<Insert title page here!>

# Introduction

The Expected Value Bayesian Estimation process was developed to address several concerns in calibrating a model to match targets:

## Problem 1

When running a microsimulation model, the model uses the random number generator to select particular options from probability distributions. This is done so that the individual agents in the model have specific behaviour that can be tabulated and investigated, and to support more realistic probability distributions. This results in different model outputs from identical runs with the same inputs. When the model is compared with target data, it could be lower on one run and higher on another run, and perhaps much different on a third run because of an extreme random number. This makes it difficult to determine whether the model is accurate or not.

This problem is addressed in the new process by pre-specifying the target values, and calculating the Expected Value of each agent contributing to each target as the probability is processed. The Expected Value of the model output is compared to the target, and this Expected Value is the same for repeated identical runs. It is necessary to pre-specify the targets, since microsimulation models have a near infinite number of possible targets.

## Problem 2

In a complex modelling system like SD the relationship between parameters and targets is not necessarily clear. In general, there is one parameter that has the most effect on one target, so if the model does a poor job of matching that target the single parameter can be adjusted, and sometimes there is a simplified mathematical relationship that can guide the parameter adjustment. However, the parameter likely affects many other targets, and changing the parameter to try to match one target can throw off the model in other targets. In SD estimation, for instance, trying to increase the amount of one type of development will generally decrease the amount of other types of development and also increase the expected intensity (FAR) of development.

This problem is addressed in the new process by calculating a partial derivative matrix that describes the current relationships between each parameter and each target. This matrix encapsulates the primary relationships and all of the secondary relationships in one mathematical structure, which can be manipulated to determine a set of consistent parameter changes that help the model better match a set of targets.

## Problem 3

Modelling involves analyst experience and bringing in as much information as possible, sometimes more information than is available in the local data. For instance, it is very common for a modelling analyst to insist that a parameter have a particular sign, or that two parameters have a specific ratio within a range (e.g. “The value-of-time should be between 10 and 30 dollars per hour”, or “high speed rail should have a similar ride-time parameter to conventional rail”). Sometimes, if a model parameter cannot be established appropriately with available data, parameters are “borrowed” from another model (another city or region) or established by “engineering judgement”.

In estimation theory, this information from experts and previous work is called a “Bayesian Prior”. There is "prior information” that should be considered in establishing the parameter values; the parameter values should not be established only with the available local data.

The PECAS SD module is based on real estate theory, which itself provides some particular assumptions about developer behaviour, like “developers shouldn’t care (much) whether they demolish a old-folks home or a school when they build a new highrise”, and “when comparing capital costs to operating costs, developers should have a future discount rate similar to the current prime interest rate”. These assumptions from real-estate theory are also contributions to the “Bayesian Prior”.

Sometimes this prior distribution is quite important because local data does not provide adequate information; traditionally this is discovered by attempting an estimation, and having it fail in some way, with failure to converge or inappropriate parameters (e.g. incorrect signs that violate the prior). After the estimation fails additional assumptions are added until the estimation process succeeds.

This problem is addressed in the new process by incorporating the prior distribution of parameters as an input file. The analyst can input values from theory, from experience, or from other models as a list of prior values, standard deviations reflecting the confidence in the prior values, and a correlation matrix describing the relationships between the prior values. The process will determine parameters that best match the targets without deviating too much from the prior values, with “too much” being defined mathematically with the standard deviations and correlations. Thus the prior information is incorporated into the mathematical automated parameter search process, rather than being incorporated in an ad-hoc manner by the analyst after the parameter search process fails to converge or fails to produce parameters that match expectations.

## System Overview

The Expected Value Bayesian Estimation process has been implemented for the PECAS SD Model. It is a software extension that allows the SD model to process the same set of parcels it would normally process and in the same order, but instead of simulating development it calculates the matrix of partial derivatives describing the relationship between parameters and Expected Values from the model that will be compared to targets. It does this repeatedly, using a search process to adjust the parameters in a way that best matches both the targets and the prior information. The inputs to the process are the full set of inputs to the SD model, as well as the prior distribution for the parameters and the target values and the confidence in the target values (also expressed as a matrix, so that target values have their own standard deviation as well as a correlation matrix that can express confidence in relationships between targets.) The outputs to the process are a new set of parameters and a report of the goodness-of-fit to each target.

# The SD model

## Model structure

On each land parcel, a multilevel logit model is used to decide which action to apply to the parcel. The possible actions are:

* No change – floorspace is not modified, buildings age by one year, vacant parcels stay vacant.
* Demolish – the parcel becomes vacant. Parcels that are already vacant cannot be demolished.
* Derelict – floorspace is not modified, but the building owner stops paying maintenance and collecting rent. Vacant parcels cannot be derelicted.
* Renovate – floorspace is not modified, but the building age is reset to the current year. Vacant parcels cannot be renovated.
* Add – extra floorspace (of the same type) is added to the existing floorspace. Vacant parcels cannot have space added.
* New – an entirely new building, of any type allowed by the zoning rules, is built on the parcel. If there is already space on the parcel (i.e. it is non-vacant), the existing space is demolished first.

The multilevel logit model that organizes these alternatives has the structure shown in Figure 1. There is one “new space” alternative for every spacetype that the zoning rules allow to be built on the parcel.

Root node

Change node

Growth node

Decay node

Build node

Build-new node

No-change alternative

Renovate alternative

Demolish alternative

Derelict alternative

Add-space alternative

Type 1

Type 2

etc.

New-space alternatives

**Figure 1: SD development alternative model structure**

## Density shaping function

The alternatives that entail adding space to the parcel – the Add-space alternative and all New-space alternatives – require a further decision: what quantity of space should be added? This decision is represented using a continuous logit model. The density shaping function gives the utility of adding space as a function of the amount of space. It is a piecewise linear function, with the function height and slope changing discontinuously at zero or more *step points*; the minimum and maximum possible FAR are also considered step points, since at these values the utility discontinuously becomes negative infinity. The function is specified by the following parameters (where is the number of step points):

* to : the locations of the step points, i.e. the FAR values at which the utility changes abruptly.
* : the base level slope (utility per unit space), composed of variable costs and benefits such as construction cost and rent.
* to : the adjustments to the slope for each interval; the slope of the interval between step points and is .
* : the base level constant utility, composed of fixed costs and benefits such as development fees.
* to : the sizes of the discontinuities at each step point; the difference between the utility immediately to the right of step point and that immediately to the left of it is . The adjustment is applied at the minimum FAR (step point ), so the utility at this point is .

In the following example, these parameters take the values

The resulting density shaping function and the associated probability distribution are shown in Figure 2.

**Figure 2: Density shaping function example**

# Using the calibrator

## The parameters

There are currently 19 parameter types that can be included in the calibration. They fall into four categories: alternative constants, dispersion parameters, density shaping function parameters, and transition constants.

Each parameter type listed below represents several parameters, usually one parameter for each spacetype (including vacant or protected spacetypes). For example, the constant type , the No-change constant, encapsulates the actual parameters for the No-change constant on parcels of spacetype 1, for the constant on parcels of spacetype 2, etc. The corresponding names in the input files would be ncconst-1, ncconst-2, etc.

**Table 1: Parameters that can be calibrated**

|  |  |  |
| --- | --- | --- |
| Symbol | Name | Description |
|  | ncconst- | Constant utility for the No-change alternative on parcels whose existing spacetype is |
|  | democonst- | Constant utility for the Demolish alternative on spacetype |
|  | drltconst- | Constant utility for the Derelict alternative on spacetype |
|  | renoconst- | Constant utility for the Renovate alternative on *non-derelict* parcels whose existing spacetype is |
|  | rendconst- | Constant utility for the Renovate alternative on *derelict* parcels whose existing spacetype is |
|  | addconst- | Constant utility for the Add-space alternative on spacetype |
|  | newconst- | Constant utility for the Build-new node on parcels whose existing spacetype is ; added to the composite utility of all the New-space alternatives rather than each individual alternative |
|  | topdisp- | Top level dispersion parameter on spacetype |
|  | chdisp- | Dispersion parameter at the Change node on spacetype |
|  | dddisp- | Dispersion parameter at the Decay node (between the Demolish and Derelict alternatives) on spacetype |
|  | randisp- | Dispersion parameter at the Growth node (between the Renovate, Add-space, and New-space alternatives) on spacetype |
|  | andisp- | Dispersion parameter at the Build node (between the Add-space and New-space alternatives) on spacetype |
|  | typdisp- | Dispersion parameter at the Build-new node (between the alternatives for building different spacetypes) on spacetype |
|  | intdisp- | Dispersion parameter for the continuous logit models that determine the quantity of space added when the Add-space or Build-new alternative is chosen. |
|  | step- | The step point in the density shaping function for new space of type |
|  | below- | The adjustment to the slope for FARs *below* the step point, in the density shaping function for new space of type |
|  | above- | The adjustment to the slope for FARs *above* the step point, in the density shaping function for new space of type |
|  | stepamt- | The size of the step point in the density shaping function for new space of type |
|  | trans-- | Transition constant for the New-space alternative to build space of type on a parcel that currently contains space of type |

## The targets

There are currently four types of targets that the model can be calibrated to. These are summarized in Table 2. As with the parameters, each target type encapsulates one target for each spacetype, or in some cases one target for each combination of spacetype and zone number.

**Table 2: Calibration targets**

|  |  |  |
| --- | --- | --- |
| Symbol | Name | Description |
|  | taztarg-- | Total amount of space of type added or built new in TAZ |
|  | luztarg-- | Total amount of space of type added or built new in LUZ |
|  | redevel- | Total amount of space of type added to *non-vacant* parcels in the entire region, either by adding space to existing buildings or by demolishing the old building and constructing a new one |
|  | fartarg- | Average FAR of new development of spacetype in the entire region. Note that this average gives equal weight to each parcel, regardless of the size of that parcel. |

## Properties file

For the calibrator to run, a file called “sd.properties” must be present in a directory within the classpath. It must contain the following properties (though if a default value is listed, that property may be omitted if the default value is acceptable):

* LandJDBCDriver: This property is not used by the calibrator, but it must be present and its value must be a valid class name.
* InputJDBCDriver: This property is not used by the calibrator, but it must be present and its value must be a valid class name.
* LandDatabase: The URL for the database containing the parcel data.
* InputDatabase: This property is not used by the calibrator, but it must be present. Use the same value as LandDatabase.
* LandDatabaseUser: The username for the database containing the parcel data.
* InputDatabaseUser: This property is not used by the calibrator, but it must be present. Use the same value as LandDatabaseUser.
* LandDatabasePassword: The password for the database containing the parcel data.
* InputDatabasePassword: This property is not used by the calibrator, but it must be present. Use the same value as LandDatabasePassword.
* UseSQLInputs: This property is not used by the calibrator, but it must be present and its value must be set to true or false.
* UseSQLParcels: This property is not used by the calibrator, but it must be present and its value must be set to true or false.
* QueueSize(=5): The number of TAZs (or random parcel batches) that can fit in the parcel queue, which connects the threads that read the parcel database to the main calculation thread.
* 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: <John and Abdel please explain this>
* NumberOfBatches(=250): If reading parcels randomly, the number of batches of random parcels that the parcel inventory should be divided into. If reading parcels by TAZ, the value is ignored.
* FetchParcelsByTaz(=false): If true, the database readers will step through the list of TAZs, returning all the parcels in a TAZ as a unit. If false, the database readers will assign the parcels randomly to batches, which are returned as a unit.
* MinParcelSize(=400.0): The size in square feet of the smallest parcel that the calibrator 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 undergo pseudo-parceling. A value of Infinity disallows pseudo-parceling entirely.
* AmortizationFactor(=0.0823746504516875): The factor used to convert one-time costs into effective annual costs, based on the time value of money (e.g. the interest rate for a typical mortgage).
* UseYearSubdirectories(=true): If true, the calibrator adds the current year to the AA results directory.
* LandInventoryClass(=com.hbaspecto.pecas.land.PostgreSQLLandInventory): The land inventory class that should be used to read the parcel data. Currently either PostgreSQLLandInventory for a PostgreSQL database or MSSQLServerLandInentory for a Microsoft SQL Server database.
* AAResultsDirectory: The directory in which the Exchange Results table will be found.
* LogFilePath: Must be included but the value is ignored. This path is used in SD for logging development events, but the calibrator does not need it since it does not actually simulate any development.
* EstimationMaxIterations(=1): The maximum number of iterations that the calibrator will do before it stops and reports its solution.
* EstimationConvergence(=1.0E-4): The convergence criterion – if every parameter changes by less than this value on three successive iterations, the calibrator will stop and report its solution.
* EstimationParameterFile: The name and directory of the parameter file.
* EstimationTargetFile: The name and directory of the target file.
* EstimationParameterVarianceAsDiagonal(=false): True if the parameter file should be read in the diagonal-only format, false if it should be read in the full-matrix format.
* EstimationTargetVarianceAsDiagonal(=false): True if the target file should be read in the diagonal-only format, false if it should be read in the full-matrix format.

## Inputs

In addition to the properties file and the SD database, two CSV files are needed for the calibration. These input files contain the following information:

* The list of parameters to calibrate. Parameters not listed will be held at their original values. The current values of the parameters in the database are used as the mean of the prior distribution.
* The prior variance of the parameters.
* The list of targets to try to shift towards their observed values. Targets not listed will be ignored when calculating the objective function.
* The observed values of these targets.
* The variance of the target values.

## The variance matrices

As raw variance matrices can be unintuitive and cumbersome to work with, each input file format specifies a matrix whose diagonal elements are the *standard deviations* of the values, and whose off-diagonal elements are the *correlation coefficients* between the values; we shall refer to such a matrix as a *deviation-correlation* matrix.

There is a simple relationship between the variance matrix and the deviation-correlation matrix: if the standard deviation of parameter is denoted , and if the correlation coefficient of parameters and is denoted , then the diagonal of the variance matrix is found as

and the off-diagonal entries are found as

One benefit of this format is that it is similar to the format used by ALOGIT; the entire off-diagonal portion of the deviation-correlation matrix for the parameters can be copied directly from ALOGIT’s output. Both the parameter matrix and the target matrix can be constructed based on the desired behaviour of the calibrator.

In the simplest case, the deviation-correlation matrix is a diagonal matrix (i.e. all of the correlations are zero); this means that all of the parameters or targets vary independently. The standard deviations in such a matrix can be thought of as “weights” on each parameter or target. If a parameter’s standard deviation is low, the calibrator will put more emphasis on keeping that parameter close to its mean than other parameters with higher standard deviations. Similarly, if a target’s standard deviation is low, the calibrator will put more emphasis on shifting that target’s modeled value to its observed value. The standard deviations also affect the balance between the targets and parameters; if the calibrator is not approaching the targets closely enough, decreasing the standard deviation of all the targets will shift focus from the parameters to the targets.

The correlation coefficients can be used to impose any *linear* relationship on two of the parameters or targets. The ratio between the standard deviations of the variables is the *magnitude* of the slope of the line, while the sign of the correlation coefficient indicates the *sign* of the slope. The magnitude of the correlation coefficient indicates the *strength* of the relationship (i.e. how much weight the calibration puts on holding the variables close to their linear relationship); 0 means there is no correlation, while values close to ±1 indicate a strong correlation. Note that positive and negative 1 themselves are illegal values and will be reported as an error, since a normal distribution cannot represent a *perfect* correlation.

For example, suppose we want to specify that the transition constant from spacetype 1 to spacetype 2 should be about twice the transition constant from spacetype 1 to spacetype 3, with moderate confidence. Then the following values could be used:

Note that the means must be consistent with the desired relationship – since is supposed to be twice , its mean must be twice the mean of .

## Parameter file format

The parameter file, conventionally called parameters.csv, holds the names of the parameters that should be calibrated and the correlation-deviation matrix for the parameters. The first column of parameters.csv is the list of parameters by the names given in Table 1. The remaining columns hold the deviation-correlation matrix. Since this matrix must be symmetric, only the lower half (including the diagonal) is needed, but the upper half can be included if it makes the file easier to produce.

If all of the parameters are independent in the prior distribution, an alternative format can be used. Instead of the full correlation-deviation matrix, only the standard deviation of each parameter (the diagonal of the correlation-deviation matrix) is given. In this format, the file only has two columns: the names of the parameters in the first column and the standard deviations in the second column. If this option is chosen, the property EstimationParameterVarianceAsDiagonal must be set to true in the properties file.

In the following example, there are 2 spacetypes (type 1 and type 2) as well as vacant (type 95). The parameters to be calibrated are the Build-new constant for each type, the Build-new dispersion parameter for each type, and the transition constants. Further, it is specified that all the dispersion parameters should be approximately equal, with high confidence. For convenience, the file is presented in table format. The grey entries are those that do not need to be included because of symmetry.

**Sample parameters.csv file:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| newconst-1 | 49.1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| newconst-2 | 0 | 10.9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| newconst-95 | 0 | 0 | 53.9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| typdisp-1 | 0 | 0 | 0 | 0.1 | 0.9 | 0.9 | 0 | 0 | 0 | 0 | 0 | 0 |
| typdisp-2 | 0 | 0 | 0 | 0.9 | 0.1 | 0.9 | 0 | 0 | 0 | 0 | 0 | 0 |
| typdisp-95 | 0 | 0 | 0 | 0.9 | 0.9 | 0.1 | 0 | 0 | 0 | 0 | 0 | 0 |
| trans-1-1 | 0 | 0 | 0 | 0 | 0 | 0 | 8.63 | 0 | 0 | 0 | 0 | 0 |
| trans-1-2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12.2 | 0 | 0 | 0 | 0 |
| trans-2-1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.59 | 0 | 0 | 0 |
| trans-2-2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9.58 | 0 | 0 |
| trans-95-1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 37.6 | 0 |
| trans-95-2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 49 |

## Target file format

The format for the target file is similar to that for the parameter file, except that the target values must also be given. The first column of targets.csv holds the list of parameters by the name given in Table 2, the second column holds the observed values of these targets, and the remaining columns hold the variance matrix. Again, only the lower half needs to be given, since the matrix is symmetric. As with the parameter file, if all the targets are independent (which is usually the case), only the standard deviations need to be given (as the third column), but the property EstimationTargetVarianceAsDiagonal must be set to true.

In the following example, there are 2 spacetypes (type 1 and type 2) and 4 TAZs (numbered 1 to 4). All the space quantity targets by TAZ and all the redevelopment and FAR targets are to be calibrated against. The targets are all assumed to be independent, and both the full-matrix and diagonal-only formats are shown. Again, the grey entries are those that do not need to be included because of symmetry.

**Sample targets.csv file, full-matrix format:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| taztarg-1-1 | 30 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taztarg-1-2 | 0 | 0 | 1E-6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taztarg-1-3 | 32 | 0 | 0 | 3.2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taztarg-1-4 | 12 | 0 | 0 | 0 | 1.2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taztarg-2-1 | 0 | 0 | 0 | 0 | 0 | 1E-6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taztarg-2-2 | 65932 | 0 | 0 | 0 | 0 | 0 | 6593.2 | 0 | 0 | 0 | 0 | 0 | 0 |
| taztarg-2-3 | 422 | 0 | 0 | 0 | 0 | 0 | 0 | 42.2 | 0 | 0 | 0 | 0 | 0 |
| taztarg-2-4 | 12209 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1220.9 | 0 | 0 | 0 | 0 |
| redevel-1 | 90 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 |
| redevel-2 | 60099 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6009.9 | 0 | 0 |
| fartarg-1 | 0.0003 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3E-5 | 0 |
| fartarg-2 | 0.102 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.0102 |

**Sample targets.csv file, diagonal-only format:**

|  |  |  |
| --- | --- | --- |
| taztarg-1-1 | 30 | 3 |
| taztarg-1-2 | 0 | 1E-6 |
| taztarg-1-3 | 32 | 3.2 |
| taztarg-1-4 | 12 | 1.2 |
| taztarg-2-1 | 0 | 1E-6 |
| taztarg-2-2 | 65932 | 6593.2 |
| taztarg-2-3 | 422 | 42.2 |
| taztarg-2-4 | 12209 | 1220.9 |
| redevel-1 | 90 | 9 |
| redevel-2 | 60099 | 6009.9 |
| fartarg-1 | 0.0003 | 3E-5 |
| fartarg-2 | 0.102 | 0.0102 |

# Bayesian update theory

Let be the event that the model using the parameter vector is accurate, and let be the event that the target values are observed. Then by Bayes’ theorem,

The terms in this equation are defined as follows:

* is the *prior distribution* over the possible parameters. This prior distribution could be any statistical distribution, but in this section and in the software it is assumed to be a multivariate normal distribution. The mean and variance of this distribution can be transferred model parameters, expert judgment, disaggregate maximum likelihood estimation, or some other source. If there are parameters, the mean is , and the variance is , then the prior distribution is given by
* is the *likelihood function*, the probability that a correct model using parameters matches the target values. The model currently only produces a mean, rather than a full probability distribution. Therefore, assume that the *target* values are uncertain, and that the true target values follow a multivariate normal distribution around the actual observed value. The likelihood function is then the probability that the true target values are equal to the modeled values, or

where is the number of targets, is the target values produced by the model using parameters , is the targets, and is the variance of the distribution that the targets follow.

* is the probability of seeing the actual target values. Since these target values did in fact occur, is taken as 1.
* is the *posterior distribution* over the possible parameters, taking into account the evidence provided by the targets. Then the most likely values for the parameters are at the *mode* of the distribution; the task is to find the value of that maximizes

Combining these terms using Bayes’ theorem gives

which can be simplified, without changing the solution, to

The maximum value of can be found by using the Newton optimization algorithm, with modifications suggested by Gauss and Marquardt, as described in Bard 1974.

# Sample run

<insert results from sample run>

# Implementation

The software was written by adding classes to and extending existing classes in the SD packages, all written in the Java programming language. Here, the basic class structure of the calibrator is outlined, and further detail is provided for the classes that handle the parameters and targets, as these are the sections most likely to need extension. Further documentation in the form of Javadocs can be found in the source files.

## Classes

SDEstimation: This is the main class for the calibrator. It reads in the base year and interval from the command line, creates a StandardSDModel, and calls calibrateModel().

StandardSDModel: An extra method is added, calibrateModel(), that reads the inputs using an EstimationReader and sets up and runs the MarquardtMinimizer.

EstimationReader: This interface specifies methods that read in the parameters and targets. The methods are readTargets(), readTargetVariance(), readCoeffs(), readPriorMeans(), and readPriorVariance(). Note that method readTargets() returns the list of EstimationTarget objects with their target values already set using setTargetValue(), so there is no need for a readTargetValues() method. The readCoeffs() method returns a list of Coefficient objects representing the parameters.

EstimationTarget: This is the base class for the targets. See the section “Class structure of the target types” for details.

Coefficient: This is the interface for the parameters. See the section “Class structure of the coefficient types” for details.

MarquardtMinimizer: This class implements the Newton/Marquardt optimization algorithm. It is initialized with an objective function (interface ObjectiveFunction) and an initial guess. Its most important method is iterateToConvergence(), which modifies the parameters iteratively until the objective function reaches its minimum or the algorithm has done the maximum allowed iterations. After iterateToConvergence() returns, several utility methods are available to find information about the results, such as whether the algorithm converged and the current value of the objective function.

ObjectiveFunction: This interface represents an objective function of the kind that can be used in a Newton/Marquardt optimization. The three important methods are getValue(), getGradient(), and getHessian(), which return the current value, first derivative, and second derivative, respectively, for given parameter values.

GaussBayesianObjective: An implementation of ObjectiveFunction, this class is responsible for converting the current parameter values, prior distributions, modeled values (from a DifferentiableModel), and target distributions into an objective function. It also applies the Gauss approximation, which allows it to calculate a “Hessian” matrix for getHessian() using only first derivatives.

DifferentiableModel: This interface represents a model that can produce modeled values at given parameter values, as well as the first derivatives of the modeled values with respect to the parameters. It specifies the two methods getTargetValues() and getJacobian().

ExpectedTargetModel: An implementation of DifferentiableModel, this class is the link between the calibrator and the SD classes. Both getTargetValues() and getJacobian() iterate through the list of parcels, adding up the contribution from each parcel, and return the total.

ZoningRulesI: Extra methods are added here that add the contribution from a single parcel. They defer to the ParameterSearchAlternative methods of the root-level LogitModel for that zoning rule.

EstimationMatrix: This storage class holds the current totals of the modeled values and their derivatives.

ParameterSearchAlternative: All the classes that implemented the interface Alternative in SD (including LogitModel) are retrofitted to implement the subinterface ParameterSearchAlternative as well. This interface adds the methods getUtilityDerivativesWRTParameters(), getExpectedTargetValues(), and getExpectedTargetDerivativesWRTParameters(). The derivatives of the alternative’s utility are needed so that the derivatives of the expected value will propagate up the tree structure properly.

## Class structure of the parameter types

Parameter types are handled by classes that implement the Coefficient interface. The current class hierarchy is shown in Figure 3.

<<interface>>

**Coefficient**

***SpaceTypeCoefficient***

**TransitionConstant**

**NoChange**

**Constant**

**Demolish**

**TransitionConstant**

**etc.**

***DispersionParameter***

**NoChange**

**Dispersion**

**Change**

**Options**

**Dispersion**

**etc.**

**Figure 3: Class diagram for the parameter classes**

Class TransitionConstant is used for the trans-i-j constants, which require two spacetypes: an existing spacetype and a new spacetype. All other constants are defined by only one spacetype, the existing spacetype, and these extend the abstract base class SpaceTypeCoefficient. The dispersion parameters have special behaviour because of the logarithm transformation, so they have their own abstract base class, DispersionParameter.

Each parameter type has its own class (NoChangeConstant, DemolishTransitionConstant, etc.), but since these classes are small, they are written as inner classes within SpaceTypeCoefficient and DispersionParameter. Each parameter class is responsible for its own database access, while the name, spacetype, and transformations are handled by the abstract base classes.

The parameter classes cannot be instantiated directly (the constructors are private); instead, TransitionConstant, SpaceTypeCoefficient, and DispersionParameter have static methods that return the parameter object of the appropriate type. These methods are set up so that if the same method is called more than once with the same parameters, it will return the same object every time. In this way, equality comparisons between parameter objects can be done using object identity, as two parameter objects that represent the same parameter must actually be the same object.

## Adding new parameter types

The existing parameter types are a good starting point for calibration, but additional or different parameter types may be needed. For example, it may be desirable to allow calibrator to adjust the amortization factor, or to allow the density shaping function to have more than one step point, or to rearrange the logit model structure. In any of these cases, the following guidelines will help in modifying the calibrator to accommodate the new parameter types.

1. **Create the class.** Ensure that the new class is written as an inner class within the correct abstract base class: DispersionParameter for dispersion parameters, SpaceTypeCoefficient for all other parameters that apply to a particular spacetype. Parameters that do not fit one of these categories should have their own classes that directly implement Coefficient.
2. **Define a parameter name** that will be used in the parameter file to refer to the new parameter type. This name should consist of a *type code* – a string unique to that parameter type – followed by any values that define the specific parameter – such as the spacetype to which it applies – separated by hyphens. The type code should be stored as a string constant in the appropriate abstract base class, or in the parameter class itself if there is no base class.
3. **Implement the methods** specified by the Coefficient interface. The methods are:
   1. getName(): returns the parameter name as defined in step 2. This method is useful for logging.
   2. getValue(): retrieves the current parameter value from the SimpleORM cache.
   3. setValue(): writes a new parameter value to the SimpleORM cache (which will be transferred to the database on the next commit).
   4. getTransformedValue(): retrieves the current parameter value after applying the transformation (e.g. taking the logarithm for dispersion parameters). This is how the calibrator will see the value internally. If the parameter does not have a transformation, this method should simply defer to getValue().
   5. setTransformedValue(): takes a transformed value as an argument, and writes the true value by applying the *inverse* of the transformation to the transformed value (e.g. taking the exponential for dispersion parameters). This is how the calibrator changes the value internally. If the parameter does not have a transformation, this method should simply defer to setValue().
   6. getTransformationDerivative(): returns the derivative of the transformed value (as returned by getTransformedValue()) with respect to the true value (as returned by getValue()). If the parameter does not have a transformation, this method should return 1.
   7. getInverseTransformationDerivative(): returns the derivative of the true value (as returned by getValue()) with respect to the transformed value (as returned by getTransformedValue()). If the parameter does not have a transformation, this method should return 1.

All the methods except getValue() and setValue() are implemented in the abstract base classes, so parameters that extend a base class usually only need to implement those two methods.

1. **Create a static getter method** in the abstract base class, or in the parameter class if there is no abstract base class. This must return the same object every time it is called with the same arguments.
2. **Replace all references** to the parameter value in the development alternatives with calls to the getValue() method of the new Coefficient implementation. The parameter object should be retrieved using the static getter method.
3. **Add the derivatives** with respect to the new parameter. Each alternative that uses the parameter should, in its getUtilityDerivativesWRTParameters() and getExpectedTargetDerivativesWRTParameters() methods, search for the new parameter in the list of parameters and insert the derivative in the corresponding place in the derivative matrix.
4. **Modify the CSV reader** to read the new parameter type from its name.

## Class structure of the target types

The target types are represented by subclasses of the abstract base class EstimationTarget, while the expected values of these targets are calculated by implementations of interface ExpectedValue. The current class hierarchy is shown in Figure 4.

<<interface>>

**ExpectedValue**

***EstimationTarget***

**SpaceTypeTAZTarget**

**SpaceTypeLUZTarget**

**RedevelopmentInto**

**SpaceTypeTarget**

**SpaceTypeIntensity**

**Target**

**ExpectedBuild**

**NewEvents**

**Expected**

**FARSum**

**Figure 4: Class diagram for the target classes**

As shown in the diagram, there are currently four subclasses of EstimationTarget: SpaceTypeLUZTarget, SpaceTypeTAZTarget, RedevelopmentIntoSpaceTypeTarget, and SpaceTypeIntensityTarget, which correspond to the names luztarg, taztarg, redevel, and fartarg respectively. The base class provides methods to store and retrieve the target value as provided in the target file. The main abstract method it specifies is getAssociatedExpectedValues(), which returns the list of ExpectedValue implementations that are used to calculate the EstimationTarget’s modeled value.

The most important method that must be defined in implementations of ExpectedValue is getModelledValueForParcel(). This method takes three arguments: the type of space that was built, the amount of space added, and the amount of space built new. It returns the expected value under those conditions. Effectively, this method is a filter, which decides whether a particular development event counts towards the expected value. Other classes take care of the actual calculations involved in weighting this expected value by its probability and summing over all the parcels.

For example, a SpaceTypeTAZTarget is created with a particular TAZ number and spacetype; its getModelledValueForParcel() method will check that the current parcel is in the correct TAZ and that the added spacetype matches the target spacetype. If both of these conditions are met, the SpaceTypeTAZTarget will return the sum of the added space and new space. Otherwise it will return zero.

Three of the existing target types are simple enough that they implement ExpectedValue themselves. On the other hand, SpaceTypeIntensityTarget is a compound: to find the average FAR of new space of a particular type, the calibrator must add up the expected FAR for that type on all the parcels, then divide this total by the expected *number* of parcels on which the Build-new alternative for that spacetype will be chosen. To accomplish this, SpaceTypeIntensityTarget uses two inner classes that implement ExpectedValue: ExpectedFARSum, which adds up the FARs, and ExpectedBuildNewEvents, which counts the number of times the Build-new alternative is selected.

## Adding new target types

As with the parameters, additional target types may be needed in the future. The following guidelines will help in modifying the calibrator to accommodate the new target types.

1. **Create the class(es).** If the target is simple (i.e. it measures the total development of some kind), only one class should be needed, which should both inherit from EstimationTarget and implement ExpectedValue. If the target is more complex, one subclass of EstimationTarget and at least one associated implementation of ExpectedValue will be needed.
2. **Define a target name** that will be used in the target file to refer to the new target type. This name should consist of a *type code* – a string unique to that target type – followed by any values that define the specific type – such as the spacetype that counts towards the target value – separated by hyphens. The type code be stored as a string constant in the EstimationTarget subclass. Note that any associated ExpectedValue classes should *not* have their own target names!
3. **Implement the abstract methods** specified by the EstimationTarget class. The methods are:
   1. getName(): returns the target name as defined in step 2. This method is useful for logging.
   2. getModelledValue(): calculates the modeled value of the target by combining the values passed to the setModelledValue() methods of the associated ExpectedValue objects. If the target class itself serves as its associated ExpectedValue class, this method may simply return the value passed to setModelledValue() unchanged.
   3. getAssociatedExpectedValues(): returns the list of associated ExpectedValue objects. This method should return the same ExpectedValue objects every time it is called. If the target class itself serves as its associated ExpectedValue class, the target object should simply return a list containing only itself.
4. **Implement the methods** specified by the ExpectedValue interface in each ExpectedValue class. The methods are:
5. appliesToCurrentParcel(): it is permissible for this method to always return true. However, the model may run much faster if this method is implemented to return false in cases where the current parcel cannot possibly contribute to the expected value. For example, this method in SpaceTypeTAZTarget returns false if the current parcel is not in the correct TAZ.
6. setModelledValue(): the partner to getModelledValue() in EstimationTarget, this method is called by the calibrator once it has iterated over all the parcels, passing it the total expected value. It should set a variable that can be read by getModelledValue().
7. getModelledValueForParcel(): Returns the expected value on the current parcel, given the amount and type of space added to the parcel.
8. getModelledValueDerivativeWRTAddedSpace(): Returns the derivative of the expected value on the current parcel with respect to the amount of space added to the parcel.
9. getModelledValueDerivativeWRTNewSpace(): Returns the derivative of the expected value on the current parcel with respect to the amount of new space on the parcel.
10. **Modify the CSV reader** to read the new target type from its name.