PECAS - for Spatial Economic Modelling

Bayesian SD Calibration – User Guide

System Documentation Technical Note
WORKING DRAFT

HBA Specto Incorporated

Calgary, Alberta
September 2018

Table of Contents

1. Introduction

This document is a user's guide to the SD Bayesian calibration software. The first two sections of the document summarize the context needed to understand SD calibration: how it works and the theory behind why it is done the way it is. More detail can be found in other documents: the SD model is described in the PECAS manuals and the Density Shaping Function reference; the calibration theory is described in the SD Calibration Theory reference.

The remaining sections are a comprehensive guide to setting up the SD calibration feature, performing calibration runs, and fixing problems encountered during runs.

2. SD model overview

PECAS is a spatial economic model for forecasting and policy analysis. The Space Development (SD) module of PECAS represents the behavior of profit-motivated developers as they consider constructing buildings (or other improvements) on land. The land of the region is represented as a database of parcels. Each parcel has specific locational attributes that modify the market price from nearby properties, as well as specific zoning regulations and conditions that are used to calculate the cost of development of each type on the each parcel. The "market price from nearby properties" is calculated in the other PECAS Module, Activity Allocation, for each Land Use Zone, LUZ, and then smoothed using a distance decay function with an exponent of two. (The smoothing represents a slightly wider view that developers consider, in particular in situations where an LUZ has a very small quantity of a type of space, developers being unwilling to make large decisions based on a small sample of very local price observations.)

There is an additional aspatial component, called "Density Shaping", representing the desirability (potential increased rent) and/or construction costs associated with different intensities of development, or Floor Area Ratios. For example: as high-rises get taller the amount of footprint required for elevators increases, increasing the costs per usable square foot of development; and lower intensity single family dwelling development are associated with a higher price per area of building space, all other things being equal, because of the larger residential yards.

PECAS SD considers parcels one-by-one, in a random order, once per year, calculating the expected maximum profit over the range of allowed intensities for each space type, and uses a joint discrete-continuous logit formulation to select, using a random number generator, both a type of development event (for example “do nothing”, the most common type, “intensify”, “demolish and build new”), and, in the case of new development, an intensity of development.

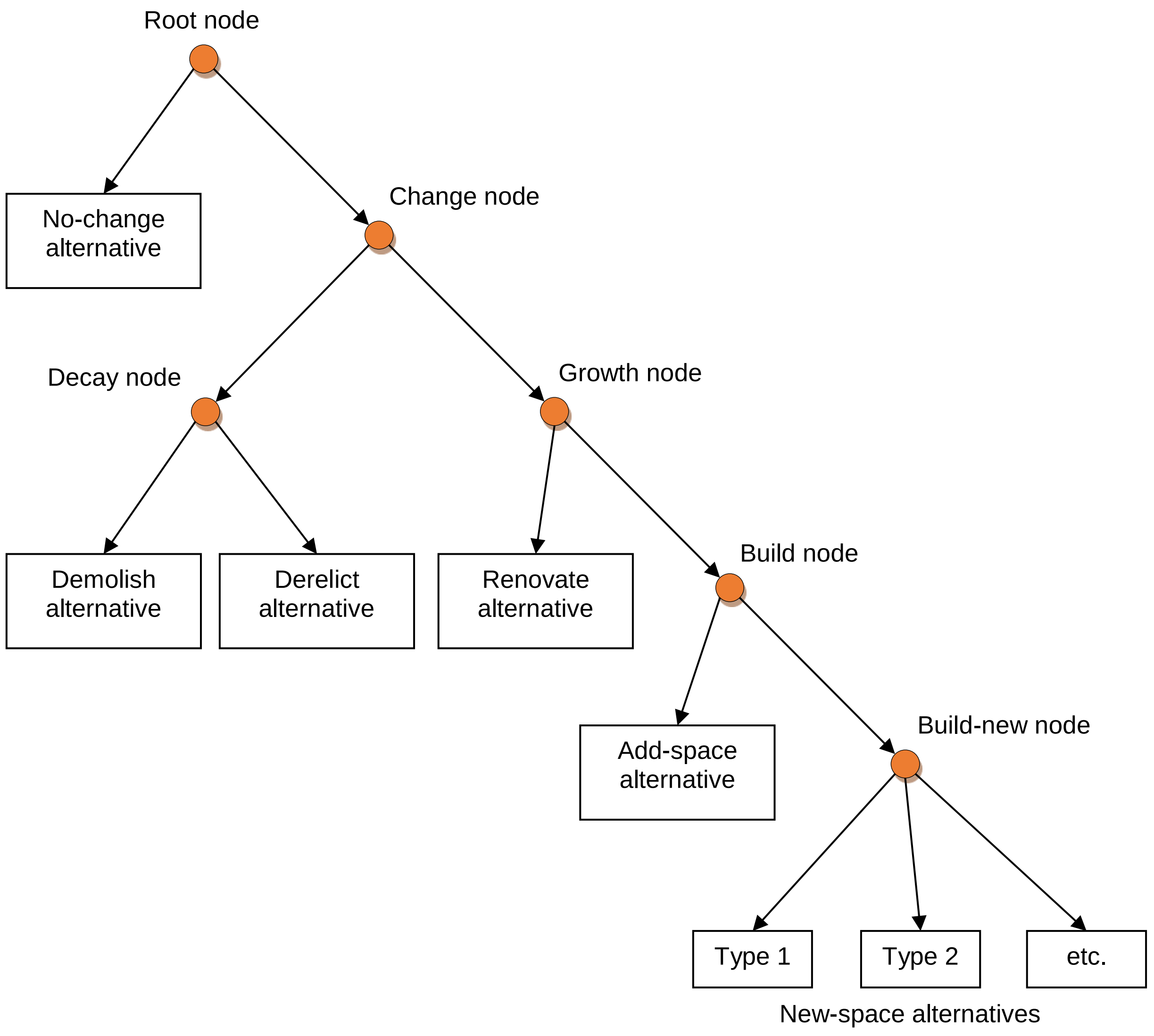

Figure 1: Logit model structure for development options in SD

3. Calibration theory overview

3.1. Approach

The SD module is calibrated so that it accurately represents the costs of construction, the potential return-on-investment of new development, and the behavior of developers. To represent regional development behavior, observed development in the recent past is used to develop estimation or calibration targets. However development is a slow process, and even in a large region there are seldom enough observations of development to accurately reveal the preferences, costs, profit potential and other factors that motivate developers. Multiple linear correlations also exist, for example it can be difficult to estimate statistically whether developers of one space type are avoiding specific types of locations or whether developers of a different space type are outbidding them for those locations. In general, observations of past development behavior have proven to be insufficient to forecast the long-term future behavior of developers. To properly represent the behavior of developers, primary data on construction costs and rent adjustments is required, along with a strong theoretical basis on the profit motivations of developers. Observations of development on other jurisdictions can also be employed. For example, if developers build highrises in Los Angeles under certain situations, it is an indication that developers may be likely to build future highrises in a different city in the future, should that other city ever evolve to be more like present-day Los Angeles.

The strategy for calibration the SD module takes all this into account. The SD parameters are established initially through published data sets, literature review, theory, and past estimations of similar PECAS models. The parameters are then adjusted based on the observations of development in the region. A Bayesian approach is used, so that the analyst can control the degree to which prior knowledge and theory can be adjusted in response to the limited data.

The connection to the size of the construction industry in AA’s representation is not normally turned on during SD calibration. In the calibration year, SDs parameters are calibrated to produce the observed amount of construction, and AAs construction industry size is also based on the observed amount of construction. In application to future years during a forecast, the SD module is constrained to match AAs construction industry size through the *Construction Capacity* feature, and AAs construction industry size can adapt over time to profits reported by SD through the *Construction Expected Profit* feature, but these features are both turned off during base year calibration.

3.2. Mathematical formulation

From Bayes’ theorem, the most likely parameter values are those such that

$$\boldsymbol{\Phi}(\boldsymbol{c}) = \sum_{i=1}^{m} \frac{(\theta_i(\boldsymbol{c}) - \bar{t}_i)^2}{\sigma_{Ti}^2} + \sum_{j=1}^{n} \frac{\left(c_j - \overline{c_j}\right)^2}{\sigma_{Cj}^2} \tag{1}$$

is as small as possible. In this equation, c_j, $\overline{c_j}$, and σ_{Cj} are the current value, prior mean value (established by the analyst based on previous knowledge and experience), and tolerance for the jth parameter, while $\overline{t_i}$ is the ith target value and σ_{Ti} is that target's tolerance. The corresponding modelled value $\theta_i(\boldsymbol{c})$ is the mathematical expectation of the quantity of interest given the current parameter values. The function $\boldsymbol{\Phi}(\boldsymbol{c})$ defined in the equation is the *objective* function, whose value SD calibration will try to bring to a minimum.

The objective function takes the form of a *weighted sum of squares of errors*. In other words, it is calculated by squaring each error term, dividing by the square of the weight, and then summing the results. Two types of error terms occur: parameter errors, representing the difference between a parameter value and its prior mean; and target errors, representing the difference between a target and SD's expectation of that quantity. Each is weighted by its tolerance value.

SD calibration finds the minimum value of the objective function using quasi-Newton optimization. Starting with an initial guess $\boldsymbol{c}_0$, it iterates the matrix equation

$$\boldsymbol{c}_{i+1} = \boldsymbol{c}_i - \boldsymbol{H}^{-1}(\boldsymbol{c}_i)\boldsymbol{g}(\boldsymbol{c}_i) \tag{2}$$

until either the convergence criterion or the iteration limit is reached. In this equation, $\boldsymbol{g}(\boldsymbol{c}_i)$ is the *gradient* – the derivative of $\boldsymbol{\Phi}(\boldsymbol{c})$ with respect to each of the parameters; $\boldsymbol{H}(\boldsymbol{c}_i)$ is the approximate *Hessian* matrix – the matrix of *second* derivatives of $\boldsymbol{\Phi}(\boldsymbol{c})$ with respect to each pair of parameters.

The gradient is calculated using the matrix equation

$$\boldsymbol{g}(\boldsymbol{c}) = 2\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{c})\boldsymbol{\Sigma}_T^{-1}(\boldsymbol{\theta}(\boldsymbol{c}) - \bar{\boldsymbol{t}}) + 2\boldsymbol{\Sigma}_C^{-1}(\boldsymbol{c} - \bar{\boldsymbol{c}}) \tag{3}$$

Here:

- $\boldsymbol{\Sigma}_T$ is a square matrix with the squared target tolerances (σ_T^2) along the diagonal and zeros everywhere else. $\boldsymbol{\Sigma}_C$ is a similarly-constructed matrix with the squared parameter tolerances (σ_C^2) along the diagonal.

- $\boldsymbol{J}(\boldsymbol{c})$ is the *Jacobian* matrix – the matrix containing the derivative of the expectation of each modeled value with respect to each parameter.

The Hessian is calculated using the matrix equation

$$\boldsymbol{H}(\boldsymbol{c}) = 2\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{c})\boldsymbol{\Sigma}_T^{-1}\boldsymbol{J}(\boldsymbol{c}) + 2\boldsymbol{\Sigma}_C^{-1} + \lambda\boldsymbol{D} \qquad (4)$$

Here:

- $\boldsymbol{D}$ is a square matrix created by taking only the diagonal elements of $2\boldsymbol{J}^{\mathrm{T}}(\boldsymbol{c})\boldsymbol{\Sigma}_T^{-1}\boldsymbol{J}(\boldsymbol{c}) + 2\boldsymbol{\Sigma}_C^{-1}$ (i.e. the rest of equation 4). If $\boldsymbol{H}$ is replaced by $\boldsymbol{D}$ in equation 2, then the solution algorithm degenerates to a form of gradient descent, which is slower but more reliable.
- λ is a step size control parameter. When it is large, the $\boldsymbol{D}$ term in equation 4 dominates the other terms, resulting in a conservative gradient descent step. When it is small, the rest of equation 4 dominates, and the solution algorithm closely approximates the faster Newton's method. The λ parameter is adjusted every iteration in response to the current estimation progress: if the most recent iteration improved (reduced) the value of $\boldsymbol{\Phi}(\boldsymbol{c})$, then λ is decreased to allow faster convergence; otherwise, the last iteration is retried with a larger value of λ in hopes that a more tentative step down the gradient will yield an improvement in $\boldsymbol{\Phi}(\boldsymbol{c})$.

SD calibration can be configured to print out many of the above values every iteration, which is instrumental in fixing a calibration attempt that does not converge or produces undesirable results.

As with any estimation based on least-squares, SD calibration will try to reduce as many errors as possible. Because the errors are *squared*, large errors are considered much worse than small ones (e.g. doubling an error results in a fourfold increase in the contribution to the objective function).

Of course, it is not in general possible to reduce all error values to zero; the algorithm must balance between the different parameters and targets. Since each error term is weighted by its tolerance, the tolerance values control how important it is to match each target or prior mean. In general, the algorithm sees a "one standard deviation error" (i.e.

an error equal in magnitude to the tolerance) in any parameter or target as equally bad, and will expend equal effort in reducing those errors.

There are two main ways that this property should be considered when assigning tolerance values. Firstly, if there is an obvious relationship between a given parameter and a given target, the tolerances assigned to each will control the balance between matching the target and staying close to the prior mean.

For example, the amount of construction of residential space in a typical SD model is largely controlled by the build constant for residential space. If the tolerance on the constant is large while that on the construction amount is small, this implies a belief that the observation of the construction amount should be matched, and the data sources used to derive the target is reliable and consistent with other model inputs. In this case, SD calibration will try to closely match the target, even if that means deviating significantly from the prior mean. Contrarily, if the tolerance on the constant is small while that on the construction amount is large, this implies a poor data source or high variability in the model, combined with specific prior knowledge about the prior value (such as a constant transferred from a similar model in a similar city).

Secondly, the relative sizes of the tolerances for different targets indicate the modeler’s relative confidence in those targets. If the same parameter affects several different targets, then SD calibration will search for a value for that parameter such that the modeled quantities are closest to the targets with smaller tolerances.

4. Software setup

SD calibration is performed in a PECAS scenario that is already set up to at least run SD for one year. The standard sdcalib.py script must be copied into the scenario’s root directory, and the calibration-specific properties set in the sd.properties file (see Section 5.4.3). Then a calibration run can be started by running sdcalib.py from the command line.

The features and settings described in this document are available starting in PECAS version 2.10, revision 6236. Earlier versions of the PECAS JAR file may be missing some features or behave differently than documented here.

5. Input files

The SD calibration feature requires the usual SD inputs, including the sd.properties file and the SD database. In addition, two comma separated value (CSV) input files specific to calibration are needed, along with some additional CSV files if certain features are used. This section describes the format of each of these files and guidelines on how to construct them. It also details the configuration properties that have been added to the sd.properties file to control the calibration.

5.1. The parameters file

5.1.1. File structure

The parameters file lists the parameters that SD calibration should adjust, and assigns their prior means and tolerances. The parameters are specified using a variable number of columns, depending on the information needed by each parameter type.

These four columns are always required:

- `ParameterType`: A standard code indicating the type of parameter, such as `TransitionConstant` or `IntensityDispersion`; see Section 5.1.2 for a list of valid parameter type codes, their meanings, and the additional columns they require.
- `PriorMean`: The mean of the prior distribution for each parameter, indicating what the modeler believes is the most likely value for that parameter.
- `StartValue`: The starting value or initial guess for each parameter. These are usually either set equal to the corresponding prior value or copied from the current value in the SD database; the latter is necessary if running SD calibration several times in sequence to refine the parameter values.

- `Tolerance`: The tolerance for a parameter represent the standard deviation of the prior distribution around its mean, as an estimate of the "true" value of the parameter. See Section 5.1.3 for more information on how tolerances should be assigned.

The above order is conventional, though the software obeys the headers regardless of order. A parameter can be excluded from the calibration, and therefore not changed from its current value, simply by omitting it from the file.

In addition to the above columns, most parameter types require one or more columns to specify which parameter of that type is intended. If different parameter types need different columns, *all* columns required by at least one parameter type must be included in the file. These columns must still contain values in rows for parameters that do not use them (due to a software limitation), but they are ignored; it is conventional to fill these fields with zeroes.

If specifying correlations between parameters, an extra column *CorrelationGroups* must be included (see Section 5.3).

Table 1 shows part of the contents of an example parameters file, taken from the application of SD calibration to the Alberta PECAS model. In this example, only two space types, light industry (8) and office space (12), along with unimproved parcels (95) are shown, and not all of the possible parameters are included.

Table 1: Example parameters file

ParameterType	From Space Type	ToSpace Type	Prior Mean	Start Value	Tolerance
NewToConstant	0	8	0	394.0	700
NewToConstant	0	12	0	184.8	700
NewFromConstant	8	0	-400	-432.1	700
NewFromConstant	12	0	-400	-524.5	700
NewFromConstant	95	0	85.2	-1049.5	200
TransitionConstant	95	8	0	469.0	700
TransitionConstant	95	12	0	14.2	700
AddConstant	8	0	-400	23.7	700
AddConstant	12	0	-400	623.7	700
RenovateConstant	8	0	-350	575.0	700
RenovateConstant	12	0	-350	1016.7	700
IntensityDispersion	0	8	-4.4	-4.4	0.1
IntensityDispersion	0	12	-4.4	-4.2	0.1
IntensityDispersion	0	95	-4.4	-4.6	0.1

5.1.2. Types of parameters

There are currently 22 parameter types that can be included in the calibration. They fall into five categories: alternative constants, dispersion parameters, density shaping function parameters, transition constants, and local constants. All of these parameters are stored in tables in the SD database, and updated in place as they are calibrated.

Alternative constants are constant utility values that are added to a specific alternative, defining the attractiveness of that alternative with all else being equal. They are the parameters with the simplest effects on the targets: if the renovate constant on office space is increased, then there will be more office renovation. Alternative constants are specific to one type of space. If the *existing* space type on the parcel determines whether the constant applies, the parameter requires a `FromSpaceType` field in the parameters file; if the *new* space type that is being built matters, a `ToSpaceType` field is needed instead. Alternative constants are stored in the `space_types_i` table.

Dispersion parameters sit at the nodes of the tree structure in Figure 1. They control the sensitivity of the alternatives below them to variations in utility from the space price and other local factors. As with alternative constants, they are specific to one type of space, and require a `FromSpaceType` field or a `ToSpaceType` field in the parameters file, depending on whether the existing space type or the new type of space being built determines which constant applies. Also like alternative constants, they are stored in the `space_types_i` table.

Since dispersion parameters cannot be negative, it makes more sense to approximate their prior distributions as *log-normal* rather than normal; therefore, SD calibration always works with the natural logarithm of the dispersion parameter rather than the dispersion parameter itself. Rows for dispersion parameters in the parameters file show the mean, tolerance, and starting value for the logarithm, and all values in the output file that refer to dispersion parameters are based on the logarithm. SD calibration automatically converts back to the real dispersion parameter when writing updated parameter values to the database. For example, the light industry (type 8) intensity dispersion parameter in Table 1 is listed as having a starting value of -4.4; when SD calibration starts, it will write $\exp(-4.4) = 0.012$ to the database as the dispersion parameter value.

Density shaping function parameters control the distribution of space intensities for new developments and additions. As with alternative constants, they are specific to one type of space, and they always require a `FromSpaceType` field in the parameters file. They also require a `StepPoint` field to indicate which part of the density shaping function the parameter controls. They are stored in the `density_step_points` table in the SD database. The density shaping function and its parameters are described in detail in the Density Shaping Function reference.

Transition constants are unique in that they are specific to a *pair* of space types. They are constants that affect the probability of replacing one type of space (specified by the `FromSpaceType` field) with another type of space (specified by the `ToSpaceType` field) as part of a build-new action. Transition constants are found in the `transition_constants_i` table in the SD database.

Local constants are restricted to a specific geographical area, usually a city, town, or neighbourhood. The geographical areas are defined in the `taz_groups` table, while the mappings from zones to geographical areas are defined in the `tazs_by_taz_group` table. Local constants require a `TazGroup` field in the parameters file holding the group number of the geographical area that the constant applies to. Local constants can be specific to a space type – these are found in the `taz_group_space_constants` table – or they can apply to all space types equally within the boundaries of the geographical area – these are found in the `taz_group_constants` table.

The currently available parameter types are described in Tables 2 through 5. The tables show the type code for each parameter type (which is how the parameter must be referred to in the parameters file), the SD database field in which the parameter is stored, and a description of how the parameter affects SD. Note that the type codes and field names are always written in `CamelCase` with no spaces between words. The ability to calibrate additional types of parameters can be added to the software upon request.

Table 2: Types of alternative constants that can be calibrated

Type code	Space type field	Database field	Description
`NoChange Constant`	`FromSpace Type`	`no_change_ transition_ const`	Constant utility for the no-change alternative
`Demolish Constant`	`FromSpace Type`	`demolish_ transition_ const`	Constant utility for the demolish alternative
`Derelict Constant`	`FromSpace Type`	`derelict_ transition_ const`	Constant utility for the derelict alternative
`Renovate Constant`	`FromSpace Type`	`renovate_ transition_ const`	Constant utility for the renovate alternative on *active* parcels
`Renovate Derelict Constant`	`FromSpace Type`	`renovate_ derelict_ const`	Constant utility for the renovate alternative on *derelict* parcels
`AddConstant`	`FromSpace Type`	`add_ transition_ const`	Constant utility for the add-space alternative
`NewFrom Constant`	`FromSpace Type`	`new_from_ transition_ const`	Constant utility for the build-new node on parcels with the given *existing* space type
`NewToConstant`	`ToSpace Type`	`new_to_ transition_ const`	Constant utility for build-new alternatives with the given *new* space type
`CostModifier`	`ToSpace Type`	`cost_ adjustment_ factor`	Constant added to the construction cost for build-new and add-space alternatives with the given *new* space type

Table 3: Types of dispersion parameters that can be calibrated

Type code	Space type field	Database field	Description
`TopDispersion`	`FromSpace Type`	`nochange_ dispersion_ parameter`	Top-level dispersion parameter
`Change Dispersion`	`FromSpace Type`	`gk_ dispersion_ parameter`	Dispersion parameter at the change node
`Decay Dispersion`	`FromSpace Type`	`gw_ dispersion_ parameter`	Dispersion parameter at the decay node (between the demolish and derelict alternatives)
`GrowDispersion`	`FromSpace Type`	`gz_ dispersion_ parameter`	Dispersion parameter at the growth node (between the renovate, add-space, and new-space alternatives)
`Build Dispersion`	`FromSpace Type`	`gy_ dispersion_ parameter`	Dispersion parameter at the build node (between the add-space and new-space alternatives)
`NewType Dispersion`	`FromSpace Type`	`new_type_ dispersion_ parameter`	Dispersion parameter at the build-new node (between the alternatives for building different space types)
`Intensity Dispersion`	`ToSpace Type`	`intensity_ dispersion_ parameter`	Dispersion parameter for the continuous logit models that determine the quantity of space added when the add-space or build-new alternative is chosen.

Table 4: Types of density shaping function parameters that can be calibrated

Type code	Database field	Description
`StepPoint`	`step_point_ intensity`	The FAR at a given step point in the density shaping function (p_i)
`AboveStepPoint`	`slope_ adjustment`	The adjustment to the slope for FARs above the given step point (m_i)
`StepPoint Amount`	`step_point_ adjustment`	The absolute adjustment to utilities for FARs above the given step point (a_i)

Table 5: Other types of parameters that can be calibrated

Type code	**Database table**	**Database field**	**Description**
`Transition Constant`	`transition_constants_i`	`transition_constant`	Transition constant for the new-space alternative to build one given space type overtop of another
`TazGroup Constant`	`taz_group_constants`	`construction_constant`	Constant for building any space in the given zone group
`TazGroup Space Constant`	`taz_group_space_constants`	`construction_constant`	Constant for building a specific type of space in the given zone group

5.1.3. Parameter file guidelines

How should the prior distribution for a parameter be set?

- If there is strong prior information, such as a parameter value from a previous model or a sound theoretical reason for the parameter to have a particular value, then that value should be used as the prior mean and the prior tolerance should be relatively small. Getting the tolerance exactly right is not critical – Bayesian updating can withstand inaccurate prior distributions within reason. However, it is worth expending effort to make the tolerance reflect the true degree of uncertainty about the prior value. The tolerance represents the standard deviation of the prior distribution, and about 70% of the normal distribution is within one standard deviation of the mean. Therefore, set the tolerances so that you expect about 70% of prior means to differ from the true parameter values by less than their tolerance.
- If there is no strong reason to set the prior mean to any particular value, then the prior mean should be an “educated guess”, and the tolerance should be relatively large. The aim here is to create a prior that is “non-informative” under abundant data; any reasonable value of the parameter should differ from the mean by less than the tolerance. For example, if a constant could reasonable be anywhere from 200 to 1000, depending on the local conditions in the model region, then set the prior mean to 600 and the tolerance to 400. Given such a tolerance, SD

calibration will generally defer to the local data instead of allowing an educated guess to sway its results. However, if there is not enough relevant local data to establish an empirical value for the parameter, then the prior distribution will prevent extreme observations in the data from setting the parameter to, say, 5000 or -3000.

5.2. The targets file

5.2.1. File structure

The targets file lists the output measures that the SD model should try to match, and assigns tolerances to each one. Like the parameters file, it has a variable number of columns depending on the information needed by each target type.

These three columns are always required:

- `TargetType`: A standard code indicating the type of target, such as `TotalBuilt` or `AverageFar`; see Section 5.2.2 for a list of valid target type codes, their meanings, and the additional columns they require.
- `TargetValue`: The target amount for this output measure, indicating what was actually developed according to available data.
- `Tolerance`: The tolerance for a target represents the standard deviation of the uncertainty distribution around the target value, combining measurement error and sampling error. See Section 5.2.5 for more information on how tolerances should be assigned.

The above order is conventional, though the software obeys the headers regardless of order. If a target is not included in the file, then SD will not take that target into account when updating the parameters.

In addition to the above columns, most target types require one or more columns to specify which target of that type is intended. If different target types need different columns, *all* columns required by at least one target type must be included in the file. These columns must still contain values in rows for targets that do not use them (due to a software limitation), but they are ignored; it is conventional to fill these fields with zeroes (for numerical columns) or "None" (for string columns).

If specifying correlations between targets, an extra column *CorrelationGroups* must be included (see Section 5.3).

Table 1 shows part of the contents of an example targets file from the application of SD calibration to the Alberta PECAS model. Only two space types, light industry (8) and office space (12), are shown, and not all of the possible target types are included.

Table 6: Example targets file

TargetType	SpaceType	TazGroup	TargetValue	Tolerance
TotalBuilt	8	0	1955138	195514
TotalBuilt	12	0	1026624	102662
Addition	8	0	215445	21545
Addition	12	0	0	1000
Renovation	8	0	1172615	117262
Renovation	12	0	1228445	122845
AverageFar	8	0	0.1366	0.005
AverageFar	12	0	0.1941	0.005
TazGroupTotalBuilt	8	46	644063	86985
TazGroupTotalBuilt	12	46	68848	86985
TazGroupTotalBuilt	8	98	8845	51331
TazGroupTotalBuilt	12	98	61468	51331

5.2.2. Types of targets

There are currently 10 types of targets that can be set for SD calibration to try to meet. Most targets are specific to one or more space types. If the target applies to one space type, use the `SpaceType` column to specify that type; if it applies to multiple space types, define a group for those space types (see Section 5.2.3).

Table 7 shows all of the target types that can be included in the targets file and a description of which kinds of development events SD calibration counts towards the target. The list of target types has grown over time as the need has arisen, and further types can be added to the software upon request.

Table 7: Types of available calibration targets

Type code	Fields required	Associated modelled value
`TotalBuilt`	`SpaceType` or `SpaceTypeGroup`	The total amount of the given space type(s) built on parcels in the entire region, including both additions and new constructions
`Redevelopment`	`SpaceType` or `SpaceTypeGroup`	The total amount of the given space type(s) built on *non-vacant* parcels in the entire region, either through additions or as part of build new actions that required demolishing space first
`Renovation`	`SpaceType` or `SpaceTypeGroup`	The total amount of space of the given space type(s) renovated in the region
`Addition`	`SpaceType` or `SpaceTypeGroup`	The total amount of space of the given space type(s) added in the entire region through addition actions only
`Demolition`	`SpaceType` or `SpaceTypeGroup`	The total amount of space of the given space type(s) demolished in the region, including space demolished as part of a build new action to make way for the new construction
`AverageFar`	`SpaceType` or `SpaceTypeGroup`	The average floor area ratio (FAR) of space of the given space type(s) built as new constructions in the entire region; gives equal weight to each parcel, regardless of the size of that parcel
`TazTotalBuilt`	`TazNumber` AND `SpaceType` or `SpaceTypeGroup`	The total amount of space of the given space type(s) added to parcels in the specified TAZ, including both additions and new constructions
`LuzTotalBuilt`	`LuzNumber` AND `SpaceType` or `SpaceTypeGroup`	The total amount of the given space type(s) added to parcels in the specified LUZ, including both additions and new constructions
`TazGroupTotalBuilt`	`TazGroup` AND `SpaceType` or `SpaceTypeGroup`	The total amount of space of the given space type(s) added to parcels in the specified TAZ group, including both additions and new constructions; the TAZ groups are those defined in the `taz_groups` and `tazs_by_taz_group` tables in the SD scenario schema
`CustomTazGroupTotalBuilt`	`TazGroup` AND `SpaceType` or `SpaceTypeGroup`	The total amount of space of the given space type(s) added to parcels in the specified “custom” TAZ group, including both additions and new constructions; custom TAZ groups are defined outside the database in a separate file (see Section 5.2.4).

Table 8: Targets associated with each event type

Event type	Associated targets
No change	Not counted by any targets
Demolish	If the demolished space is of type i, the amount of removed space is counted by each demolition target that lists type i in its type code.
Derelict	Not counted by any targets
Renovate	If the renovated space is of type i, the amount of renovated space is counted by each renovation target that lists type i in its type code.
Add space	Only the difference between the new quantity and old quantity is ever counted because the old space is unaltered. If the added space is of type i, then this difference is counted towards: • tottarg-i; • redevel-i; • addition-i; • taztarg-i-z if the space is added to a parcel in TAZ z; • luztarg-i-z if the space is added to a parcel in LUZ z; • grouptarg-i-g if the space is added to a parcel in group g. It is also counted towards each multiple-type addition target that lists type i in its type code.
Build new	First, if the build new event occurs on a non-vacant parcel containing space of type j, then that space is demolished and counted towards each demolition target that lists type j in its name. Then the amount of new space (of type i) is counted towards: • tottarg-i; • redevel-i, if the space was non-vacant before the event; • taztarg-i-z, if the space is added to a parcel in TAZ z; • luztarg-i-z if the space is added to a parcel in LUZ z; • grouptarg-i-g if the space is added to a parcel in group g. In addition, the intensity (FAR) of the new development is included in fartarg-i.

It is critical to understand exactly which events each target counts so that the correct real-world events can be included when assigning each target value. Some types of errors can be found and eliminated by looking at target types with overlapping definitions; for example, both the total built and addition targets include add events.

These targets have clear relationships that must be satisfied in a consistent targets file, so any violations are likely to mean that the target definitions have been misunderstood.

The following relationships should always hold:

- A total built target counts everything covered by the corresponding redevelopment target, plus new construction on vacant parcels. Therefore, **a `TotalBuilt` target should always be greater than or equal to the `Redevelopment` target** on the same space types.
- A redevelopment target, in turn, counts everything covered by the corresponding addition target, plus new construction on *non*-vacant parcels. Therefore, **a `Redevelopment` target should always be greater than or equal to the `Addition` target** on the same space types.
- Demolition targets include space demolished as part of a build new action. Therefore, there has to be enough total quantity across all the demolition targets to account for the amount of redevelopment required: **the sum of all `Demolition` targets should be greater than or equal to the sum of all `Redevelopment` targets minus the sum of all `Addition` targets**. This inequality only necessarily holds if there is a target for demolition, redevelopment, and addition for all space types that allow those actions.
- The four spatial target types (TAZ targets, LUZ targets, and the two types of TAZ group targets) have the same inclusion criteria as the total built targets. Therefore, as long as all spatial divisions have a target, **the sum of the targets should equal the corresponding TotalBuilt target**. Similarly, spatial divisions that nest within each other should be consistent; the sum of all TAZ targets in an LUZ should be equal to that LUZ's target, etc.

5.2.3. Defining space type groups

If any targets applicable to more than one space type are used, a space groups file must be provided. A space groups file is a CSV file with two columns: `SpaceType` and `GroupName`. The `SpaceType` column lists each space type number, while the `GroupName` column indicates the name of the group that the space type belongs to.

Table 9 shows an example of a part of a space groups file, taken from the application to the Alberta PECAS model. It shows light industrial space (8) and office space (12) being categorized as `NonResidential`, while single-family detached space (23) and low-density multi-family (26) are categorized as `Residential`.

Table 9: Example space groups file

SpaceType	GroupName
8	NonResidential
12	NonResidential
23	Residential
26	Residential

5.2.4. Defining custom TAZ groups

If any `CustomTazGroupTotalBuilt` targets are used, a custom TAZ groups file must be provided. It must be a CSV file with two columns: `Taz` and `TazGroup`. The `Taz` column lists each TAZ number that should be part of a custom group, while the `TazGroup` column indicates the group number that the TAZ should be assigned to. Note that custom targets are only needed if the targets need to be aggregated at a different level than the TAZ group constants are applied; otherwise, ordinary `TazGroupTotalBuilt` targets should be used.

Table 10 shows an example of a part of the TAZ groups file used for the Alberta PECAS model. It shows neighbourhoods in the cities of Edmonton and Calgary being grouped into custom TAZ groups covering the entire cities (46 for Calgary, 98 for Edmonton).

Table 10: Example custom TAZ groups file

Taz	TazGroup
62401	46
210101	46
220101	98
225801	98

5.2.5. Target file guidelines

The targets are generally compiled from municipal development records, ideally using data from multiple years so that the targets are more representative of typical behavior in the region. If no data is available for a given target, then an educated guess can be made and a wide tolerance assigned, in the same way as can be done with the parameters file (see Section 5.1.3). Alternatively, the target can be omitted from the targets file, causing the corresponding modeled value to be ignored during the calibration.

The tolerance values in the targets file represent uncertainty derived from two sources. Firstly, there is some natural variability from year to year in the amount of development. SD calibration uses expected values within the model to remove the random element that is normally present in SD, but this randomness cannot be removed from the real world. Therefore, this portion of the tolerance reflects how much the amount of development in the years or sub-regions captured by the data source can be expected to reflect the long-run average in the whole region. Secondly, the data collection process itself is not perfectly accurate, so the target data source does not exactly reflect the true pattern of development. This is especially true in the case of fragmentary or absent data, where educated guesses are made and so this portion of the uncertainty dominates.

An alternative interpretation of the target tolerances is that it represents how closely the modeler wants each modeled value to match the corresponding target. The software will expend more effort in matching targets with narrow tolerance values than those with wide tolerance values. These interpretations align because it is more important to closely match stable targets based on solid data than targets that vary wildly or have poor observational support.

In some cases, the target is *zero*; no development events covered by the target appear in the data set. These targets must be handled differently depending on the situation. If such development events are possible but unlikely, then the target should be handled like a poor-quality nonzero target, given a relatively permissive tolerance to allow the prior information (or other targets) to inform the choice of the parameters associated with the target. If, on the other hand, there is theoretical reason to disallow such development events entirely, then this prohibition should be enforced directly by setting the constants on the events to a huge negative number (e.g. -10^{99}) and removing both the target and parameters from the calibration. Including them would only slow down the calibration and clutter the output files with parameters whose values are already known.

The example target file in Table 6 shows the former approach in action. There were no addition events of office space in the San Diego dataset, so the target value is zero. However, there was no theoretical reason preventing the construction of additional office buildings on a parcel already containing office space, so a tolerance of 1000 square feet was assigned. This allows SD calibration to assign a constant that leads to a small but nonzero probability of office additions, enabling SD to respond to scenarios that encourage office additions.

If there were a theoretical reason for prohibiting office additions, this would have been implemented by assigning -10^{99} as the value of the office add constant (addconst-4), removing it from the parameters file, and removing the addition target from the targets file.

5.3. Correlations

While the information in the previous sections is enough for most applications of SD calibration, sometimes additional expressiveness is needed in the prior or target distribution. The *correlations* feature allows not just tolerance values, but also a *linear relationship* between parameters or targets, to be specified in the input files. Both parameters and targets can be given correlations, though the feature has only been used for parameters so far. Correlations between parameters suggest to the calibration algorithm that those parameters should move up together or down together unless there is compelling evidence to separate them. Correlations between targets suggest that those targets convey redundant information and so the calibration algorithm should pay less attention if the corresponding output measures all deviate from their targets in the same way.

To specify correlations, an additional parameter correlations file and/or target correlations file must be provided. These files have two columns: `CorrelationGroup`, which names the group of parameters or targets that should have correlated variations, and `CorrelationCoefficient`, which indicates how strongly the parameters or targets are correlated.

Each correlation coefficient must be nonnegative and less than 1 – correlations equal to 1 will result in a "matrix not SPD" error. A coefficient of zero indicates no correlation, so the parameters should be moved independently, while a coefficient close to 1 indicates a high positive correlation, so when one parameter moves up, the others in the group should also move up.

All of the parameters or targets in a group will be correlated pairwise to each other with the same correlation coefficient. This ensures that the probability distribution remains valid.

In order for the correlations to be applied, the parameters file and/or the targets file must be augmented with a column called `CorrelationGroups`, indicating which correlation group(s) the parameter falls into. If a parameter falls into several correlation groups, list the names of the groups separated by pipe characters ("|"). Note that it is invalid for two parameters to share two different correlation groups, as this would leave ambiguous which of the two correlation coefficients to use between the two parameters.

5.4. The properties file

SD calibration makes use of the same properties file as SD, usually at /AllYears/Inputs/sd.properties relative to the scenario directory. This section lists the properties that a regular SD run accepts, with emphasis on their applicability to calibration, as well as the new properties that have been added specifically to configure the calibration feature. Some properties revert to a default value if absent from the file; these are specified below in the form `PropertyName(=DefaultValue)`. All other properties *must* be present in the file unless otherwise specified.

5.4.1. General SD properties

- `LandJDBCDriver`: The database driver class used by SD to interact with the land database. Set to `org.postgresql.Driver` if using a PostgreSQL database; set to `com.microsoft.sqlserver.jdbc.SQLServerDriver` if using Microsoft SQL Server.
- `LandDatabase`: The URL for the database containing the land data (including parcel data, zoning rules, construction costs, etc.).
- `LandDatabaseUser`: The username for the land database
- `LandDatabasePassword`: The password for the land database
- schema: The database schema; usually set to %SD_SCHEMA% so that the schema can be set from the run script
- `ReadExchangeResults(=true)`: Whether to copy the AA space prices into the exchange_results database table; usually set to true unless external price smoothing is in place
- `SmoothPrices(=false)`: Whether to apply price smoothing to the AA space prices; usually set to false for calibration purposes
- `QueueSize(=5)`: The number of parcel batches that can be queued up waiting for processing by the calculation threads; can be left at the default value for most purposes
- `IgnoreErrors(=false)`: Whether to ignore overflow errors in the logit model. If true, the calibrator will crash on overflow errors; if false, it will log the error, assume no development on the parcel, and continue. Set this to true if there are hard-to-avoid overflow errors on a small minority of the parcels.
- `CapacityConstrained`: Whether to limit SD construction events in each zone to match the quantity of construction activity that AA allocated to that zone; usually set to false for calibration purposes.
- `NumberOfBatches(=250)`: The number of batches of random parcels that the parcel inventory should be divided into. This is ignored if `FetchParcelsByTaz` is true, and its value only makes a difference if `CapacityConstrained` is true; otherwise, it should be left at its default value.

- `FetchParcelsByTaz(=false)`: If true, each batch of parcels read from the database comprises all of the parcels in a TAZ; if false, the parcels are assigned randomly to a number of batches specified by `NumberOfBatches`.
- `MinParcelSize(=400.0)`: The size in square feet of the smallest parcel that SD (and, therefore, the calibration feature) should consider; smaller parcels are ignored and assumed to have no development.
- `MaxParcelSize(=Infinity)`: The size in square feet of the largest parcel that should be processed as a whole; larger parcels will be split into chunks at most as big as `MaxParcelSize` through pseudo-parceling. A value of Infinity disallows pseudo-parceling entirely.
- `AmortizationFactor(=0.0823746504516875)`: The factor used to convert up-front development costs into effective annual costs, based on the time value of money assuming a typical mortgage interest rate
- `UseYearSubdirectories(=true)`: If true, SD adds the current year to `AAResultsDirectory` when reading AA outputs. For example, if `AAResultsDirectory` is “W00”, then SD will read from “W00\2011” in model year 2011.
- `LandInventoryClass(=com.hbaspecto.pecas.land.PostgreSQLLandInventory)`: The Java class that should be used to interact with the land database; replace with `com.hbaspecto.pecas.land.MSSQLServerLandInventory` when using Microsoft SQL Server as the database system.
- `AAResultsDirectory`: The directory in which SD should look for the AA space prices; has no effect if `ReadExchangeResults` is false.
- `LogFilePath`: The path where SD logs its development events; not used by the calibration feature (which does no actual development), but must be present.

5.4.2. Land database table and column names

When SD is interacting with the land database, it uses these properties to determine which tables or columns to query. This can be used if the land database has non-standard table names or to temporarily substitute alternative tables into the model. All of these properties default to the portion of the property name after the last dot if unspecified. For example, the default value of `sdorm.parcels` is `parcels`, while that of `sdorm.parcels.parcel_id` is `parcel_id`.

- `sdorm.parcels`: The name of the main parcel table; column names:
  - `sdorm.parcels.parcel_id`
  - `sdorm.parcels.pecas_parcel_num`
  - `sdorm.parcels.year_built`
  - `sdorm.parcels.taz`
  - `sdorm.parcels.space_type_id`
  - `sdorm.parcels.space_quantity`
  - `sdorm.parcels.land_area`
  - `sdorm.parcels.available_services_code`
  - `sdorm.parcels.is_derelict`
  - `sdorm.parcels.is_brownfield`
- `sdorm.sdprices`: The name of the table containing the space prices from AA; column names:
  - `sdorm.sdprices.commodity`
  - `sdorm.sdprices.luz`
  - `sdorm.sdprices.price`
- `sdorm.sitespec_totals`: The name of the table containing known or postulated future developments; column names:
  - `sdorm.sitespec_totals.space_type_id`
  - `sdorm.sitespec_totals.year_effective`
  - `sdorm.sitespec_totals.space_quantity`
- `sdorm.zoning_permissions`: The name of the table containing the restrictions on development due to zoning; column names:
  - `sdorm.zoning_permissions.zoning_rules_code`
  - `sdorm.zoning_permissions.space_type_id`

  - `sdorm.zoning_permissions.min_intensity_permitted`
  - `sdorm.zoning_permissions.max_intensity_permitted`
  - `sdorm.zoning_permissions.acknowledged_use`
  - `sdorm.zoning_permissions.penalty_acknowledged_space`
  - `sdorm.zoning_permissions.penalty_acknowledged_land`
  - `sdorm.zoning_permissions.services_requirement`

5.4.3. Properties specific to SD calibration

- `EstimationMaxIterations(=1)`: The maximum number of iterations that the calibration feature is allowed to do before it stops and reports its best solution. The software may stop before then if it reaches its convergence criterion. Typical calibrations converge in 20 to 150 iterations, so any maximum iteration value in that range may be reasonable, depending on the available running time and desired accuracy.
- `EstimationConvergence(=1.0E-4)`: The convergence criterion. If every parameter changes by less than `EstimationConvergence` times its tolerance on three successive iterations, the software will stop and report its solution.
- `EstimationParameterFile`: The name of the parameters file (see Section 5.1).
- `EstimationParameterCorrelationsFile`: The name of the parameter correlations file; can be omitted if no parameter correlations are being used (see Section 5.3).
- `EstimationTargetFile`: The name of the targets file (see Section 5.2).
- `EstimationTargetSpaceGroupsFile`: The name of the space groups file (see Section 5.2.3).
- `EstimationTargetCustomTazGroupsFile`: The name of the file defining custom TAZ groups (see Section 5.2.4).
-
- `EstimationStdErrorFile(=stderror)`: The file where the standard error of the estimated parameters is written out. The .csv extension is automatically added to the filename. See Section 6.8 for more detail on this output file.

- `EstimationCalculationThreads`: If this property is specified, SD calibration runs in multi-threaded mode using the specified number of threads, which is faster than single-threaded mode. The number of threads should usually be set equal to the number of cores in the CPU on the machine running the software. If this property is omitted, SD calibration runs in single-threaded mode.
- `InitialLambda(=600)`: The initial value for the λ parameter in equation 4. A large initial value is usually preferred so that SD calibration takes cautious steps at first, moving to more aggressive steps only if the model seems well-behaved. If running SD calibration again starting from the previous run's parameter values, `InitialLambda` should be set to the lambda value from the end of the previous run, as reported in calibration.log.

Further, there are five optional parameters that control where SD calibration writes its output files every iteration: `EstimationParametersFolder`, `EstimationTargetsObjectiveFolder`, `EstimationGradientFolder`, `EstimationHessianFolder`, and `EstimationDerivativesFolder`. SD calibration creates these folders if they do not already exist, and empties them if they already contain files, backing up any existing files first. Omitting any of the properties turns off that particular output feature, but this behavior is mainly for backward compatibility; it is strongly recommended that all of the outputs be turned on to make troubleshooting easier. Sections 6.4 through 6.8 describe the format of these files and what they mean for the calibration.

6. Output files

6.1. Iteration-based filenames

Many of SD calibration's outputs are written as a series of files, one for each iteration, in the same folder. These files are named using the format "<*output-type*><i>.csv". Here, *output-type* indicates whether the file contains parameter values, gradients, etc., while i is the number of the *most recently completed* iteration. For example, the parameter values file written after iteration 12 has the name “params12.csv”. The first iteration is numbered 1.

The parameters, targets, and objective are written *after* each iteration is complete, since they record the algorithm’s accomplishments so far. In contrast, the gradient, derivatives, and Hessian are written *before* each iteration, since they describe properties of the objective function that the algorithm will use during that iteration to decide which parameter values to use next. As a result, in a 30-iteration run, the *params* and *targobj* files are numbered from 1 to 30, while the *gradient*, *derivatives*, and *hessian* files are numbered from 0 to 29.

In addition, whenever an iteration is retried with a larger λ parameter (see Section 3.2), the later attempts at that iteration produce files in the format “<*output-type*><*i*>-<*j*>”, where *j* is the number of times that iteration *i* has already been attempted. For example, if iteration 6 fails to make progress, the files *params6*, *targobj6*, etc. will still be produced, but the second attempt at iteration 6 will yield files *params6-1*, *targobj6-1*, etc. This allows the user to investigate why the first attempt failed and in what way the second attempt is different from the first.

6.2. The event log file

This file, found under the name “event.log” in the scenario’s root directory, is the complete log of the calibration run. Within each iteration, it traces the progress of the run as it steps through the parcels in the region to add their contributions to the modelled values. It also contains a summary at the end of each iteration, showing the current parameter values, target values, and total error. It also reports the final outcome of the calibration; whether it converged, stopped at the maximum iterations, or failed for some other reason.

The main use of the event log file is in troubleshooting program failures. If the program stops early with an error message, the stack trace from the log file can help in diagnosing and solving the problem. For this reason, help requests to HBA staff should always include a copy of the log file. The iteration summaries can also give some indication of how the well the run is performing, though the more detailed CSV-format output files detailed below are generally more useful.

6.3. The calibration log file

This file, found under the name “calibration.log” in the scenario’s root directory, contains only the calibration-relevant output from the event log file. As such, it can be useful for getting a general overview of the calibration progress before searching for more detail in the CSV output files. Of particular note are the reported **lambda** values, which gives an indication of whether the software thinks the calibration run is making progress. Lambda values that get smaller as the run progresses or stay relatively stable mean that the run is still making progress, while increasing lambda values mean that the calibration is “stuck” and cannot find a reasonable next step.

6.4. The parameter values files

The parameter values are written to a series of files in the folder given by the `EstimationParametersFolder` property. Each file records the values as they were after a given iteration; the file name is always *params*<i>*.csv*, where i is the iteration number.

The files are in CSV format with a variable number of columns. The first several columns are copies of the first columns from the *parameters.csv* input file, listing the parameters and their prior means. The remaining two columns are the values calculated for the parameters after the ith iteration (CurValue) and the values *before* the ith iteration (“PrevValue”). The difference between the contents of these two columns indicates how rapidly the calibration was making progress at that iteration. Small differences mean that either the run was close to convergence or that it was “stuck”, unable to find a reasonable next step. These two scenarios can be distinguished by looking at the lambda values in calibration.log.

More sophisticated analysis is possible with the parameter values files. For example, plotting the CurValue column over many iterations allows the trend in parameter movement to be seen. Usually, each parameter should move around rapidly in the early iterations, then level off as the run converges. If the run is not converging, then the plots can reveal which parameter is preventing convergence and in what way; maybe its value is oscillating, or maybe it is drifting endlessly in the same direction. Further analysis can be done on these files in combination with the other types of output files; some of these applications are discussed in later sections.

6.5. The targets/objective files

Each iteration, the current modelled values are written to a file in the folder given by the EstimationTargetsFolder property. These files also contain summary information about the amount of calibration error. Each file has the name “targobj<i>.csv”, where i is the iteration number.

The first four rows of the file are the “objective” portion of the file, giving an overall picture of how well the calibration was doing at the ith iteration. First is the value of the objective function $\boldsymbol{\Phi}(\boldsymbol{c})$ after the ith iteration (labelled “CurObj”); second is the value of $\boldsymbol{\Phi}(\boldsymbol{c})$ *before* the ith iteration (labelled “PrevObj”). Recall that the objective function is simply a weighted sum of the squares of the errors, including the deviation from both the targets and the priors. As such, looking at the difference between CurObj and PrevObj is a quick way of seeing how much progress the calibration made during that iteration. Because of the quasi-Newton approach used by SD calibration, CurObj should *always* be less than or equal to PrevObj – the total error cannot increase from one iteration to the next.

The third and fourth rows show how much the targets and parameters each contribute to the total error. The row labelled “ParamError” gives the sum of the squares of the differences between the parameters and the corresponding prior means, weighted by the prior tolerances: $ParamError = \sum_{j=1}^{n} \frac{\left(c_j - \bar{c}_j\right)^2}{\sigma_{Cj}^2}$. Conversely, the row labelled “TargetError” gives the sum of the squares of the differences between the targets and the corresponding modelled values: $TargetError = \sum_{i=1}^{m} \frac{(\theta_i(\boldsymbol{c}) - \bar{t}_i)^2}{\sigma_{Ti}^2}$. From equation 1, it can be seen that, for a given iteration, CurObj = ParamError + TargetError. Unlike the objective function, neither the parameter error nor the target error necessarily decreases from one iteration to the next (though of course their sum must decrease). It is normal for a calibration run to show improvement in the target error at the expense of the parameter error, as it moves away from the priors to better match the targets.

The remainder of a targets/objectives file gives the current modelled values in a format similar to that of the parameter values files. It has four columns with headers: TargetName and TargetValue copied from the targets.csv input file, CurValue giving the modelled value after the ith iteration, and PrevValue giving the modelled value *before* the ith iteration. Tracing the progression of modelled values from iteration to iteration can give an idea of which targets are holding up a calibration run that is not converging.

6.6. The gradient files

Each iteration, the calibration software produces a file called “gradient<i>.csv”, where i is the iteration number. This file reports the *gradient* of the objective function (i.e. $\boldsymbol{g}(\boldsymbol{c}_i)$ from equation 2). The gradient gives insight into why SD chose the parameter values that it did, so it can help in diagnosing convergence failures and other calibration problems.

A gradient file has two columns with headers. The first column (“Parameter”) contains the names of all the parameters that were calibrated. Each value in the second column (“Derivative”) is the derivative of the objective function with respect to the corresponding parameter. The derivative for a given parameter represents how much the calibration software thinks the objective function will change if it adds 1 to that parameter. This tells the software in which direction it should adjust the parameter, as well as how much progress it can expect to make through such an adjustment. For example, if the derivative for newconst-1 is 3.5, then adding 1 to newconst-1 is expected to add 3.5 to the objective function. Since the aim is to *reduce* the objective function, this indicates that some quantity needs to be subtracted from newconst-1.

The simplest use of the gradient file is a convergence check. At convergence, all of the derivatives in the gradient file should be “approximately zero” – that is, orders of magnitude smaller than they were before the first iteration. For example, suppose that a calibration run converged after 30 iterations. If the derivative with respect to the newconst-1 parameter is 5 in gradient0.csv, but 0.005 in gradient29.csv, then that parameter's convergence is probably acceptable. If instead the derivative in gradient29.csv is 2, then the EstimationConvergence property may need to be tightened. It is up to the user's judgment how to balance convergence precision with calibration run time.

The gradient file is also useful if the model is not converging within the number of iterations specified. The parameters whose derivatives have the largest magnitude are likely to be the ones causing the convergence problems, and so they are the ones that should be investigated further.

6.7. The derivatives files

The derivatives files, produced under the name "derivs<i>.csv", report the derivative of every modelled value with respect to every parameter value. This corresponds to the $\boldsymbol{J}(\boldsymbol{c})$ matrix from equations 3 and 4. It gives information about which parameters affect which modelled values, and therefore which parameters would need to be modified to improve the match to each target.

Each derivatives file is in a rectangular table format. The parameter names run across the top row, accompanied by their prior mean values in the second row. The target names run down the first column, with the corresponding values in the second column. The remainder of the file consists of derivatives; each number is the derivative of its row's modelled value with respect to its column's parameter. It represents how much the calibration software thinks the modelled value will change if it adds 1 to that parameter. For example, suppose that the first parameter is newconst-1 and the first target is tottarg-1. These would reside in the third column and third row, respectively, since the headers take up the first two rows and columns. If the value in the third row and third column is 100, then adding 1 to newconst-1 is expected to add 100 to tottarg-1.

Consulting the derivatives files helps when the calibration appears to be adjusting a parameter in the “wrong direction”. Such situations are sometimes caused by secondary dependencies from unexpected targets. In one instance, the expected FAR of space type 4 was too low; to correct this, SD calibration should have been increasing above-4 to make high-density construction more desirable. Instead, it was decreasing above-4 every iteration, making the FAR even lower. It turned out that above-4 also influenced the redevelopment target (redevel-4), since making high-density construction more desirable also makes construction *in general* more desirable. The tolerance on the redevel-4 target was too small, making what should have been a secondary dependency the dominant force on the above-4 parameter. The solution was to increase the tolerance on the redevel-4 target.

In conjunction with equation 3 and the other output files already mentioned, the derivatives file allows for tracing the calculation of the gradient, determining which targets are contributing to each gradient value. This is useful if some of the gradient values appear to have the wrong sign or an unusual magnitude.

6.8. The Hessian and standard error files

The Hessian files, produced under the name “hessian<i>.csv”, report the Hessian matrix of second derivatives of the objective function with respect to each pair of parameters. The Hessian represents the nature of the “curvature” of the objective function, which is hard to interpret meaningfully in the context of SD calibration. It can, however, be used in conjunction with equations 2 and 4 to trace the full calculation of the next iteration step if it appears to be moving in a direction other than down the gradient.

The format of the Hessian is as a square CSV table. The parameter names run across the top row as column headers, and also along the first column as row headers. Each entry in the table is the second derivative of the objective function with respect to the parameters at the head of its row and column. Since the order in which one takes partial derivatives does not matter for “smooth” functions, the table is symmetric about its main diagonal; interchanging rows and columns has no effect on the table.

Of more direct relevance to the modeler is the standard error file. This file is produced only at the *end* of the entire calibration run, rather than every iteration, under the name “stderror.csv”. It summarizes the statistical properties of the final parameter values, allowing the modeler to judge how reliable the calibration results are.

Like the Hessian files, this file is in a square format with the list of parameter names as both the column and row headers. Along the main diagonal (from upper-left to lower-right) are the standard deviations of the calibrated parameters. Each of these values indicates how accurate the software believes its estimate of that parameter to be, with smaller values indicating greater accuracy. For example, if the calibrated value of newconst-1 is 150 while the standard error file reports a standard deviation of 5 for newconst-1, then the value of the build-new constant for space of type 1 can be reported as 150 ± 5.

The entries off the main diagonal are the correlation coefficients between the parameters. Each entry is the degree of correlation between the parameter at the head of its row and the one at the head of its column. These values range from -1 to 1, and reveal linear relationships between the uncertainties in the parameters; a value of 0 indicates no correlation. For example, if two parameters both have large standard deviations, but their correlation coefficient is 0.9, then the software is reporting that even though it is not confident that those parameters’ calibrated values are individually correct, it strongly believes that their values are the same or differ by a constant. Such an outcome would suggest that the behaviors represented by those parameters are really the result of the same underlying factor.

Mathematically, the contents of the standard error file are derived from the standard error matrix $\boldsymbol{\Sigma}_s$, which is related to the final Hessian matrix as follows:

$$\boldsymbol{\Sigma}_s = 2\boldsymbol{H}^{-1} \qquad (5)$$

The standard error matrix is a *variance-covariance* matrix, rather than the *deviation-correlation* matrix needed for the file. It is transformed into the correct format using the equations for the standard deviation σ_i and correlation coefficient ρ_{ij}, namely:

$$\sigma_i = \sqrt{\sigma_i^2} \quad (6)$$

$$\rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j} \quad (7)$$

where σ_i^2 is the variance of parameter i and σ_{ij} is the covariance between parameters i and j.

This transformation is done because the standard deviation and correlation coefficient are easier to interpret than the variance and covariance.

7. Troubleshooting

7.1. The software stops with a “matrix not SPD” error

This indicates that either the parameter variance matrix ($\mathbf{\Sigma}_C$) or the target variance matrix ($\mathbf{\Sigma}_T$) could not be inverted. The most common cause of this is a zero or near-zero tolerance on a parameter or target. All tolerance values must be positive, and should be a significant fraction of the target value itself. There are two ways of changing a zero tolerance:

- If there is uncertainty in the target or parameter value, then the tolerance should be set to reflect that uncertainty. For targets, this usually involves setting a minimum tolerance value. If a target is so small that its tolerance would otherwise be set *below* the minimum, the tolerance is set equal to the minimum instead.
- If there is *no* uncertainty in a parameter (e.g. the value is forced by policy), then the parameter should simply be removed from the calibration by deleting its entry from the parameters file. The parameter should be set manually to its prior value and the calibration attempted again. Targets with no uncertainty can be handled by setting their controlling parameters manually. Such targets can occur when they describe “impossible” events; for example, the total demolished target for a given space type might be zero because demolition of that space is disallowed in all zones. In this case, the “demolition” and “transition-from” constants for that space type should be manually set to very small numbers (e.g. -10^{99}) and removed from the calibration, along with the demolition target.

7.2. The calibration process fails to converge

Sometimes, failure to converge is simply a result of not allowing enough iterations in the SD properties file. In other cases, it means that the algorithm is stuck due to problems in the input files or bad starting values for the parameters.

- Check calibration.log for reports of the **lambda** value at each iteration. If the lambda value is very large (e.g. over 100) and not decreasing consistently, then the algorithm is struggling to make progress.