

Agile Modeling for Urban and Environmental Systems: The Open Platform For Urban Simulation

Hana Ševčíková^{a,1,*}, Liming Wang^c, Paul Waddell^d, Alan Borning^e

^a*Center for Statistics and the Social Sciences, University of Washington, Box 354322, Seattle, WA 98195-4322, USA*

^b*Puget Sound Regional Council, 1011 Western Avenue, Suite 500 Seattle, WA 98104-1035, USA*

^c*Institute of Urban and Regional Development, University of California, 316 Wurster #1870, Berkeley, CA 94720-1870 USA*

^d*Department of City and Regional Planning, University of California, Berkeley, 228 Wurster Hall #1850, Berkeley, CA 94720-1850, USA*

^e*Department of Computer Science & Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA*

Abstract

The Open Platform for Urban Simulation (Opus) is a general framework for *agile modeling* in dynamic urban and environmental systems. It has been developed as a generalization of UrbanSim to provide a common and productive platform to enable modelers to rapidly develop and refine component models that can be integrated within a model system. The resulting system can then simulate complex interactions within and among the domains of real estate markets, transportation, land cover change, emissions and resource consumption. By leveraging existing open source software libraries and designing a highly modular framework for building, estimating and simulating from models of various types, Opus provides an open source toolbox for agile modeling.

*Corresponding Author. Tel.: +1 206 - 685-8145, Fax: +1 206 - 685-7419

Email addresses: hanas@uw.edu (Hana Ševčíková), lmwang@berkeley.edu (Liming Wang), waddell@berkeley.edu (Paul Waddell), borning@cs.washington.edu (Alan Borning)

Keywords: Urban simulation, open source framework, land use, environmental and transportation modeling, agent-based microsimulation models, model estimation, Opus

1. Introduction

The Open Platform for Urban Simulation (Opus) is a new open source software platform for *agile modeling* of dynamic urban and environmental systems. This paper is intended to provide an introduction to Opus for researchers interested in urban spatial and environmental modeling, and in the software tools with which to do so. Opus emerged as a generalization of earlier software development efforts in support of the UrbanSim model system (Waddell, 2002; Noth et al., 2003; Waddell et al., 2003). Its overarching goal is to create a robust, modular and extensible open source system for developing and applying model components and integrated model systems that can represent complex urban and regional spatial dynamics, including real estate markets, transportation, emissions, resource use, and land cover change. The Opus system is intended to provide a common platform to facilitate the growth and development of the integrated urban and environmental modeling community, just as the R project (Ihaka and Gentleman, 1996) has supported the development of the statistical computing community, the SWARM project has facilitated the growth of the Agent Based Modeling community, the Biogeme software (Bierlaire, 2003) has broadened access to discrete choice modeling, and the PySAL project (Anselin and Rey, 2007) is beginning to support the area of spatial econometrics. In keeping with the spirit of open source software development, Opus avoids re-inventing existing functionality, leveraging these related open source software libraries to maximize the productivity of model developers and users. A preliminary description of Opus is provided in (Waddell et al., 2005); this paper is the first full publication describing the Opus platform and its use.

Model systems for integrated assessment of human-natural systems (Alberti and Waddell, 2000), and for land use-transportation-emissions modeling (Dowling et al., 2005), are emerging rapidly. These integrated model systems are designed to analyze the complex interactions among urban and environmental systems in order to better assess the social, economic and environmental impacts of urbanization and to design and evaluate public

policies to address them. The challenges of developing useful model systems in these domains are substantial (Waddell, 2011), not the least of which is the need for a robust and widely available software platform that contains the necessary functionality to develop models, estimate their parameters, and integrate the models and the data they use into a dynamic simulation system that includes interactions among many urban and environmental processes. Opus was developed to make it much easier to iteratively build and refine such systems.

We use the term *agile modeling* to describe a methodology that supports quick and fluid iteration among all the different phases of modeling, including data preparation, model creation and specification, scenario construction and execution, result analysis, and interaction with policy stakeholders (Kriplean et al., 2010). A key element in supporting this methodology is a software platform that facilitates many of these varied tasks and their coordination. There are software platforms, both open source and proprietary, that partially meet these needs, but none focus on the kinds of spatial and microsimulation models we wish to support. The closest platform to meeting our needs would be R, which has many user-contributed packages for spatial models and discrete choice models. We also note the availability of Biogeme and PySAL (described earlier in this section as well). However, these platforms provide tools to estimate the parameters of individual models of different types, whereas we also want to combine multiple models, and efficiently manage simulations with the resulting model system. There are other platforms that have been developed for agile modeling within specific modeling frameworks, such as SWARM (Luna and Stefannson, 2000) and numerous derivative platforms for Agent-based Models (ABM). There are several integrated model systems for the land use and transportation modeling domain, such as ITLUP (Putman, 1983), TRANUS (de la Barra, 1995), MEPLAN (Echenique et al., 1990). In the domain of demographic microsimulation, full integrated simulation frameworks include DYNAMOD (Abello et al., 2002), and SOCSIM (Hammel and Mason, 1990), and at least one platform that can be described as an agile modeling platform for this domain: LifePaths (Gribble, 2000). None of these existing platforms, however, covers the full scope of our objectives outlined below.

The design goals for the Opus platform are:

- Provide a comprehensive framework to support the development of inte-

grated urban and environmental models, supporting the entire iterative life-cycle of model development and application for high-performance simulation using large datasets;

- Provide efficient methods for implementing microsimulation models using individual agents, such as households and persons, even for large regions;
- Make model development and testing as agile as possible, through modularity, inheritance, templates, and ease of use;
- Make model development more accessible by reducing the need for modelers to have extensive software development expertise, with model templates, an intuitive GUI, and a powerful domain-specific expression language;
- Make it easy for others to extend the system by offering a simple Application Programming Interface (API) for contributed packages.
- Keep the system open and widely accessible by building on platform-independent, open source libraries.

The remainder of this paper is organized as follows. We start with describing the Opus architecture in Section 2. This description includes a higher level discussion of Opus’s capabilities for using and manipulating data, implementing models, combining models into a model system, defining variables and expressions, and creating indicators. Section 3 elaborates on this description, including examples in Python code. In Section 4 we discuss Opus packages that extend the basic Opus core package, such as a graphical user interface and the UrbanSim package. Section 5 summarizes the current status of the platform as well as plans for further development. Readers without programming expertise should still find most of the paper accessible, though some of the material will be more helpful to readers already somewhat familiar with software development in Python or other object-oriented languages.

2. Opus Architecture

Opus is implemented in the Python programming language. This high-level programming language has become widely popular in the modeling and GIS communities, due to its relatively rapid learning curve, development productivity, and the steadily increasing availability of specialized open-source libraries. Opus leverages numerous Python libraries, but central to its computational efficiency is the numpy Python library, which provides fast multi-

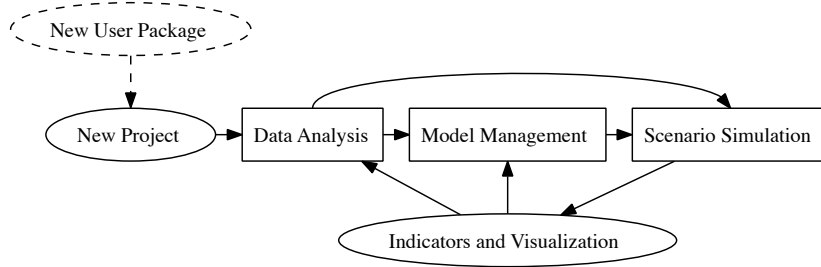


Figure 1: Opus work flow. Model management includes creating, configuring and estimating models; Scenario simulation includes configuring and running a system of models.

dimensional array operations. In addition to leveraging Python libraries, Opus also contains interfaces to external packages, for example, in addition to several pre-defined R-based methods, users have access to all R functions via the Python package `rpy2`. Python is well-known as an excellent ‘glue’ language, making it relatively simple to connect software systems written in other languages, such as C, C++, or Fortran, with Python programs.

The core design for Opus is object-oriented and highly modular, allowing users to easily compose models from small pre-defined modules and, using its inheritance feature, to implement their own modules by overwriting only the necessary part of an existing module without having to implement the whole model. One of the main goals of Opus is to support *agile modeling* (Section 1). From our experience, work flow in a dynamic simulation system includes iterating between analyzing available data, configuring models reflecting processes of the system, estimating model parameters and selecting a preferred specification, then running these models to simulate the evolution of the system over a period of time and analyzing the results, often including the creation of a variety of indicators and their visualization. Opus provides tools supporting modelers in each of these phases, each of which will be discussed in the following sections. Figure 1 shows a typical work flow with Opus. In practice, the process inevitably involves iteration among the steps and experimentation within each step.

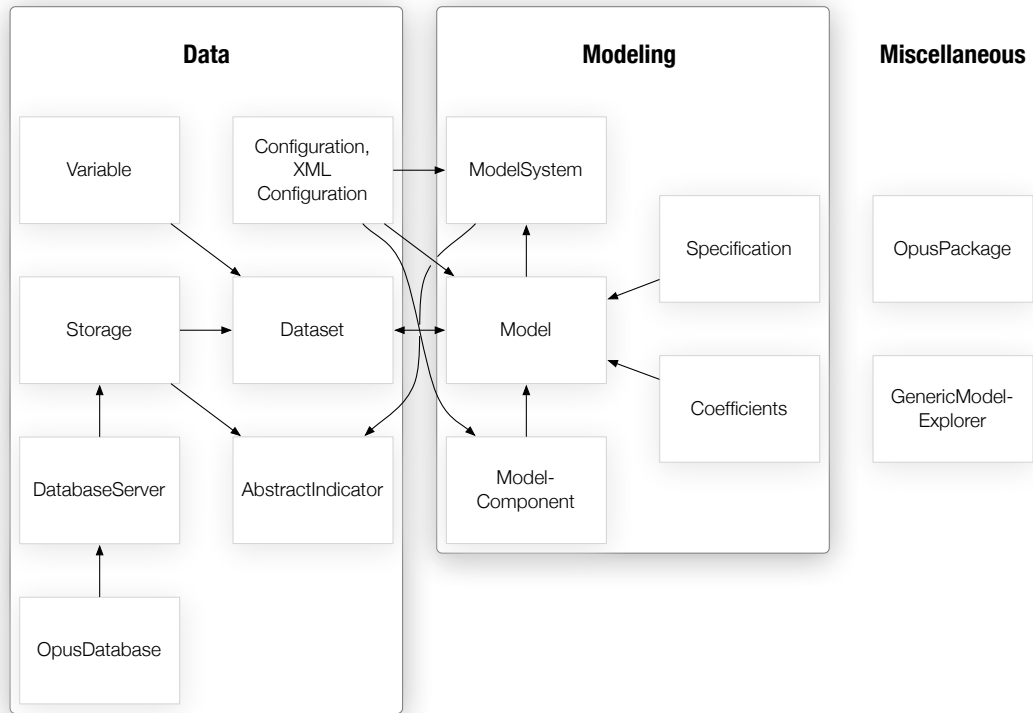


Figure 2: Opus Architecture.

Figure 2 shows a collection of main Opus classes structured into groups that roughly correspond to such work flow: classes for developing and analyzing data, classes supporting developing and specifying models, and a miscellaneous group that supports the creation of an Opus package and interactive exploration of models and data. Object-oriented and modular design allow modelers to experiment with and customize modeling system to fit their needs.

The basic unit of organization in Opus is a *package*. An Opus package is a standard Python package that conforms to a few additional requirements, mainly certain directories and files that define a standard API within Opus. The most general functionality of Opus is implemented in a package called *opus_core*, which is the focus of this paper. It provides general modeling

functionality applicable to a variety of domains. Section 4 describes several extensions to *opus_core*, namely Opus packages that are domain-specific.

2.1. Developing and Analyzing Data

Data is both the beginning and end of modeling. It is a necessary prerequisite to begin building a model, and if a model produces new data, such as predictions, again, data analysis must be performed at the end of the modeling process in order to validate or use the results of the model.

Data is encapsulated in Opus by a Python class called Dataset. A dataset is a collection of attributes for a particular type of entity or object, such as a set of counties, or a set of persons, or a set of buildings. Each member of this set has the same characteristics, such as age of persons or the size of buildings. Conceptually, a dataset can be considered to be a $n \times m$ table where n is the number of entries and m is the number of characteristics, also called attributes. Attributes can be read from a disk-based data store, which we refer to as *primary attributes*, or, computed on-the-fly with Opus variables and expressions or models, which we call *computed attributes*.

When loaded, each dataset attribute is stored in memory as a numpy array, which allows highly efficient computations by performing any arithmetic operations on all elements of the array at once. This feature overcomes a well-known performance bottleneck in Python: iterating over an array or a list of elements.

Opus allows users to read and write data from and to different types of data stores, depending on their needs, while keeping a consistent internal representation of the data. This is handled by subclasses of the class Storage. There are several pre-defined subclasses of Storage, while new ones can be easily incorporated into the system. Currently, users can choose from the following types of storage:

- ASCII, tab or comma delimited (CSV) files;
- SQL storage, which supports various relational database systems such as MySQL and Postgres;
- Dbase formatted files;
- ESRI formatted files;
- Numpy (Python) arrays on disk.

Opus also implements a subclass of Storage for reading and writing data held in main memory using Python dictionaries, which can be useful for smaller data sets.

Each type of storage is implemented as a subclass of the Storage class. Each dataset contains a pointer to an object of the particular storage class, and any I/O operations on the dataset are delegated to this object. A dataset can also use one storage object for reading data and another storage object for writing. Thus, one can use the Dataset class for converting data from one format to another. In addition, we provide tool scripts to help users convert data between most of the data types.

The Opus Dataset class also provides a rich set of functions for data manipulation and analysis, including adding and removing attributes, computing data descriptives (summary, correlation), join, group, plot (scatter plot, histogram, correlation image) or map (for raster data). In combination with variables and expressions, Dataset provides most commonly used functions in data development and analysis. Section 3.1 demonstrates an example of working with Dataset.

2.2. Implementing Models

Models in Opus are implemented in a highly modular way. Researchers should have the flexibility to experiment with different models by exchanging parts of a model using either ready-to-use model components or components developed by the modelers themselves. The collection of the ready-to-use model components can be easily extended as the need arises. Opus also offers a functionality for managing a group of models combined into a model system that can be used for simulating over a period of time (discussed in more detail in Section 2.3).

There is only one requirement for a piece of code to be an Opus model, namely that it should be defined as a class that inherits from the Opus class Model. Additionally, if an Opus model is to be combined in a model system, then it must have a method called ‘run’ which contains the functionality that is to be repeated in each simulated time period. Optionally, if the model is being estimated, the estimation functionality is implemented in a method called ‘estimate’. There are no further requirements for an Opus model.

To make the process of implementing models easier, Opus offers a few commonly used models as well as various plug-in model components (Figure 3).

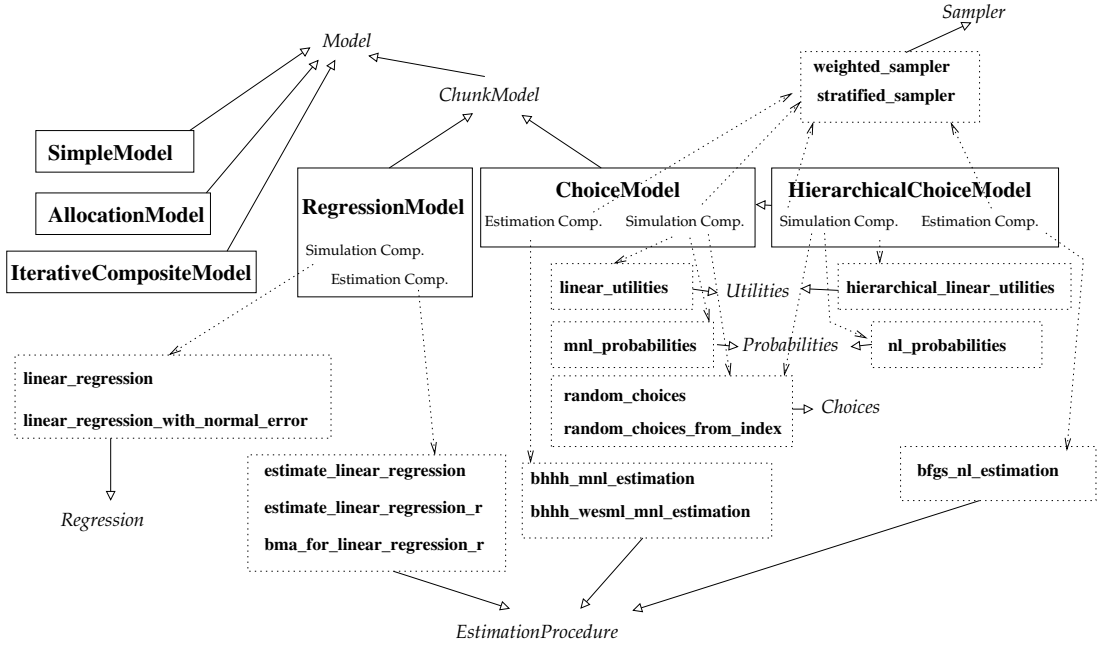


Figure 3: Opus models and model components. The solid arrows show inheritance relationships; the dotted arrows show optional plug-in relationships. Abstract models are shown in italic font. Dotted boxes group components that have the same parent.

Table 1 gives a brief description of these models. The regression, choice, and hierarchical choice models all have a simulation component and an estimation component. Users can choose from an existing set of such components, or implement their own. In the following, we briefly describe the main functionality of the two choice models. Examples will be given in Section 3.

The choice models are used for constructing binary, multinomial, and nested logit models. The theoretical foundation for this approach is the Random Utility Model introduced by Daniel McFadden (McFadden, 1974, 1981), which postulates that agents make choices to maximize their utility, and that their utility is derived from the underlying attributes of choice alternatives. Here, the simulation component consists of three parts: computing utilities, computing probabilities, and making choices. Each of those parts can be again either selected from predefined modules (such as `linear_utilities`, `mnl_probabilities`, `random_choices`), or user-defined modules can be eas-

Model/Comp. name	Description	Example
Model	Generic class that is the superclass of all models. Includes some logging and timing features.	Any of the following models.
Chunk Model	Separates the observations into chunks (possibly in a random order) and handles running the model in a loop for all chunks.	Can be used if the number of observations passed to a model is too large for performing array operations on them as a whole.
Simple Model	Modifies an attribute of a given dataset with results from an expression.	An aging model: pass a dataset of persons and an expression of age+1. In each simulated time period, the model increments the age attribute of persons.
Allocation Model	Allocates a given quantity according to weights while meeting capacity restrictions.	Provides a simple way to proportionally distribute an aggregate quantity, such as visitor population, to locations such as zones, or buildings of type='hotel', based on a weight variable, such as the number of hotel rooms.
Iterative Composite Model	Runs a set of given models until a given condition is reached. It iterates within one time period. It provides a capacity to create two models that are inter-dependent on information from each other, and are intended to iterate until some convergence criterion is met.	Coupling a household location choice model, which depends in part on housing prices, with a housing price adjustment model, which depends in part on housing demand within a time period, which can in turn be computed as an aggregation over all households of their location choice probabilities. In this context, one could couple the household location choice model with a price adjustment model, and run the two as an iterative composite model until the market reaches a stable market clearing outcome.
Regression Model	Framework for combining modules to estimate and simulate a multiple regression model.	Price model.
<code>linear_regression</code>	Returns a linear combination of given data and coefficients.	
<code>linear_regression_with_normal_error</code>	Adds a normally distributed model error with given mean and variance to a linear combination of given data and coefficients.	
<code>estimate_linear_regression</code>	Assumes linearity in model variables and uses Ordinary Least Squares estimation.	
<code>estimate_linear_regression_r</code>	Uses R functions to estimate a linear model.	
<code>bma_for_linear_regression_r</code>	Performs a variable selection using the R package BMA (Raftery et al., 1997).	
Choice Model	Framework for combining modules to estimate and simulate a discrete choice model (page 9).	
Hierarchical Choice Model	Framework for combining modules to estimate and simulate a nested discrete choice model (page 11).	
Samplers	Abstract class that groups modules used to draw samples from a dataset.	
<code>weighted_sampler</code>	Sample observations using weights.	As part of a choice model for sampling alternatives.
<code>stratified_sampler</code>	Sample observations using strata.	As part of a choice model for sampling alternatives.

Table 1: Description of Opus models and model components.

ily plugged in. The `linear_utilities` component computes the systematic part of the utility under the assumption that it is given by a linear combination of the coefficients and predictors. The module `mnl_probabilities` computes probabilities (given utilities) using the multinomial logit formula, a widely used theory that assumes that the random part of the utility is independent identically Gumbel-distributed. Given probabilities, the module `random_choices` selects randomly a choice for each agent. To estimate the choice model, one can choose the module `bhhh_mnl_estimation` which implements the Berndt-Hall-Hall-Hausman maximum likelihood algorithm (BHHH) (Berndt et al., 1974) . An extension to it is implemented in the `bhhh_wesml_mnl_estimation` module, namely weighted endogenous sampling maximum likelihood (Manski and Lerman, 1977), which weights observations before passing them to BHHH, to take into account undersampled or oversampled observations.

The Hierarchical Choice Model is constructed in a similar way as the Choice Model, and is used for applications with hierarchical structure such as the nested logit models. In this case the utilities and probabilities can be computed using the modules `hierarchical_linear_utilities` and `nl_probabilities`, respectively. The model can be estimated using the Broyden-Fletcher-Goldfarb-Shanno (BFGS) optimizations method for nested logit implemented in the `bfgs_nl_estimation` module.

All the modules described in this section and in Table 1 are heavily based on numpy’s efficient matrix operations, and thus the computation is done for all observations (or chunks of observations) at once. Also note in Figure 3 that there is no strict line between models and model components. In fact, all root nodes in the figure, such as Model, Sampler, Regression, and Utilities, are implemented as children of the ModelComponent class (not shown in the figure).

As mentioned above, we define a model being an Opus model if it is a child of the class Model and defines a method called ‘run’. This method should implement a simulation of the model. Optionally, Opus models can also have a method ‘estimate’ that implements an estimation of model parameters. This framework allows estimation and simulation to be performed on the same instance of a model.

If a model is to be estimated, its specification can be managed by the classes EquationSpecification and Coefficients, as for the regression and choice mod-

els. From a given specification object, the ‘estimate’ method estimates the parameters and creates a corresponding Coefficients object, which can be passed to the ‘run’ method. Opus also supports saving estimation results for later use in a simulation session.

A model can have several sub-models, that is, the same piece of code (defined by the ‘run’ or ‘estimate’ method) is applied to different groups of dataset members separately. This allows for the same model to differentiate specifications by different groups of agents without having to take special care of the specific cases – the model applies it automatically. The EquationSpecification and Coefficients classes support this kind of grouping. In addition, in case of the choice models, the specification can distinguish between different equations or nests.

2.3. Configuring Model Systems and Managing Runs

In Opus, one can combine a set of models into a model system that runs iteratively for a given time period. For example, a model system can be configured to run a set of models annually for a total of 30 years. Each model and the relationships between models can be configured using a Python dictionary object (supported by the class Configuration, as shown in Figure 2) or using an XML format (supported by the class XMLConfiguration). In both cases, outputs of one model can be configured as inputs of another model.

Data on disk is stored in a binary format in a *cache directory* so that the model system as well as each model has fast access to all data without having to keep everything in main memory. This also allows models to output datasets into a cache for use by other models. For each time increment (e.g. a year), a new subdirectory is created in order to record results of each iteration. Thus, at the end of the simulation, the base cache directory contains as many subdirectories as there were iterations, each of them containing datasets from that particular time point.

There is also a tool for managing model system runs that archives information about different runs in a SQL database, allowing restarting a simulation at any time point.

2.4. Variables and the Expression Language

Usually, the process of creating new models, calibrating, estimating and results evaluation involves considerable investigation and experimentation us-

ing different model variables that describe attributes of actors, processes and geographies of the simulated environment. In many cases, the original variables must be transformed or combined to create new variables that are more suitable for the analysis.

As mentioned in Section 2.1, a variable (or computed attribute) is considered to be an additional attribute of a dataset, and thus is represented by a numpy array of the same size as there are records in the dataset. Its computation is invoked by the Dataset method `compute_variables(variable_name, ...)` where the syntax of *variable_name* depends on the definition of the variable. There are two ways to define variables in Opus:

1. Using a domain-specific expression language;
2. As a Python module.

In the latter case, the variable is implemented as a class in a single Python file of the same name that is located in a directory of the same name as the dataset name. It is a child of the class `Variable` and has a required method called ‘compute’ that contains the actual computation. The only requirements for this method is that it returns a numpy array of the same size as the dataset to which the variable belongs to. An optional method named ‘dependencies’ should return all variables that this variable depends on. The Opus framework uses lazy evaluation: the ‘compute’ method is only invoked if either the variable has not been computed before, or if the values of one or more of the dependent variables have been updated. The system maintains a version number for each variable to keep track of the bookkeeping for this. A variable defined this way will be referenced by a name composed from its actual name, its dataset name and the particular Opus package. For example, if a user working in an Opus package called ‘my_package’ implements a variable called ‘my_var’ for a dataset of type ‘my_dataset’, the computation of this variable can be invoked by

```
ds.compute_variables(['my_package.my_dataset.my_var'], ...)
```

Here, `ds` is an object of the ‘my_dataset’ class. Details regarding the remaining arguments, and a detailed example, are given in Section 3.1.

Opus includes a domain-specific programming language that, for most variables, eliminates the need to code the variable as a Python module. An expression consists of the name of some existing variable, or a function or

operation applied to other expressions. This definition is recursive, so that a unitary function or binary operator can be applied to expressions composed from other expressions. The unitary functions are any of the functions available in numpy, such as `exp` and `sqrt`. Similarly, all of the numpy operators can be used in Opus expressions, including `+` `-` `*` `/` `**` `<` `>` `==`. An important class of methods that provide special operators beyond those in numpy are used to aggregate and disaggregate variable values over two or more datasets. The ‘aggregate’ method associates information from one dataset to another for a many-to-one relationship, while the ‘disaggregate’ method does the same for a one-to-many relationship. Functions for aggregation include taking the sum, the mean, the min, or the max of the values being aggregated. Analogously, the ‘disaggregate’ method takes information from a coarse set of entities and allocates it to a finer set of entities. Aggregation and disaggregation can be done over multiple levels of geography or other agents in a single expression, for example aggregating the number of jobs from buildings, up to parcels, then up to zones, then to cities.

In order to work with variables that describe the interaction between two datasets, Opus includes a subclass of `Dataset` called `InteractionDataset`. Attributes of this class are stored as two-dimensional arrays. A full range of operators and functions are available for these 2-d interaction arrays as well. For example, the expression

```
income_x_cost = ln(household.income) * zone.average_housing_cost
```

returns an $n \times m$ array, where n is the number of households and m is the number of zones. The $(i, j)^{th}$ element of the array is the log of the i^{th} household’s income times the average housing cost for the j^{th} zone.

Expressions can be passed directly to the ‘compute_variables’ method or to model specifications. They can also be collected into an ‘aliases.py’ file placed in the same directory as other variables implemented as Python modules; then the user can refer to the variable simply by its alias which can be helpful in case of reusing definitions.

See (Borning et al., 2008) and the *Opus/UrbanSim Users Guide and Reference Manual* (2011) for more details on variables and the expression language. The `opus_core` package contains a few examples which can be found in the directory ‘test’.

Finally, the time-incremental structure of the cache directory described in Section 2.3 supports Opus’s framework for working with time series variables. Such variables are defined by simply adding the suffix ‘_lag x ’ to an existing variable name, where x is the number of time increments to move backwards when computing the specific variable. For example, say the computation of the ‘my_var’ from the beginning of this section is a part of a model system that simulates for five years, from 2010 to 2015. At the end of the simulation we can access values of ‘my_var’ from 2014 by invoking a computation of ‘my_var_lag1’ or from 2013 using ‘my_var_lag2’. Similarly, we could use ‘income_x_cost_lag1’ to access values from the previous time period of the interaction expression above.

2.5. Indicator Framework

One way to analyze data in the Opus environment is via the indicator framework. It is designed to help analyze simulation results from running a model system as described in Section 2.3 and diagnose possible data and modeling anomalies. It operates on cache directories, i.e., data stored in the binary storage format.

The basic idea is to create a batch of indicator definitions (using variable names and the expression language), along with a pointer to the cache directory, the time point of interest, and the output format, one per each indicator. Available output formats are a tab or comma delimited ASCII file, map, chart, DBase table, Geotiff or Arcgeotiff maps. The whole batch is processed and the results can be viewed through an HTML interface.

In addition to computing simple indicators, the framework also supports change indicators. These are quantities computed as a difference in indicator values between two time periods or between outcomes of different simulations.

3. A Detailed Example of Using the Architecture

In this section we present an example to illustrate some of the functionality of the Opus platform described in the preceding sections. The discussion will be accompanied by examples of using Opus code. Basic knowledge of Python syntax is thus advantageous for further reading, but not required.

We assume that Opus has already been installed on the user’s computer. The system is cross-platform, and is in regular use with Windows, Linux, and

parcels.tab				households.tab		
parcel_id	land_value	x	y	household_id	income	parcel_id
1	500	1	1	1	15	1
2	350	1	2	2	12	2
3	400	1	3	3	13	3
4	430	1	4	4	35	4
5	510	1	5	5	10	4
6	550	2	1	6	25	5
7	480	2	2	7	30	6
8	480	2	3	8	22	7
9	500	2	4	9	18	7
10	530	2	5	10	41	8
11	570	3	1	11	45	9
12	500	3	2	11	52	10
13	560	3	3	13	48	11
14	600	3	4	14	15	11
15	620	3	5	15	50	12
16	580	4	1	16	85	13
17	600	4	2	17	20	13
18	690	4	3	18	55	14
19	750	4	4	19	82	15
20	800	4	5	20	51	16
				21	49	17
				22	90	18
				23	85	19
				24	99	20

Figure 4: Example of a parcel dataset and a household dataset stored as tab delimited ASCII files.

Macintosh operating systems. An installer is available for Windows; the other platforms require that the user first installs Python (if not already there) and a variety of packages, as well as the Opus code itself. Documentation and links to the source are available at www.urbansim.org.

3.1. Interacting With Data

Suppose we have a set of 20 parcels, for each of which we have observed its land value. We have also data on 24 households living on those parcels, namely their income and location. For simplicity, we assume the parcel centroids can be ordered into a square grid, whose coordinates are assigned to attributes `x` and `y`, respectively. In addition, each member of the two datasets has an unique identifier, here called `parcel_id` for parcels and `household_id` for households. The data are stored in tab delimited ASCII files, called ‘parcels.tab’ and ‘households.tab’, respectively, and are shown in Figure 4.

A dataset object uses a storage object pointing to the physical storage, in

this example of type ‘tab_storage’. We can create Opus datasets for our set of parcels and set of households as follows:

```
from opus_core.storage_factory import StorageFactory
from opus_core.datasets.dataset import Dataset

storage = StorageFactory().get_storage(
    type = "tab_storage",
    storage_location = "/directory/to/the/data"
)

parcels = Dataset(in_storage = storage,
                  in_table_name="parcels",
                  id_name="parcel_id",
                  dataset_name = "parcel"
)

households = Dataset(in_storage = storage,
                     in_table_name="households",
                     id_name="household_id",
                     dataset_name = "household"
)
```

Note that each dataset has a name that is used in variable names and expressions as discussed in Section 2.4. In this case, we use ‘parcel’ and ‘household’, respectively. Also note that in the ‘examples’ directory of the *opus_core* package, users can find examples for creating datasets using various available storage types. In addition to ‘tab_storage’, other available storage types are ‘csv_storage’, ‘flt_storage’, ‘dbf_storage’, ‘sql_storage’, ‘esri_storage’, and ‘dict_storage’.

The Dataset class has a substantial number of methods that allow for data manipulation, variable computation, analysis, and visualization. For example, `parcels.summary()` will give a statistical profile of the parcel’s attributes; `parcels["land_value"]` returns a numpy array of the `land_value` column; and `parcels.r_histogram("land_value")` creates a histogram of those values, including a density line.

A variable is usually defined as a transformation of an existing attribute, or a combination of several attributes of one or more datasets. For this purpose, Opus has a concept of a *dataset pool*, a collection of datasets that can be passed as one object to the computation procedure. During the computation, dependent datasets are accessed by their names. Assume, for example, we

want to compute the average income for each parcel. This involves data from both parcels and households. Thus, we create a dataset pool containing those two datasets and pass it to the computation method:

```
from opus_core.datasets.dataset_pool import DatasetPool
dataset_pool = DatasetPool(
    datasets_dict={"parcel": parcels, "household": households})
parcels.compute_variables(
    ["average_income=parcel.aggregate(household.income,function=mean)"],
    dataset_pool=dataset_pool)
```

The new variable, called ‘average_income’, computes an array of incomes, where each element of the array is the mean of the incomes of the households living on the corresponding parcel. The first part of the name ‘household.income’ (‘household’) determines in which dataset of the pool to look for the attribute ‘income’.

The above computation adds the variable ‘average_income’ to the parcels dataset as an additional attribute, which can be then accessed using `parcels["average_income"]`. Now, one can for example look the relationship between land value and average income of parcels by plotting a scatter plot:

```
parcels.plot_scatter("land_value", "average_income")
```

or alternatively `parcels.r_scatter(...)`. One can also visualize attributes in 2-d, for example

```
parcels.r_image("average_income", white_background=False,
                coordinate_system=("x", "y"))
```

or

```
parcels.plot_map("average_income", coordinate_system=("x", "y"))
```

Results of the `r_scatter(...)` and `r_image(...)` calls are shown in Figure 5. It can be seen that households with low income are placed into the north-west corner of the grid, whereas high income households live in south-east. An image of a correlation between dataset attributes can be also created, e.g. by

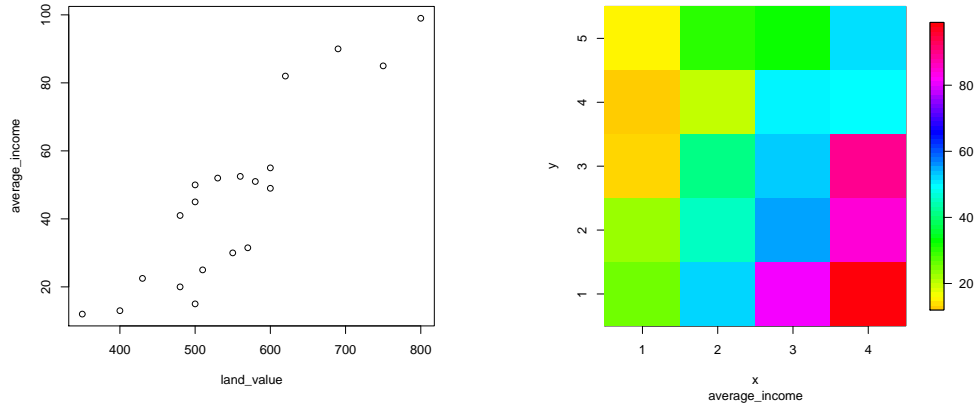


Figure 5: Simple examples of visualization methods: `r_scatter(...)` (left panel) and `r_image(...)` (right panel).

```
parcels.correlation_image(["land_value", "average_income", "x", "y"])
```

Using the Python module `rpy2`, one can apply any R functions to the data. For example, to fit a linear model to the scatter plot in Figure 5, one can do:

```
import rpy2.robjects as robjects
r = robjects.r
fit = r.lm("y ~ x", data=r.list(x=parcels["land_value"],
                               y=parcels["average_income"]))
print r.summary(fit)[3]
```

More information regarding the available methods for Opus Dataset is available in Appendix D of the *Opus/UrbanSim Users Guide and Reference Manual* (2011).

3.2. Specifying and Estimating Models in Opus

As mentioned in the previous sections, an Opus model is represented as a class that inherits from the class `Model`.

3.2.1. Hello World Model

The simplest model can have just a few lines of code:

```

from opus_core.models.model import Model
class MyModel(Model):
    def run(self):
        print "\n\nHello World, I'm an Opus model!"

```

If you store such code into a file called ‘mymodel.py’, you can run the model by typing

```

from mymodel import MyModel
MyModel().run()

```

When running, in addition to the outputs defined by the model, Opus can provide information about the run-time of the model. For conveniently formatted output, one can use the `logger` class, setting the model name to be used in the log outputs:

```

from opus_core.models.model import Model
from opus_core.logger import logger
class MyModel(Model):
    model_name = "My Opus Model"
    def run(self):
        logger.log_status("Hello World, I'm an Opus model!")

```

Using again `MyModel().run()` gives

```

Running My Opus Model (from __main__): started on May 31 10:07
    Hello World, I'm an Opus model!
Running My Opus Model (from __main__): completed.....0.0 sec

```

In addition to the `log_status` method, one can use `log_warning`, `log_error`, or `log_note`.

3.2.2. A Simple Model Example

In order not to have to implement commonly used models from scratch, Opus offers pre-defined model classes, as discussed in Section 2.2. The Simple Model, for example, modifies an attribute of a given dataset. An aging model mentioned in Table 1 could be implemented as follows, using the `households` dataset from above and after adding an attribute ‘age’ to it containing random values in specific range:

```

from numpy import random
households.add_primary_attribute(name="age",
                                data=random.randint(20,80,households.size()))
households["age"]

from opus_core.models.simple_model import SimpleModel
aging_model = SimpleModel()
aging_model.run(dataset=households,
                expression="new_age=household.age+1",
                outcome_attribute="age")
households["age"]

```

3.2.3. A Regression Model Example

There are several ways to specify predictors of an Opus model. A model specification is represented by the class `EquationSpecification`. We first give an example of a simple specification for fitting a linear model to the data shown in the scatter plot of Figure 5:

```

from opus_core.equation_specification import EquationSpecification
specification = EquationSpecification(
    variables = ("constant", "parcel.land_value"),
    coefficients = ("b_0", "b_1"))

```

Note that ‘constant’ is an Opus reserved word for specification of intercepts. Additional predictors and their coefficients would be simply added to the ‘variables’ tuple and to the ‘coefficients’ tuple, respectively. To estimate the model, we would proceed as follows:

```

from opus_core.models.regression_model import RegressionModel
rm = RegressionModel()
coefs, info = rm.estimate(specification, dataset=parcels,
                          outcome_attribute="parcel.average_income",
                          procedure="opus_core.estimate_linear_regression")

```

Note that the argument ‘procedure’ defines the estimation component of the model, as described in Section 2.2 and shown in Figure 3. The result of the ‘estimate’ method is a tuple, where the first element is an object of class `Coefficients` and the second element gives additional info about the estimation results in form of a Python dictionary. The estimated coefficients can be now passed to the ‘run’ method of the model:

```
result = rm.run(specification, coefs, dataset=parcels)
```

The ‘run’ method returns outcomes of the regression. One can replace the existing values of the outcome attribute by the new ones using

```
parcels["average_income"] = result
```

or equivalently using the Dataset method `modify_attribute(...)`. The visualization methods described above can then be used to explore model results.

As mentioned in Section 2.2, a model can have several sub-models. For this purpose, the constructor of the EquationSpecification class has an argument called ‘submodels’, which is again a tuple, as are the arguments ‘variables’ and ‘coefficients’. All of the arguments must have the same length if the ‘submodels’ argument is present, and hold the corresponding variables and coefficients for the different sub-models. The regression model must be then initialized with an argument ‘submodel_string’ which is an attribute of the dataset; its values distinguish the membership of dataset agents in the different sub-models.

For more complex models with many sub-models, this technique can become quite inconvenient. Therefore Opus also supports a dictionary-based specification. We give an example of a specification for 2 sub-models:

```
EquationSpecification(specification_dict = {
    "_definition_": [
        ("dataset.some_primary_attribute", "b_1"),
        "my_var=log(package.dataset.variable1*package.dataset.variable2)",
        ("package.dataset.variable1", "b_2"),
        "var2 = log(package.dataset.variable2*1000)",
        ("bias", "bias", 1)
    ],
    1: ["constant", "some_primary_attribute", "my_var", "var2"],
    5: ["constant", "variable1", "bias"]
})
```

An optional section ‘_definition_’ pre-defines all variables which are then further referenced by their aliases in the sub-model sections. Each definition is either a string defining the variable (in such a case the corresponding coefficient will have the same name), or a tuple of two or three elements. The

second element is the corresponding coefficient name; the third element, if present, defines a fixed value for the coefficient. In such a case, the coefficient is not estimated (see the ‘bias’ entry above). A specification of this form contains one section per sub-model, each of which is a list of references to variables in the ‘_definition_’ section. In the example above, we have two sub-models: 1 and 5. Sub-model 1 has three predictors and an intercept; sub-model 5 has an intercept, a predictor and an additional term that is not estimated. Alternatively, the definition entries can be put directly into the sub-model’s lists, in which case the ‘_definition_’ section can be omitted.

3.2.4. A Location Choice Model Example

This section shows an example of how to use the ChoiceModel class implemented in Opus. If we wish to predict choices of locations that households make, using the data in Figure 4, first we compose the choice model as follows:

```
from opus_core.models.choice_model import ChoiceModel
cm = ChoiceModel(choice_set = parcels,
                 utilities = "opus_core.upc.linear_utilities",
                 probabilities = "opus_core.upc.mnl_probabilities",
                 choices = "opus_core.upc.random_choices")
```

The choice set is here the set of parcels. Furthermore, we plug-in selected model components for computing utilities, probabilities and making choices.

To create a specification for a ChoiceModel one can proceed the same way as in the case of regression model. In addition, one can distinguish between equations – the EquationSpecification class supports an argument ‘equations’ for this purpose. As predictors in this example, we take household income interacted with parcel land value, as well as the parcel land value itself, and will not distinguish between equations:

```
specification = EquationSpecification(
    variables = ("household.income*parcel.land_value",
               "parcel.land_value"),
    coefficients = ("INtimesLV", "LV"))
```

Then we can estimate and run the model:

```

coefs, info = cm.estimate(specification, agent_set=households,
    procedure="opus_core.bhhh_mnl_estimation")
choices = cm.run(specification, coefs, agent_set=households)

```

The resulting choices are the parcel ids that each household chose.

An object of class `Coefficients` as well as of `EquationSpecification` can be stored in any Opus storage type. Thus, a model can be estimated once and the resulting coefficients subsequently re-used many times. For example, to store the computed coefficients in the tab storage created in Section 3.1, and then re-load it from there, we can use

```

coefs.write(out_storage=storage, out_table_name="my_coefficients")
from opus_core.coefficients import Coefficients
new_coefs = Coefficients()
new_coefs.load(in_storage=storage, in_table_name="my_coefficients")

```

Analogous code can be used with the `EquationSpecification` class. Both classes also support the method `summary()` for a quick check of the contents of those objects.

3.3. Combining Models into a Model System

Opus can use an XML configuration to configure individual models, as well as to combine them into a model system in which the models run sequentially in each time period. The model system operates on a cache directory where all data are stored in an efficient binary format.

3.3.1. Organizing Data

The architecture allows each model to use the cache location for collecting its own inputs. Moreover, model outputs can be again stored in the same location and used by other models as their inputs.

Each dataset is stored in a directory with the same name as the table name from which the data were imported, e.g. `parcels` or `households` in the example in Figure 4. Each attribute of a dataset is a numpy binary file located in this directory.

At the beginning of a simulation, the data collection in the base cache is copied into the given *run cache* directory as a sub-folder labeled with the

base year, i.e. the start time of the simulation. The model system creates a new sub-folder for each time increment, or year, of the simulation. For example, after two time periods of running a model system that involves only the household dataset from Figure 4, starting in year 2005, the run cache will contain the following directories:

```
run_xx_time/
  2005/
    households/
      household_id.li4
      income.li4
      parcel_id.li4
  2006/
    households/
      household_id.li4
      income.li4
      parcel_id.li4
  2007/
    households/
      household_id.li4
      income.li4
      parcel_id.li4
```

The name of the run directory is constructed automatically from the available runs on disk and the current date and time. Each of the sub-folders contains results of the simulation at the end of the corresponding time period.

To create a base cache, Opus provides several Python scripts for converting data from various formats into a cache type data. They are located in `opus_core/tools` and are named `do_export_type_to_cache.py`, where *type* is the storage type of the data, such as `tab`, `csv`, `sql`, `dbf` and `esri`. The user can call the script from the command line and set the options `-d` giving the location of the data to be converted, option `-t` giving the table name, option `-c` giving the cache location, and option `-y` giving the base year. To convert data from cache to *type* one can use the scripts `do_export_cache_to_type.py`. See the `--help` option for its arguments.

3.3.2. XML Configurations

Models and a model system can be configured using an XML file, which is then parsed by the `XMLConfiguration` class. The basic structure of such a

configuration is as follows (with the last section being optional):

```
<opus_project>
  <general> ... </general>
  <model_manager> ... </model_manager>
  <scenario_manager> ... </scenario_manager>
  <result_manager> ... </result_manager>
</opus_project>
```

The `<general>` section specifies general information of an Opus project, such as project name, description, or parent configuration. The main part of this section is the `<expression_library>` section, which contains a collection of variable definitions and expression used in model specifications.

The `<model_manager>` section configures individual models: it defines how it is imported, what are the arguments and outputs of each method, and the specification of the model if its coefficients need to be estimated. For example, the following model manager section defines the Simple Model from Section 3.2.2:

```
<models name="Models">
  <model name="simple_model" type="model">
    <structure type="dictionary">
      <import name="import" type="dictionary">
        <class_module type="string">opus_core.models.simple_model</class_module>
        <class_name type="string">SimpleModel</class_name>
      </import>
      <init type="dictionary">
        <name name="name" type="string">SimpleModel</name>
      </init>
      <run type="dictionary">
        <argument name="dataset">household</argument>
        <argument name="expression" type="string">new_age=household.age+1</argument>
        <argument name="outcome_attribute" type="string">age</argument>
        <output name="output" type="string">result</output>
      </run>
    </structure>
    <specification type="dictionary"/>
  </model>
</models>
```

The `<import>` section defines the module and class name of the model. The `<init>` section defines the name (and optionally arguments) of the constructor method. The `<run>` section has an `<argument>` node for each argument of the method, and an `<output>` node to define any return values of the model. Here, `result` could be passed to other models as input, if needed. The `<specification>` node usually contains model predictors, if applicable. For models that are estimated, the configuration would also contain the node `<estimate>` defined analogously to the `<run>` node. Intuitively, the configuration contains one `<model>` section per each model included in the model system.

The `<scenario_manager>` section defines scenarios to be simulated, including models to run, time period for the simulation, and location of cache directory. For example, to run the above aging model for 5 years, the scenario manager section contains the following:

```
<scenario executable="True" name="simple_scenario" type="scenario">
  <base_year type="integer">2010</base_year>
  <years_to_run config_name="years" type="tuple">
    <firstyear type="integer">2011</firstyear>
    <lastyear type="integer">2015</lastyear>
  </years_to_run>
  <models_to_run config_name="models" type="selectable_list">
    <selectable name="simple_model" type="selectable">True</selectable>
  </models_to_run>
  < ... >
</scenario>
```

A sample of a full XML configuration can be found in `opus_core/examples/simple.xml`.

3.3.3. *Launching a Simulation*

Once an XML configuration is created, a simulation can be launched via a tool script `'start_run.py'` from a command line shell:

```
python opus_core/tools/start_run.py \
  -x opus_core/examples/simple.xml -s simple_scenario
```

Similarly, model estimation can be invoked using the `'start_estimation.py'`

script once the specification of the model is provided in the `<specification>` section of the XML configuration, for example:

```
python opus_core/tools/start_estimation.py \  
-x opus_core/examples/simple.xml -m auto_ownerhsip_choice_model
```

3.3.4. *Creating Indicators*

Opus supports a versatile set of indicator types, including data table (tab or comma delimited format), chart (matplotlib) and map (geotiff, mapnik, matplotlib). Table and chart indicators can be created on almost any Opus dataset, while map indicators are usually created on a geography dataset. Indicators are configured in the `<results_manager>` section of the XML configuration. Indicators on the same dataset can be grouped into an indicator batch. For example, two table indicators on a geography dataset called ‘large_area’ can be configured as follows:

```
<results_manager>  
  <indicator_batches name="Indicator Batches" setexpanded="True" type="group">  
    <indicator_batch name="untitled_indicator_batch" type="indicator_batch">  
      <batch_visualization type="batch_visualization" name="summary">  
        <indicators>["number_of_households", "population"]</indicators>  
        <output_type>tab</output_type>  
        <visualization_type>tab</visualization_type>  
        <dataset_name>large_area</dataset_name>  
      </batch_visualization>  
    </indicator_batches>  
    <...>  
  </results_manager>
```

The indicators can be then generated using the script ‘make_indicators.py’:

```
python urbansim/tools/make_indicators.py \  
-x ../project_configs/seattle_parcel_default.xml \  
-i untitled_indicator_batch -y "(2000,)" -r base_year_data
```

This process of configuring and generating indicators, as well as running model estimations and simulations, is simplified if using the Opus GUI, see Section 4.3.

3.4. *Calibrating and Assessing Uncertainty*

In large-scale microsimulation that involve stochastic, or random, components, it is clearly important to assess the uncertainty in model predictions. (It is important for other model types as well, but is often ignored by modelers.) A method for assessing uncertainty in urban simulations, called Bayesian Melding (Raftery et al., 1995; Poole and Raftery, 2000), has been implemented in Opus (Ševčíková et al., 2007, 2011). The idea is to draw model inputs (such as coefficients) from a prior distribution and run a Monte Carlo simulation with these inputs. The simulation results at a specific time point are compared to observed data which yield weights for the different runs, a model bias, and variances. These quantities are propagated forward and can be applied to simulation results at the prediction time point.

In order to draw coefficient values from normal distribution given by the estimates and standard errors of the coefficients, the Coefficients class has the method ‘sample_values_from_normal_distribution’. One can for example construct a model system where different draws from estimated coefficients are passed to models and run a set of simulations for these draws.

Having results from a set of simulation runs, each run corresponding to one cache directory, one can investigate the results using the class MultipleRuns. It is initialized by pointing to the set of cache directories. One can then use for example the methods ‘compute_values_from_multiple_runs’ or ‘get_datasets_from_multiple_runs’ to explore results across runs, the latter allowing the use of any Dataset methods.

A child class of MultipleRuns, called BayesianMelding, supports passing observed data (in any storage type) and assessing uncertainty via a likelihood function. The likelihood function is implemented as a model component that is passed to the ‘compute_weights’ method, and thus it is easily exchangeable. By default, the likelihood is computed using a mixture of normal distributions. The likelihood for each run is proportional to its weight, which in principle evaluates that run with respect to the observed data. Thus, the highest weight implies that the run has the *best* inputs. This approach allows us to compare runs based not only on different coefficient values, but also runs that differ in other inputs, as well as in the set of models and their structure.

Furthermore, BayesianMelding supports propagating the uncertainty assessed using observed data forward in time. This in turn allows the user

to determine the theoretical posterior distribution of the outputs. Sampling from this posterior distribution is also possible, enabling the approximation of the posterior distribution of aggregated quantities.

4. Extending Opus

Building on the *opus_core* package, Opus can be extended in a modular way by adding other packages that import functionality from *opus_core* and adopt a standardized API for defining new Opus packages. Interfaces to completely separate software systems, models, and tools can also be incorporated.

4.1. Creating an Opus Package

Users who wish to create new models, define new datasets or implement new variable definitions will need to create an Opus package. The package is considered as a container for any new user-written Python code. Note that an Opus package is not needed by users who simply wish to reuse existing models or construct model specifications using existing datasets, variables or by creating new expressions. In such a case, an XML configuration is sufficient to define the models and their specifications.

An Opus package is a Python package conforming to Opus conventions that extends and customizes existing Opus functionality. By default, the package is placed in an Opus workspace, alongside other Opus packages such as *opus_core* and *urbansim* (see Section 4.2). The following Python code creates a skeleton of an Opus package called ‘atlantis’ pre-populated with the required folder structure and files:

```
from opus_core.opus_package import create_package
create_package('c:/opusworkspace', 'atlantis')
```

4.2. UrbanSim

UrbanSim (Waddell, 2002) is implemented as an Opus package, called *urbansim*, bundling several land use models into one model system. These are models for predicting land prices, simulating real estate development, predicting increase and decrease in population of jobs and households, and simulating choices of moving jobs and households of new locations. The models are implemented as subclasses of the existing models in *opus_core*, as described in

Section 2.2. For example, the `LandPriceModel` is a child of `RegressionModel`, and the `EmploymentLocationChoiceModel` and `HouseholdLocationChoiceModel` are both subclasses of the `ChoiceModel` class, supporting sampling agents using the *opus_core* sampling components. The *urbansim* package extends the collection of model components that can be plugged into models. For example, as an alternative to `random_choices` simulating agents making random choices (Figure 3), *urbansim* implements a component called `lottery_choices`, which takes into account capacity of locations and allows agents to re-decide if their choice is not available. This component is by default plugged into both the employment and household location choice models. (In general, there are ready-to-use configurations of all models containing useful default settings.)

The package pre-defines many useful variables for different geographies and different types of agents. These can be used as predictors for UrbanSim models or as indicators.

UrbanSim can be further customized to run at different geography levels: *urbansim_gridcell*, *urbansim_parcel*, and *urbansim_zone* are Opus packages that extend and customize *urbansim* to run at the gridcell, parcel, and zone level respectively. Each package can override geographies defined in *urbansim* or type of agents being modeled, as well as the set of models being included in the model system.

4.3. Graphical User Interface (*opus_gui*)

A Graphical User Interface (GUI) for *opus_core* and *urbansim* is implemented as an Opus package called *opus_gui*. It uses the Qt4 library and its Python wrapper pyQt, and exposes most of the underlying functions in a user friendly interface. The GUI can be launched using the Python script `opus_gui/opus.py`.

The GUI is built around XML configurations and is organized to facilitate the work flow of data management, model management, scenario management, and results management. The main window of the GUI reflects this work flow, including four tabs in the left-hand panel of the main window labeled *Data*, *Models*, *Scenarios*, and *Results*, plus a *General* tab. Each of the four tabs provides a container for configuring and running a variety of tasks, organized into the main functional areas involved in developing and using a simulation model.

- The *Data* tab organizes the processes related to moving data between the Opus environment and, doing data processing both within Opus, and also remotely in a database or GIS environment. Opus can use Python to pass data and commands to a database system like Postgres or MS SQL Server, or to a GIS system like ArcGIS or PostGIS. Tasks can be organized in the Data Manager as scripts, and run as a batch, or alternatively, they may be run interactively.
- The *Models* tab organizes the work of developing, configuring, and estimating the parameters of models, and of combining models into a model system. Within this tab, new models can be configured from a templates; arguments to individual models can be edited; specification of models whose coefficients need to be estimated from data can be entered and estimated.
- The *Scenarios* tab organizes the tasks related to configuring a scenario of input assumptions, and to interact with a run management system to actually run simulations on scenarios. Tools to monitor and interact with a running simulation are provided here.
- The *Results* tab provides the tools to explore results using indicators and visualization methods. It uses Opus indicator framework to generate a variety of indicators within GUI for both diagnostic and evaluation purposes. It also provides functionality to visualize indicators as charts, maps, and tables, and to export results to other formats for use outside of Opus.

The GUI can load an existing project file or create a new one, for example from a project template. A project file is an XML file containing configuration information, as described in Section 3.3.2. More details regarding the GUI can be found in (Kriplean et al., 2010) and in the *Opus/UrbanSim Users Guide and Reference Manual* (2011).

5. Status and Further Development

The Opus platform is now relatively feature-complete for the initial objectives outlined at the beginning of this paper. It has been used to re-implement the UrbanSim model system (Waddell, 2002; Noth et al., 2003; Waddell et al., 2003), resulting in faster execution than the previous Java platform, and dramatically increased ease of model specification, estimation and simulation

resulting from the increased modularity of the platform, the integration of model estimation algorithms, and the incorporation of a high-level domain-specific programming language to make it much easier to create new variables without Python programming (Borning et al., 2008). Laboratory studies and interviews provided tangible evidence of these and other benefits to users from the development of a Graphical User Interface for Opus and UrbanSim (Kriplean et al., 2010), as well as supporting the utility of an agile modeling methodology. The UrbanSim system, coupled with the Opus platform, has been used by a rapidly growing list of researchers and planning agencies throughout the United States, Europe, Asia and Africa. Applications using Opus and UrbanSim have been published with empirical results for Detroit (Waddell et al., 2008), Paris (de Palma et al., 2007), Salt Lake City (Waddell et al., 2007), San Francisco (Waddell et al., 2010), Seattle (Ševčíková et al., 2011), and several others, in addition to numerous unpublished applications.

The Opus platform has begun to be appropriated into environmental modeling domains as well. In the domain of landscape ecology, Opus was used to develop and estimate a land cover change model for the Puget Sound region of Washington, and this in turn was used to assess impacts of rapidly urbanizing environments on avian populations (Hepinstall et al., 2008). A second environmental application developed using the Opus platform was a spatially-explicit model of water consumption, which also used predictions from UrbanSim as inputs to the water demand model (Polebitski et al., 2011).

Taken together, this considerable body of use, along with lab studies and interviews, provides substantial evidence of the utility of the Opus platform. However, more remains to be done to make it even more productive for the urban and environmental geospatial modeling communities, and for researchers and planners more generally. Some specific areas that offer targets of opportunity for further development include:

- Making the Graphical User Interface more dynamic using custom forms and wizards, while moving some of the clutter of declarative specifications embedded in XML out of the way (so that it is still available for those who wish to modify these settings, but does not confuse users who do not wish to do so).
- Developing a more robust mechanism for error trapping and handling of error messages for users. The heavy nesting of Python classes and calls from a Graphical User Interface can result in some very arcane Traceback

messages that are daunting to many users.

- Experimenting with alternative approaches to improve computational performance. Although performance is reasonably good for the current uses of the system, more interactive use (for example for interactively supporting deliberation and discussion of simulation alternatives) will require much faster execution. There are many opportunities here for the careful introduction of parallel processing methods.
- Creating a well documented and tested Applications Programming Interface to allow programmers to extend the system and interface to it through the API.
- Coupling Opus and UrbanSim to a new 3D geometric analysis and visualization platform currently under development.
- Generating a broader engagement of users, software developers and modelers through the UrbanSim wiki and through workshops and conferences.

As the movement towards Open Source strategies and collaborative development efforts gain momentum, we are optimistic that platforms such as Opus can contribute to making scientists, students, and practitioners more productive in collaboratively solving the urban and environmental problems that motivate their work, and ours.

6. Acknowledgements

Thanks to the many students and staff members who have worked on the project, both at the University of California Berkeley and at the University of Washington, and to the UrbanSim user community, for their invaluable help in implementing, testing, and extending the Opus framework. This research has been funded in part by the National Science Foundation under Grants IIS-0705898 and IIS-0964412, and by a number of planning agencies, in particular the Puget Sound Regional Council, the Metropolitan Transportation Commission, and the Maricopa Associations of Governments.

References

- Abello, A., Kelly, S., King, A., 2002. Demographic Projections with DYNAMOD-2. Technical Report 21. National Centre for Social and Economic Modelling.

- Alberti, M., Waddell, P., 2000. An integrated urban development and ecological simulation model. *Integrated Assessment* 1, 215–227.
- Anselin, L., Rey, S., 2007. Pysal: A python library of spatial analytical methods. *The Review of Regional Studies* 37, 5–27.
- Berndt, E.R., Hall, B.H., Hall, R.E., 1974. Estimation and inference in nonlinear structural models. *Ann. Econ. Social Measurement* 3, 653–665.
- Bierlaire, M., 2003. Biogeme: A free package for the estimation of discrete choice models, in: *Proceedings of 3rd Swiss Transportation Research Conference*, Ascona, Switzerland.
- Borning, A., Ševčíková, H., Waddell, P., 2008. A domain-specific language for urban simulation variables, in: *Proceedings of the 9th Annual International Conference on Digital Government Research*, Montréal, Canada.
- de la Barra, T., 1995. *Integrated Land Use and Transportation Modeling: Decision Chains and Hierarchies*. Cambridge University Press.
- de Palma, A., Picard, N., Waddell, P., 2007. Discrete choice models with capacity constraints: An empirical analysis of the housing market of the greater Paris region. *Journal of Urban Economics* 62, 204–230.
- Dowling, R., Ireson, R., Skabardonis, A., Gillen, D., Stopher, P., 2005. *Predicting Air Quality Effects of Traffic-Flow Improvements: Final Report and User’s Guide*. Technical Report 535. National Cooperative Highway Research Program, Transportation Research Board, National Research Council.
- Echenique, M., Flowerdew, A., Hunt, J., Mayo, T., Skidmore, I., Simmonds, D., 1990. The MEPLAN models of Bilbao, Leeds and Dortmund. *Transport Reviews* 10, 309–322.
- Gribble, S., 2000. Lifepaths: A longitudinal microsimulation model using a synthetic approach, in: Gupta, A., Kapur, V. (Eds.), *Microsimulation in Government Policy and Forecasting*. Elsevier Science, Netherlands, pp. 383–395.
- Hammel, E., Mason, C., 1990. *SOCSIM II: A sociodemographic microsimulation program*. Revision 1.0: Operating manual. University of California,

- Berkeley. Special Publication of the Graduate Group in Demography and Program in Population Research, Working Paper No. 29.
- Hepinstall, J., Alberti, M., Marzluff, J., 2008. Predicting land cover change and avian community responses in rapidly urbanizing environments. *Landscape Ecology* 23, 1257–1276.
- Ihaka, R., Gentleman, R., 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5, 299–314.
- Kriplean, T., Borning, A., Waddell, P., Klang, C., Fogarty, J., 2010. Supporting agile modeling through experimentation in an integrated urban simulation framework, in: *Proceedings of the 10th Annual International Conference on Digital Government Research*, Puebla, Mexico. pp. 112–121.
- Luna, F., Stefannson, B. (Eds.), 2000. *Economic Simulations in Swarm: Agent-Based Modelling and Object-Oriented Programming*. Kluwer Academic Publishers.
- Manski, C.F., Lerman, S.R., 1977. The estimation of choice probabilities from choice based samples. *Econometrica* 45, 1977–1988.
- McFadden, D., 1974. Conditional logit analysis of qualitative choice behavior, in: Zarembka, P. (Ed.), *Frontiers in Econometrics*. Academic Press, New York, pp. 105–142.
- McFadden, D., 1981. Econometric models of probabilistic choice, in: Manski, C., McFadden, D. (Eds.), *Structural Analysis of Discrete Data with Econometric Applications*. MIT Press, Cambridge, MA, pp. 198–272.
- Noth, M., Borning, A., Waddell, P., 2003. An extensible, modular architecture for simulating urban development, transportation, and environmental impacts. *Computers, Environment and Urban Systems* 27, 181–203.
- Polebitski, A.S., Palmer, R.N., Waddell, P., 2011. Evaluating water demands under climate change and transitions in the urban environment. *Journal of Water Resources Planning and Management* 137, 249–257.
- Poole, D., Raftery, A.E., 2000. Inference for deterministic simulation models: The Bayesian melding approach. *Journal of the American Statistical Association* 95, 1244–1255.

- Putman, S., 1983. *Integrated Urban Models: Policy Analysis of Transportation and Land Use*. Pion, London.
- Raftery, A., Madigan, D., Hoeting, J., 1997. Bayesian model averaging for linear regression models. *Journal of the American Statistical Association* 92, 179–191.
- Raftery, A.E., Givens, G.H., Zeh, J.E., 1995. Inference from a deterministic population dynamics model for bowhead whales (with discussion). *Journal of the American Statistical Association* 90, 402–416.
- Ševčíková, H., Raftery, A., Waddell, P., 2007. Assessing uncertainty in urban simulations using Bayesian melding. *Transportation Research B* 41, 652–669.
- Ševčíková, H., Raftery, A., Waddell, P., 2011. Uncertain benefits: Application of Bayesian melding to the Alaskan Way Viaduct in Seattle. *Transportation Research A* 45, 540–553.
- UrbanSim Project, 2011. *The Open Platform for Urban Simulation and UrbanSim Version 4.3 Users Guide and Reference Manual*. The UrbanSim Project, University of California Berkeley and University of Washington. Available from <http://www.urbansim.org>.
- Waddell, P., 2002. UrbanSim: Modeling urban development for land use, transportation, and environmental planning. *Journal of the American Planning Association* 68, 297–314.
- Waddell, P., 2011. Integrated land use and transportation planning and modeling: Addressing challenges in research and practice. *Transport Reviews* 31, 209–229.
- Waddell, P., Borning, A., Noth, M., Freier, N., Becke, M., Ulfarsson, G., 2003. Microsimulation of urban development and location choices: Design and implementation of UrbanSim. *Networks and Spatial Economics* 3, 43–67.
- Waddell, P., Ševčíková, H., Socha, D., Miller, E., Nagel, K., 2005. *Opus: An Open Platform for Urban Simulation*. Presented at the Computers in Urban Planning and Urban Management Conference, London. Available from www.urbansim.org/papers.

- Waddell, P., Ulfarsson, G., Franklin, J., Lobb, J., 2007. Incorporating land use in metropolitan transportation planning. *Transportation Research Part A: Policy and Practice* 41, 382–410.
- Waddell, P., Wang, L., Charlton, B., Olsen, A., 2010. Microsimulating parcel-level land use and activity-based travel: Development of a prototype application in San Francisco. *Journal of Transportation and Land Use* 3, 65–84.
- Waddell, P., Wang, L., Liu, X., 2008. UrbanSim: An evolving planning support system for evolving communities, in: Brail, R. (Ed.), *Planning Support Systems for Cities and Regions*. Lincoln Institute for Land Policy.