
   metaControls[meta][i_meta]
   ```

   Where **low, med and meta** are geography indices and **i_low, i_med and i_meta** are the controls indices corresponding to each geography.

   In addition to control values, control definitions are also an input:

   ```
   controlDefinitions[i]
   ```

4. Importance weights for each control:

   ```
   importance[i]
   ```

   Where **i** is the control index.

5. Max expansion factor:

   ```
   maxExpansionFactor
   ```

6. Mid-level geography promotion field name: field in the seed sample which tells if a household record should be promoted for any mid-level geography zone.
7. Mid-level zone promotion factor: factor by which a household tagged to be promoted should be factored - **midZonePromotionFactor**

Aggregate controls

Low and mid-level controls are aggregated to seed level for initial balancing at the seed-level geography.

```
seedControls[seed] = Aggregate(lowControls[low][i_low],
                               medControls[med][i_med])
```

Incidence table creation

An incidence table is created separately for each seed geography for all controls [low, med and meta]:

```
incidence[seed][n][i] = createIncidence(HH[seed][n],
                                        controlDefinitions[i])
```

STEP 2: INITIAL SEED BALANCING

After initial processing of inputs, the next step is to balance the household records at the seed-level using the low and mid-level controls aggregated to the seed-level.

Balancing

```
for (seed in seedList) {
      doListBalancing{
      # with following inputs
      initialSampleWeights
      incidence[seed]
      seedControls[seed]

      #lower bound on weights
      lower_bound = 0

      #upper bound on weights
      upper_bound = max{(int)(maxExpansionFactor *
                    initialSampleWeights *
                    numberOfHouseholds[seed]/totalSampleWeights[s
                    eed] + 0.5), 1}
      #output
      initialSeedWeights[seed][n]
      }
}
```

The output of initial seed balancing is a set of initial seed balancing weights for each seed geography:

```
initialSeedWeights[seed][n]
```

Meta control factoring

As stated earlier, balancing at the meta level for the entire sample can be quite complex. Therefore, the meta level controls are distributed to form seed level controls. This is achieved by distributing the meta level controls to each seed zone proportional to the sum of initial seed balancing weights for that seed zone. In this regard, the first step is to sum the initial seed balancing weights for each seed zone for each meta control:

```
totalFactoredSeedWeights[seed][i_meta] =
SumOverN(incidence[seed][n][i_meta] *
initialSeedWeights[seed][n])
```

Next, weights are summed over all the seed zones within each meta zone to get meta-level totals of initial seed balancing weights for each meta-level control.

```
totalFactoredMetaWeights[meta][i_meta] =
SumOverSeed(incidence[seed][n][i_meta] *
initialSeedWeights[seed][n])
```

Now, the meta-level totals and controls are used to compute the scaling factors to be applied to the seed-level totals:

```
factor = metaControls[meta][i_meta]/
totalFactoredMetaWeights[meta][i_meta]
```

New seed-level controls from meta-level controls are computed as follows:

```
newMetaControls[seed][i_meta] =
int(totalFactoredSeedWeights[seed][i_meta] * factor + 0.5)
```

Create final balancing controls

The newly created meta-to-seed level are added to the existing set of seed level controls (aggregated from low and mid-level controls)

```
finalSeedControls[seed] = Concat(lowControls[seed][i_low],
medControls[seed][i_med], newMetaControls[seed][i_meta])
```

The final output from Step 2 is the final seed-level controls for each seed zone:

```
finalSeedControls[seed]
```

STEP 3: FINAL SEED BALANCING

A final balancing exercise needs to be done for each seed zone with aggregated low and mid-level controls and distributed meta-level controls.

```
for (seed in seedList) {
      doListBalancing{
      # with following inputs
      initialSampleWeights
      incidence[seed]
      finalSeedControls[seed]
      #lower bound on weights
      lower_bound = 0
      #upper bound on weights
      upper_bound = max{(int)(maxExpansionFactor *
                  initialSampleWeights *
                  numberOfHouseholds[seed]/totalSampleWeights[s
                  eed] + 0.5), 1}
      #output
      finalSeedWeights [seed][n]
```

```
        }
}
```

The final output from Step 3 are household weights for each seed zone

```
finalSeedWeights[seed][n]
```

STEP 4: MID & LOW-LEVEL ALLOCATION

From this point forward, the total weight for each seed zone does not change. In other words, the sum total of weight on a household record would not change. However, a household record in a seed zone can be allocated to multiple mid-level zone within the seed zone, thereby splitting the weight received by this record at the end of step 3. Similarly, the split weight received by a household record allocated to a mid-level zone can be further split to allocate the household record to multiple low-level zones within the mid-level zone. This splitting of weights from the seed zone to mid and low-level zone is termed as mid and low-level allocation. Algorithmically, this is achieved by performing a list balancing on each mid-level zone within a seed zone and setting the upper bound on weights to the starting weights of seed sample at end of Step 3. Zones are currently processed in order of increasing number of households (as determined by number of HHs control). After each iteration, the starting weights are updated by subtracting the resulting weights of the mid-level zone from the starting weight of the seed zone. Records with zero weights are removed from the seed sample after each iteration. The last mid zone is not processed whilst the remaining seed weights are allocated to the last mid-zone. The same procedure is repeated for each low zone within each mid zone. The pseudocode is presented below:

```
for (seed in seedList) {

finalSeedIntegerWeights = DoIntegerizing(incidence[seed],
seedInitialWeights[seed], finalSeedControls[seed])

rd1 = midZoneAllocation ()
rd2 = lowZoneAllocation()

}

midZoneAllocation (){

# sort mid geography zones by number of HHs in increasing order
sortedIndices = getSortedIndexArray()

if(sortedIndices.length == 1){
    # Allocate all HHs to available mid-level geography
}else{
    # loop over all mid-level geography
    for (int p=0; p < sortedIndices.length; p++) {
      if (the zone with non-zero HHs is the last zone)
        allocate all HHs to that zone, break

      # for all zones but the last zone
      initialWeights = finalSeedIntegerWeights
```

```
      # Factor initial weights with promotion factor for records
      with the current zone marked as promotion zone
      initialWeights = initialWeights * tazPromotionFactor

      doListBalancing{
      # with following inputs
      # filter records in seed by >0 initial weight
      initialWeights
      incidence[seed]
      medControls[med]
      lower_bound = 0                #lower bound on weights
      upper_bound = initialWeights  #upper bound on weights

      #output
      finalMidZoneWeights
      }

      finalMidZoneIntegerWeights = doIntegerizing()

      #Update initial weights
      initialWeights = initialWeights -
      finalMidZoneIntegerWeights

    }
}
}

lowZoneAllocation(){

# sort low-zones by number of HHs in increasing order
sortedIndices = getSortedIndexArray()

if(sortedIndices.length == 1){
    # Allocate all HHs to available low-zone
}else{
    # loop over all low-level geography
    for (int p=0; p < sortedIndices.length; p++) {
      if (the zone with non-zero HHs is the last zone)
        allocate all HHs to that zone, break

      # for all zones but the last zone
      initialWeights = finalMidZoneIntegerWeights

      doListBalancing{
      # with following inputs
      # filter records in seed by >0 initial weight
      initialWeights
      incidence[seed]
      lowControls[low]
      lower_bound = 0                #lower bound on weights
      upper_bound = initialWeights  #upper bound on weights

      #output
      finalLowZoneWeights
      }
```

```
        finalLowZoneIntegerWeights = doIntegerizing()

        #Update initial weights
        initialWeights = initialWeights -
        finalLowZoneIntegerWeights

    }
}
}
```

OUTPUT FILE CREATION

PopSyn III is written in Java and for all the data processing needs, it interfaces with MySQL or SQL server. The idea behind using a SQL database was that most of the data structures used in PopSyn III are tables, and SQL can come in handy for handling tables. However, since most of the operations are limited to SELECT and INSERT queries, other solutions can be employed for data handling. In the current version of PopSyn III, all raw input data files are required to be in CSV format. A pre-processing SQL script formats the data and loads them into a SQL server database. The core PopSyn III program in Java accesses these inputs via SQL queries. Intermediate temporary tables are also stored in the SQL database and final output tables are also written to SQL server, which can be exported in CSV format using a post-processing SQL script.

The seed input data consists of list of households and persons for each seed geography. Using seed data and controls specifications, PopSyn III builds incidence tables (Table 1) for each seed geography. The seed incidence table is used for Initial & Final Seed Balancing at the seed geography level. For low and mid-level geography balancing (or allocation), an incidence table for the corresponding upper seed geography is used. At the end of Final Seed Balancing, a final integer weight is added to each household record in the seed table.

The final outputs from PopSyn III are a household and person table. The household table attributes include a final household ID, original ID in the seed sample, initial weights, meta geography ID, seed ID, mid-geography ID, low-geography ID and final seed weights on the record. The formation of output table begins only at the Mid & Low-Level Allocation step in the PopSyn III algorithm. First a temporary household table with these fields is created and then rows are added as the algorithm progresses. The input to Mid & Low-Level Allocation step is a list of households for each seed geography with final integer weights. Mid and low-level allocation is implemented for each seed geography separately. The incidence table, household IDs and final seed integer weights are passed to each mid-level allocation subroutine call (for each mid-zone). Mid-level allocation returns a list of households for each mid-level geography with their mid-level geography weights. PopSyn III algorithm ensures that the sum of mid-geography weights across all mid-zones is equal to the total seed integer weights for the seed geography. Next, the outputs from mid-level allocation are passed to each low-level allocation subroutine call (for each low-zone). Low-level allocation returns a list of households for each low-level geography with their low-level geography weights. A low-zone ID and corresponding mid-zone, seed and meta IDs are

attached to the list of households for each low-level geography. This final list of households for each low-zone is appended to the temporary household table.

Figure 1 shows how tables are created at each step of the allocation process. For an implementation with S seed zones, allocation process starts with S input seed household tables. Each seed table multiplies into M number of tables after mid-level allocation, where M is the number of mid-zones within the seed geography. However, a multiplied table for a mid-zone would contain only records which were allocated to that mid-zone. Similarly, a mid-level table gets multiplied into low-level household tables for each low-zone within a mid-zone. For a region with S seed zones, M mid-zones and L low-zones, a total of S * M * L tables are produced. Each of these tables are appended to the temporary household table at the end of each iteration in the allocation process. Some of these tables can contain zero records if no households were allocated to a mid or low-zone. A record in the seed sample can appear in any mid or low-zone within the seed zone but cannot be shared across seeds. While records can be duplicated across geographies, the total weight across all geographies within each seed remains equal to the sum of final integer weights at the end of the final seed balancing. At the end, household attributes are added to the temporary household table by joining to the original seed sample table. The resulting table is the final household table. The final person table is also created by joining the seed person sample to the final household table. A post processing script converts the final outputs to a fully expanded household and person table. This is achieved by duplicating each record *W* times, where *W* is the final integer weight on each record.

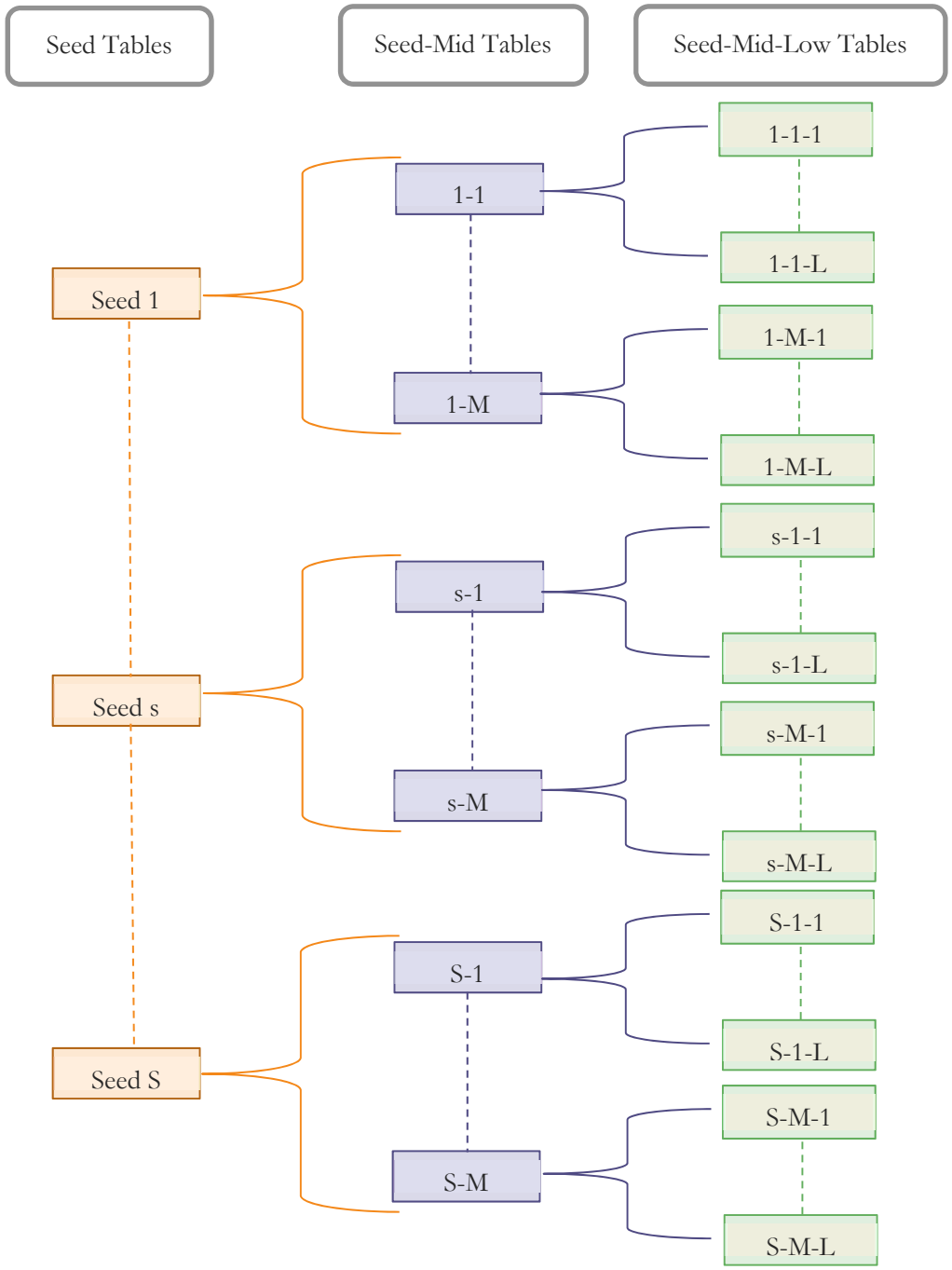


FIGURE 1: TABLE MULTIPLICATION IN ALLOCATION PROCESS

3.2 | ENHANCEMENTS TO EXISTING ALGORITHM

While PopSyn III includes various innovative features and controls, there are some issues which can lead to errors. Once such issue with PopSyn III algorithm is the way in which the mid and low-level allocation is implemented. In the current implementation, the allocation step iterates through mid and low zones sequentially, allocating households without replacement. However, a direct implication of this sequential processing is that the last zone is allocated all the remaining household records. Since households are not allocated to the last zone via list balancing, it's hard to match the controls for this zone. For a zone with less number of household the resulting error can be big. In order to avoid this, mid and low zones are sorted in increasing order of total number of households and processed in the same order. The idea being that the last zone, which has the most number of households, can easily absorb the resulting error in control matching. This however may not work always as desired especially for controls specified for a minority segment of the population (e.g., university students). A new methodology is proposed below which avoids the sequential processing of mid and low zones in the allocation process.

TABLE 2: INCIDENCE TABLE

	Mid Zone X			**Mid Zone Y**			**Mid Zone Z**			Weights
HHID	$i=1$	$i=2$	$i=3$	$i=1$	$i=2$	$i=3$	$i=1$	$i=2$	$i=3$	
$n=1$										x_1
$n=2$										x_2
$n=3$	***I***			***0***			***0***			x_3
$n=4$										x_4
$n=5$										x_5
$n=1$										y_1
$n=2$										y_2
$n=3$	***0***			***I***			***0***			y_3
$n=4$										y_4
$n=5$										y_5
$n=1$										z_1
$n=2$										z_2
$n=3$	***0***			***0***			***I***			z_3
$n=4$										z_4
$n=5$										z_5
Controls	A_{11}	A_{12}	A_{13}	A_{21}	A_{22}	A_{23}	A_{31}	A_{32}	A_{33}	

At the heart of the allocation process is List Balancing with lower and upper bounds on household weights. This new methodology proposes applying list balancing on all mid and low zones simultaneously. As a result, no single zone would end up collecting the cumulative error which was the case with sequential processing. Simultaneous list balancing requires

creation of a super incidence table. Table 1 presents an incidence table for a single seed zone. Table 2 presents a super incidence table for a seed zone with three mid zones, where *I* is the incidence table for the given seed zone and *0* is a matrix of same size but contains all zeros.

In the Table 2 example, there are three controls at the mid-zone level. The last row contains the control totals for each control in each mid-zone. The household weights assigned to each record for each mid-zone are represented by x, y and z variables. For this incidence table, the list balancing problem can be formulated as:

$$\min_{\{x_n, y_n, z_n\}} \sum_n x_n \ln \frac{x_n}{w_n} + y_n \ln \frac{y_n}{w_n} + z_n \ln \frac{z_n}{w_n},$$

Subject to constraints:

$$\sum_n a_{in} \times \{x_n, y_n, z_n\} = A_i, (\alpha_i),$$

$$\{x_n, y_n, z_n\} \geq 0,$$

$$x_n + y_n + z_n = S_n$$

Where S_n is the final integer seed weight on a household record.

The above formulation can be solved using the same iterative Newton Raphson method described in Section 1.1. x_n, y_n, z_n weights at the end of each iteration will need to be scaled to satisfy the $(x_n + y_n + z_n = S_n)$ constraint. The original list balancing procedure had just one control defined as the total households control which is constrained to be satisfied without any relaxations. Under this methodology, total household controls for all mid zones will have to be constrained to satisfy the total households for each mid zone. Finally, integerizing can be applied to obtain integer weights. The same procedure will be applied to allocate households records from mid to low zones.

4.0 SOFTWARE IMPLEMENTATION

4.1 | OVERVIEW

PopulationSim will be implemented in the ActivitySim framework. As summarized here, being implemented in the ActivitySim framework means:

- Design
  - Implemented in Python, making heavy use of the vectorized backend C/C++ libraries in pandas and numpy.
  - Vectorization instead of for loops when possible
  - Runs sub-models that solve Python expression files that operate on data tables
- Data Handling
  - Inputs are in CSV format, with the exception of settings
  - CSVs are read-in as pandas tables and stored in an intermediate HDF5 binary file that is used for data I/O throughout the model run
  - Key outputs are written to CSV files
- Key Data Structures
  - pandas.DataFrame - A data table with rows and columns, similar to an R data frame, Excel worksheet, or database table
  - pandas.Series - a vector of data, a column in a DataFrame table or a 1D array
  - numpy.array - an N-dimensional array of items of the same type, such as a matrix
- Model Orchestrator
  - ORCA is used for running the overall model system and for defining dynamic data tables, columns, and injectables (functions). Model steps are executed as steps registered with ORCA.
- Expressions
  - Model expressions are in CSV files and contain Python expressions, mainly pandas/numpy expression that operate on the input data tables. This helps to avoid modifying Python code when making changes to the model calculations.
- Code Documentation
  - Python code according to pep8 style guide
  - Written in reStructuredText markup, built with Sphinx and docstrings written in numpydoc
- Testing

  - A protected master branch that can only be written to after tests have passed
  - pytest for tests
  - TravisCI for building and testing with each commit

4.2 | MODEL STEPS (I.E. PROCESSORS)

The model steps are likely to be:

- orca.run(['input_pre_processor']) - read input tables, processes with pandas expressions, and creates tables in the datastore.
- orca.run(['setup_data_structures']) - setup geographic correspondence, seeds, control sets, weights, expansion factors, and incidence tables
- orca.run(['initial_seed_balancing']) - seed (puma) balancing, meta level balancing, meta control factoring, and meta final balancing
- orca.run(['final_seed_balancing']) - final balancing for each seed (puma) zone with aggregated low and mid-level controls and distributed meta-level controls
- orca.run(['mid_and_low_level_allocation']) - iteratively loop through zones and list balance on each mid-level zone within a meta zone and then each low-level zone within a mid-level zone. This is the current procedure, which will likely be revised.
- orca.run(['expand_population']) - expanded household and person records with final weights to one household and one person record per weight with unique IDs
- orca.run(['write_results']) - write the household and person files to CSV files

4.3 | FOLDER / FILE SETUP

- run_populationsim.py - runs populationSim, which runs the model steps described above
- configs folder - configuration settings
  - settings.yaml - settings such as input table names, table key fields, expression files, etc. An example settings file is below.

```
seed_households:
  filename : pums_hhs.csv
  id : hhnum
  seed_zone_id : puma
  weight : wgtp
seed_persons:
  filename : pums_pers.csv
  hh_id : SERIALNO
geographic_crosswalk:
  filename : geocwalk.csv
  seed_id : puma
  meta_id : district
  mid_id : taz
  low_id : maz
```

```
meta_control_data :
  filename : districts.csv
  id : district
mid_control_data :
  filename : tazs.csv
  id : taz
low_control_data :
  filename :  mazs.csv
  id : maz
input_pre_processor:
  seed_households : seed_households_expressions.csv
  seed_persons : seed_persons_expressions.csv
  meta_control_data : meta_control_data_expressions.csv
  mid_control_data : mid_control_data_expressions.csv
  low_control_data : low_control_data_expressions.csv
controls : controls.csv
maxExpansionFactor : 15
expanded_households :
  filename : households.csv
  output_fields : serialno, np, nwrkrs_esr, hincp
expanded_persons :
  filename : persons.csv
  output_fields : sporder, agep, relp, employed
```

- data folder - scenario inputs
  - seed households records table
  - seed persons records table
  - meta zones data table (i.e. controls and other data)
  - mid level zones data table (i.e. controls and other data)
  - low level zones data table (i.e. controls and other data)
  - geographic crosswalk
- outputs folder - key outputs
  - households.csv - expanded households
  - persons.csv - expanded persons
  - populationsim.hdf5 - HDF5 datastore

4.4 | INPUTS PRE-PROCESSOR

The inputs pre-processor reads each input table, runs pandas expressions (*_expressions.csv) against the table to create additional required table fields, and save the tables to the datastore. The inputs pre-processor exposes all five input tables to the expressions calculator so tables can be joined (such as households to persons for example). It reads the geographic crosswalk file in order to join meta, mid, and low level zone tables if needed. The format of the expressions file follows ActivitySim, as shown in the example `seed_households` expressions file below which operates on the `NPF` field:

Description	Target	Expression
HH is a family	famTag	pd.notnull(NPF) * 1

4.5 | CONTROL VARIABLES

The control variables input file (controls.csv) specifies the controls used for expanding the population. Like ActivitySim, Python expressions are used for specifying control constraints. An example file is below.

Description	Total HH Control	Geography	Seed Table	Importance	Control Field	Expression
Num HHs	True	low	households	10000	HHBASE	(WGTP > 0) & (WGTP < np.inf)
Num HHs by persons per HH	False	low	households	5000	HHSIZE1	NP==1
Num HHs by persons per HH	False	low	households	5000	HHSIZE1	NP==1
Num HHs by persons per HH	False	low	households	5000	HHSIZE1	NP >= 4
Num students by housing type	False	low	persons	10000	OSUfam	OSUTag==1
Num HH by household type	False	mid	households	100	SF	htype==1
Num persons by occupation category	False	meta	households	100	OCCP1	occp==1

Where:

- Geography can be `meta`, `mid`, and `low`
- Seed Table can be `households` or `persons` and if `persons`, then it aggregates to household using the `COUNT`