Acknowledgments

The development of OPUS and UrbanSim has been supported by grants from the National Science Foundation Grants CMS-9818378, EIA-0090832, EIA-0121326, IIS-0534094, IIS-0705898, IIS-0964412, and IIS-0964302 and by grants from the U.S. Federal Highway Administration, U.S. Environmental Protection Agency, European Research Council, Maricopa Association of Governments, Puget Sound Regional Council, Oahu Metropolitan Planning Organization, Lane Council of Governments, Southeast Michigan Council of Governments, and the contributions of many users.

The UrbanSim Development Team

UrbanSim has been developed over a number of years with the effort of many individuals working towards common aims. See the UrbanSim People page for current and previous contributors, at www.urbansim.org/people. The following persons represent the current core development team of the UrbanSim project:

Paul Waddell, City and Regional Planning, University of California Berkeley (Project Coordinator)
Alan Borning, Computer Science and Engineering, University of Washington
Fletcher Foti, City and Regional Planning, University of California Berkeley
Hana Ševčíková, Statistics, University of Washington
Liming Wang, Institute for Urban and Regional Development, University of California Berkeley
Fabian Wauthier, Electrical Engineering and Computer Science, University of California Berkeley

UrbanSim Home Page: www.urbansim.org
# Contents

## I Overview of Opus and UrbanSim

1. **The Open Platform for Urban Simulation**
   1.1 Opus Design Objectives ................................................. 8
   1.2 Key Features of Opus ................................................. 9

2. **UrbanSim**
   2.1 Design Objectives and Key Features ................................. 12
   2.2 Model System Design .................................................. 14
   2.3 Policy Scenarios ..................................................... 16
   2.4 Discrete Choice Models .............................................. 16

## II The Opus Graphical User Interface

3. **Introduction to the OPUS GUI**
   3.1 Main Features .......................................................... 21
   3.2 Introduction to XML-based Project Configurations .................. 22

4. **The Variable Library** .................................................. 25

5. **The Menu Bar**
   5.1 Tools ................................................................. 28
   5.2 Preferences .......................................................... 28
   5.3 Database Server Connections ........................................ 29

6. **The General Tab** ...................................................... 30

7. **The Data Manager**
   7.1 Opus Data Tab ....................................................... 31
   7.2 Tools Tab ........................................................... 34

8. **The Models Manager**
   8.1 Creating an Allocation Model ....................................... 37
   8.2 Creating a Regression Model ....................................... 38
   8.3 Creating a Choice Model ........................................... 41

9. **The Scenarios Manager**
   9.1 Running a Simulation ................................................. 48

10. **The Results Manager** ................................................ 53
### 18.7 Database Tables about Employment .......................... 116
### 18.8 Database Tables about Households ........................... 120
### 18.9 Database Tables about Transportation Analysis Zones .... 124
### 18.10 Other Database Tables ........................................ 125
### 18.11 UrbanSim Constants ............................................. 126

### 19 Data for Gridcell-based Applications .......................... 128
#### 19.1 Database Tables about Grid Cells ............................ 128
#### 19.2 Database Tables about Development Types .................. 130
#### 19.3 Database Tables about Development Events ................ 132
#### 19.4 Database Tables About Development Constraints .......... 134
#### 19.5 Database Tables About Target Vacancies ................... 134

### 20 Data for Parcel-based Applications .............................. 136
#### 20.1 Database Tables About Parcels ................................ 136
#### 20.2 Database Tables about Buildings ............................ 137
#### 20.3 Database Tables About Development Projects ............... 137
#### 20.4 Database Tables About Development Constraints .......... 140
#### 20.5 Database Tables About Target Vacancy Rates ............... 141
#### 20.6 Database Tables About Refinement of Simulation Results . 141

### 21 Data for Zone-based Applications .................................. 143
#### 21.1 Database Tables About Buildings ............................ 143

### V Command-line Interface to Opus and UrbanSim ................. 145

### 22 Running a Simulation or Estimation .............................. 146
#### 22.1 Running a Simulation ........................................... 146
#### 22.2 Running an Estimation ......................................... 148
#### 22.3 Configurations ................................................... 148
#### 22.4 Output .......................................................... 154

### 23 Interactive Exploration of Opus and UrbanSim .................. 156
#### 23.1 Working with Data Sets ........................................ 156
#### 23.2 Working with Models .......................................... 161
#### 23.3 Opus Variables .................................................. 171
#### 23.4 Creating a New Model .......................................... 184
#### 23.5 Troubleshooting Python ........................................ 186
#### 23.6 Interactive Estimation and Diagnostics ....................... 187

### VI Opus and UrbanSim API .............................................. 195

### 24 The opus_core Opus Package ...................................... 196
#### 24.1 Datasets .......................................................... 196
#### 24.2 Data Storage ..................................................... 200
#### 24.3 Opus Variables .................................................. 204
#### 24.4 Models ........................................................... 210
#### 24.5 Model Components ............................................. 218
#### 24.6 Specification and Coefficients ................................ 223
#### 24.7 Other Classes .................................................... 226

### 25 The urbanism Opus Package ....................................... 228
#### 25.1 Introduction ..................................................... 228
Part I

Overview of Opus and UrbanSim
The Open Platform for Urban Simulation

1.1 Opus Design Objectives

In 2005, the Center for Urban Simulation and Policy Analysis at the University of Washington began a project to re-engineer UrbanSim as a more general software platform to support integrated modeling, and has launched an international collaboration to use and further develop the Open Platform for Urban Simulation (Opus). The broad vision for the effort is to develop a robust, modular and extensible open source framework and process for developing and using model components and integrated model systems, and to facilitate increased collaboration among developers and users in the evolution of the platform and its applications.

Opus is motivated by lessons learned from the UrbanSim project, and by a desire to collaborate on a single platform so that projects can more easily leverage each other’s work and focus on experimenting with and applying models, instead of spending their resources creating and maintaining model infrastructures. A similar project that provides inspiration for the Opus project is the R project (www.r-project.org). R [Ihaka and Gentleman, 1996] is an Open Source software system that supports the statistical computing community. It provides a language, basic system shell, and many core and user-contributed packages to estimate, analyze and visualize an extremely broad range of statistical models. Much of the cause for the rapidly growing success of this system, and its extensive and actively-contributing user community, is due to the excellent design of its core architecture and language. It provides a very small core, a minimal interactive shell that can be bypassed completely if a user wants to run a batch script, and a set of well-tested and documented core packages. Equally importantly, it provides a very standard and easy way to share user-contributed components. Much of the Opus architecture is based upon the R architecture.

The design of the core Opus architecture draws heavily on the experience of the UrbanSim project in software engineering and management of complex open source software development projects, and in the usability of these systems by stakeholders ranging from software developers, to modelers, to end-users.

The high-level design goals for Opus are to create a system that is very:

- Productive - to transform what users can do
- Flexible - to support experimentation
- Fast and scalable - to support production runs
- Straightforward - to make the system easy to use and extend
- Sharable - to benefit from others’ work

The following are some of the design requirements that emerged from these design goals:

- Have a low cost for developing new models. It should be easy for modelers around the world to code, test, document, package, and distribute new models. And it should be easy for modelers to download, use, read, understand, and extend packages created by others.
- Make it easy to engage in experimentation and prototyping, and then to move efficiently to production mode and have models that run very quickly.
- Have an interactive command-line interface so that it is easy to explore the code, do quick experiments, inspect data, etc.
- Be flexible so that it is easy to experiment with different combinations of parts, algorithms, data, and visualizations.
- Make it easy to inspect intermediate data, in order to aid the often complex diagnosis of problems found in large-scale production runs.
- Be extensible so that users can modify the behaviour of existing models without modifying the parts being extended, or build new models from existing parts, or replace existing parts with others that provide the same services in a different way.
- Be easy to integrate with other systems that provide complementary facilities, such as estimation, data visualization, data storage, GIS, etc.
- Be scriptable so that it is straightforward to move from experimentation or development into the mode of running batches of simulations.
- Run on a variety of operating systems, with a variety of data stores (e.g. databases).
- Handle data sets that are significantly larger than available main memory.
- Make it easy to take advantage of parallel processing, since much of the advances in chip processing power will come in the form of having multiple ‘cores’ on a single chip.
- Provide an easy mechanism for sharing packages, so that people can leverage each other’s work.
- Provide a mechanism for communities of people to collaborate on the creation and use of model systems specific to their interests.

1.2 Key Features of Opus

1.2.1 Graphical User Interface

From the 4.2 release of OPUS, a flexible, cross-platform user interface has been added that organizes the functionality in OPUS into conveniently accessible tabs oriented towards the workflow of developing and using models. The data manager tab contains a data browser and tools for a variety of data manipulations and conversion of data to and from GIS and SQL data repositories to the OPUS data format, geoprocessing tools in external GIS systems such as ESRI and Postgis, and data synthesis tools. The model manager provides infrastructure to create new models, configure them, specify the variables to use as predictive variables, and estimate model parameters using Ordinary Least Squares or Maximum Likelihood methods, depending on the model. The Scenario Manager provides convenient organization of model scenarios and the capacity to launch simulations and monitor them. The Results Manager manages the voluminous output of simulations and provides means to compute and visualize indicators derived from simulation results, and to display these results in maps, charts and tables. The GUI is data-driven from XML files, and can be readily extended by users who wish to add tools, variables, indicators, and other functionality.

1.2.2 Python as the Base Language

One of the most important parts in the system is the choice of programming language on which to build. This language must allow us to build a system with the above characteristics. After considering several different languages (C/C++, C#, Java, Perl, Python, R, Ruby) we choose Python for the language in which to implement Opus. Python provides a mature object-oriented language with good management of the use of memory, freeing up the memory when an object is no longer needed (automatic garbage collection). Python has a concise and clean syntax that results in programs that generally are 1/5 as long as comparable Java programs. In addition, Python has an extensive set of excellent open-source libraries. Many of these libraries are coded in C/C++ and are thus very efficient. There are also several mechanisms for ‘wrapping’ other existing packages and thus making them available to Python code.

Some of the Python libraries used by Opus as building blocks or foundational components are:

- Numpy: an open-source Python numerical library containing a wide variety of useful and fast array functions,
which are used throughout Opus to provide high performance computation for large data sets. The syntax for Numpy is quite similar to other matrix processing packages used in statistics, such as R, Gauss, Matlab, Scilab, and Octave, and it provides a very simple interface to this functionality from Python. See http://numpy.scipy.org/ for more details and documentation.

- **Scipy**: a scientific library that builds on Numpy, and adds many kinds of statistical and computational tools, such as non-linear optimization, which are used in estimating the parameters for models estimated with Maximum Likelihood methods. See http://scipy.org/ for details.
- **Matplotlib**: a 2-dimensional plotting package that also uses Numpy. It is used in Opus to provide charting and simple image mapping capabilities. See http://matplotlib.sourceforge.net/ for details.
- **SQLAlchemy**: provides a general interface from Python to a wide variety of Database Management Systems (DBMS), such as MySQL, Postgres, MS SQL Server, SQLite, and others. It allows Opus to move data between a database environment and Opus, which stores data internally in the Numpy format. See http://www.sqlalchemy.org/ for details.
- **PyQt4**: a Python interface to the Qt4 library for Graphical User Interface (GUI) development. This has been used to create the new Opus/UrbanSim GUI. See http://www.riverbankcomputing.co.uk/pyqt/ for details.

Python is an interpretive language, which makes it easy to do small experiments from Python’s interactive command line. For instance, we often write a simple test of a numarray function to confirm that our understanding of the documentation is correct. It is much easier to try things out in Python, than in Java or C++, for instance. At the same time, Python has excellent support for scripting and running batch jobs, since it is easy to do a lot with a few lines of Python, and Python ‘plays well’ with many other languages.

Python’s ability to work well for quick experiments, access high-performance libraries, and script other applications means that modelers need only learn one language for these tasks. Opus extends the abstractions available in Python with domain-specific abstractions useful for urban modelers, as described below.

### 1.2.3 Integrated Model Estimation and Application

Model application software in the land use and transportation domain has generally been written to apply a model, provided a set of inputs that include the initial data and the model coefficients. The process of generating model coefficients is generally handled by a separate process, generally using commercial econometric software. Unfortunately, there are many problems that this process does not assist users in addressing, or which the process may actually exacerbate. There are several potential sources of inconsistency that can cause significant problems in operational use, and in the experience of the authors this is one of the most common sources of problems in modelling applications.

First, if estimation and application software applications are separate, model specifications must be made redundantly - once in the estimation software and once in the application software. This raises the risk of application errors, some of which may not be perceived immediately by the user. Second, separate application and estimation software requires that an elaborate process be created to undertake the steps of creating an estimation data set that can be used by the estimation software, again giving rise to potential for errors. Third, there are many circumstances in which model estimation is done in an iterative fashion, due to experimentation with the model specification, updates to data, or other reasons. As a result of these concerns, a design objective for Opus is the close integration of model estimation and application, and the use of a single repository for model specifications. This is addressed in the Opus design by designating a single repository for model specification, by incorporating parameter estimation as an explicit step in implementing a model, and by providing well-integrated packages to estimate model parameters.

### 1.2.4 Database Management, GIS and Visualization

The extensive use of spatial data as the common element within and between models, and the need for spatial computations and visualization, make clear that the Opus platform requires access to these functions. Some of these are handled internally by efficient array processing and image processing capabilities of the Python Numeric library. But database management and GIS functionality will be accessed by coupling with existing Open Source database servers such as MySQL (www.mysql.org) and Postgres (www.postgresql.org), and GIS libraries such as QuantumGIS
Interfaces to some commercial DBMS are available through the SQLAlchemy library. An interface to the ESRI ArcGIS system has been implemented and is being refined.

1.2.5 Documentation, Examples and Tests

Documentation, examples and tests are three important ways to help users understand what a package can do, and how to use the package. Documentation for Opus and UrbanSim is created in both Adobe portable document format (pdf) and web-based format (html, xml), and is available locally with an installation, or can be accessed at any time from the UrbanSim web site. The pdf format makes it easy to print the document, and can produce more readable documents. Web-based documentation can be easier to navigate, and are particularly useful for automatically extracted code documentation.

1.2.6 Open Source License

The choice of a license is an crucial one for any software project, as it dictates the legal framework for the management of intellectual property embedded in the code. Opus has been released under the GNU General Public License (GPL). GPL is a standard license used for Open Source software. It allows users to obtain the source code as well as executables, to make modifications as desired, and to redistribute the original or modified code, provided that the distributed code also carries the same license as the original. It is a license that is intended to protect software from being converted to a proprietary license that would make the source code unavailable to users and developers.

The use of Open Source licensing is seen as a necessary precondition to the development of a collaborative software development effort such as envisioned for Opus. It ensures that the incentives for information sharing are positive and symmetrical for all participants, which is crucial to encourage contributions by users and collaborating developers. By contrast, a software project using a proprietary license has incentives not to release information that might compromise the secrecy of intellectual property that preserves competitive advantage. There are now many Open Source licenses available (see www.opensource.org), some of which allow derived work to be commercialized. Some software projects use a dual licensing scheme, releasing one version of the software under a GPL licence, and another (functionally identical) version of the software under a commercial licence, which allows also distributing software as a commercial application. Opus developers have opted to retain the GPL license approach as it is a pure Open Source license, and does not generate asymmetries in the incentives for information sharing. Any packages contributed to Opus by other groups must be licensed under a GPL-compatible license â¬Å§ we encourage them to be licensed under GPL itself, or less desirably, under LGPL (the library version of GPL).

1.2.7 Test, Build and Release Processes

Any software project involving more than one developer requires some infrastructure to coordinate development activities, and infrastructure is needed to test software in order to reduce the likelihood of software bugs, and a release process is needed to manage the packaging of the system for access by users. For each module written in Opus, unit tests are written that validate the functioning of the module. A testing program has also been implemented that runs all the tests in all the modules within Opus as a single batch process. For the initial release process, a testing program is being used to involve a small number of developers and users in testing the code and documentation.

The release process involves three types of releases of the software: major, minor, and maintenance. The version numbers reflect the release status as follows: a release numbered 4.2.8 would reflect major release 4, minor release 2 and maintenance release 8. Maintenance releases are for fixing bugs only. Minor releases are for modest additions of features, and major releases are obviously for more major changes in the system.

The Opus project currently uses the Subversion version control system for maintaining a shared repository for the code as it is developed by multiple developers. Write access to the repository is maintained by a core group of developers who control the quality of the code in the system, and this group can evolve over time as others begin actively participating in the further development of the system. A repository will also be set up for users who wish to contribute packages for use in Opus, with write access.
2.1 Design Objectives and Key Features

UrbanSim is an urban simulation system developed over the past several years to better inform deliberation on public choices with long-term, significant effects\(^1\). The principal motivation was that the urban environment is complex enough that it is not feasible to anticipate the effects of alternative courses of action without some form of analysis that could reflect the cause and effect interactions that could have both intended and possibly unintended consequences.

Consider a highway expansion project, for example. Traditional civil engineering training from the mid 20th century suggested that the problem was a relatively simple one: excess demand meant that vehicles were forced to slow down, leading to congestion bottlenecks. The remedy was seen as easing the bottleneck by adding capacity, thus restoring the balance of capacity to demand. Unfortunately, as Downs (2004) has articulately explained, and most of us have directly observed, once capacity is added, it rather quickly gets used up, leading some to conclude that ‘you can’t build your way out of congestion’. The reason things are not as simple as the older engineering perspective would have predicted is that individuals and organizations adapt to changing circumstances. Once the new capacity is available, initially vehicle speeds do increase, but this drop in the time cost of travel on the highway allows drivers taking other routes to change to this now-faster route, or to change their commute to work from a less-convenient shoulder of the peak time to a mid-peak time, or switching from transit or car-pooling to driving alone, adding demand at the most desired time of the day for travel. Over the longer-term, developers take advantage of the added capacity to build new housing and commercial and office space, households and firms take advantage of the accessibility to move farther out where they can acquire more land and sites are less expensive. In short, the urban transportation system is in a state of dynamic equilibrium, and when you perturb the equilibrium, the system, or more accurately, all the agents in the system, react in ways that tend to restore equilibrium. If there are faster speeds to be found to travel to desired destinations, people will find them.

The highway expansion example illustrates a broader theme: urban systems that include the transportation system, the housing market, the labor market (commuting), and other real estate markets for land, commercial, industrial, warehouse, and office space - are closely interconnected - much like the global financial system. An action taken in one sector ripples through the entire system to varying degrees, depending on how large an intervention it is, and what other interventions are occurring at the same time. This brings us to a second broad theme: interventions are rarely coordinated with each other, and often are conflicting or have a compounding effect that was not intended. This pattern is especially true in metropolitan areas consisting of many local cities and possibly multiple counties - each of which retain control of land use policies over a fraction of the metropolitan area, and none of which have a strong incentive, nor generally the means, to coordinate their actions. It is more often the case that local jurisdictions are taking actions in strategic ways that will enhance their competitive position for attracting tax base-enhancing development and residents. It is also systematically the case that transportation investments are evaluated independently of land use plans and the reactions of the real estate market.

UrbanSim was designed to attempt to reflect the interdependencies in dynamic urban systems, focusing on the real estate market and the transportation system, initially, and on the effects of individual interventions, and combinations

\(^{1}\)This chapter draws in part on reference [Waddell, 2001a]
of them, on patterns of development, travel demand, and household and firm location. Some goals that have shaped the
design of UrbanSim, and some that have emerged through the past several years of seeing it tested in the real world,
are the following:

Outcome Goals:

• Enable a wide variety of stakeholders (planners, public agencies, citizens and advocacy groups) to explore the
potential consequences of alternative public policies and investments using credible, unbiased analysis.
• Facilitate more effective democratic deliberation on contentious public actions regarding land use, transportation
and the environment, informed by the potential consequences of alternative courses of action that include long-term
cumulative effects on the environment, and distributional equity considerations.
• Make it easier for communities to achieve a common vision for the future of the community and its broader
environment, and to coordinate their actions to produce outcomes that are consistent with this vision.

Implementation Goals for UrbanSim:

• Create an analytical capacity to model the cause and effect interactions within local urban systems that are suffi-
ciently accurate and sensitive to policy interventions to be a credible source for informing deliberations.
• Make the model system credible by avoiding bias in the models though simplifying assumptions that obscure or
omit important cause-effect linkages at a level of detail needed to address stakeholder concerns.
• Make the model design behaviorally clear in terms of representing agents, actions, and cause-effect interactions in
ways that can be understood by non-technical stakeholders, while making the statistical methods used to implement
the model scientifically robust.
• Make the model system open, accessible and transparent, by adopting an Open Source licensing approach and
releasing the code and documentation on the web.
• Encourage the development of a collaborative approach to development and extension of the system, both through
open source licensing and web access, and by design choices and supporting organizational activities.
• Test the system extensively and repeatedly, and continually improve it by incorporating lessons learned from
applications, and from new advances in methods for modeling, statistical analysis, and software development.

The original design of UrbanSim adopted several elements to address these implementation goals, and these have
remained foundational in the development of the system over time. These design elements include:

• The representation of individual agents: initially households and firms, and later, persons and jobs.
• The representation of the supply and characteristics of land and of real estate development, at a fine spatial scale:
initially a mixture of parcels and zones, later gridcells of user-specified resolution.
• The adoption of a dynamic perspective of time, with the simulation proceeding in annual steps, and the urban
system evolving in a path dependent manner.
• The use of real estate markets as a central organizing focus, with consumer choices and supplier choices explicitly
represented, as well as the resulting effects on real estate prices. The relationship of agents to real estate tied to
specific locations provided a clean accounting of space and its use.
• The use of standard discrete choice models to represent the choices made by households and firms and developers
(principally location choices). This has relied principally on the traditional Multinomial Logit (MNL) specification,
to date.
• Integration of the urban simulation system with existing transportation model systems, to obtain information used
to compute accessibilities and their influence on location choices, and to provide the raw inputs to the travel
models.
• The adoption of an Open Source licensing for the software, written originally in Java, and recently reimplemented
using the Python language. The system has been updated and released continually on the web since 1998 at
www.urbansim.org.

The basic features of the UrbanSim model and software implementation are highlighted in Table 2.1. The model
is unique in that it departs from prior operational land use models based on cross-sectional, equilibrium, aggregate
approaches to adopt an approach that models individual households, jobs, buildings and parcels (or gridcells), and their changes from one year to the next as a consequence of economic changes, policy interventions, and market interactions.

Table 2.1: Key Features of UrbanSim

| Key Features of the UrbanSim Model System | • The model simulates the key decision makers and choices impacting urban development; in particular, the mobility and location choices of households and businesses, and the development choices of developers  
• The model explicitly accounts for land, structures (houses and commercial buildings), and occupants (households and businesses)  
• The model simulates urban development as a dynamic process over time and space, as opposed to a cross-sectional or equilibrium approach  
• The model simulates the land market as the interaction of demand (locational preferences of businesses and households) and supply (existing vacant space, new construction, and redevelopment), with prices adjusting to clear market  
• The model incorporates governmental policy assumptions explicitly, and evaluates policy impacts by modeling market responses  
• The model is based on random utility theory and uses logit models for the implementation of key demand components  
• The model is designed for high levels of spatial and activity disaggregation, with a zonal system identical to travel model zones  
• The model presently addresses both new development and redevelopment, using parcel-level detail |

| Key Features of the UrbanSim Software Implementation | • The model and user interface is currently compatible with Windows, Linux, Apple OS X, and other platforms supporting Python  
• The software is implemented in the Open Platform for Urban Simulation (Opus)  
• The software is Open Source, using the GPL license  
• The system is downloadable from the web at www.urbansim.org  
• The user interface focuses on configuring the model system, managing data, running and evaluating scenarios  
• The model is implemented using object-oriented programming to maximize software flexibility  
• The model inputs and results can be displayed using ArcGIS or other GIS software such as PostGIS  
• Model results are written to binary files, but can be exported to database management systems, text files, or geodatabases |

2.2 Model System Design

The components of UrbanSim are models acting on the objects in Figure 2.1, simulating the real-world actions of agents acting in the urban system. Developers construct new buildings or redevelop existing ones. Buildings are located on land parcels that have particular characteristics such as value, land use, slope, and other environmental characteristics. Governments set policies that regulate the use of land, through the imposition of land use plans, urban growth boundaries, environmental regulations, or through pricing policies such as development impact fees. Governments also build infrastructure, including transportation infrastructure, which interacts with the distribution of activities to generate patterns of accessibility at different locations that in turn influence the attractiveness of these sites for different consumers. Households have particular characteristics that may influence their preferences and demands for housing of different types at different locations. Businesses also have preferences that vary by industry and size of
business (number of employees) for alternative building types and locations.

These urban actors and processes are implemented in model components that are connected through the software implementation shown in Figure 2.1. The diagram reflects the interaction between the land use and travel model systems, and between the land use model and the GIS used for data preparation and visualization.

UrbanSim predicts the evolution of these objects and their characteristics over time, using annual steps to predict the movement and location choices of businesses and households, the development activities of developers, and the impacts of governmental policies and infrastructure choices. The land use model is interfaced with a metropolitan travel model system to deal with the interactions of land use and transportation. Access to opportunities, such as employment or shopping, are measured by the travel time or cost of accessing these opportunities via all available modes of travel.

The data inputs and outputs for operating the UrbanSim model are shown in Table 2.2. Developing the input database is challenging, owing to its detailed data requirements. A GIS is required to manage and combine these data into a form usable by the model, and can also be used to visualize the model results. Once the database is compiled, the model equations must be calibrated and entered into the model. A final step before actual use of the model is a validation process that tests the operation of the model over time and makes adjustments to the dynamic components of the model. The steps of data preparation, model estimation, calibration and validation will be addressed in later chapters. In the balance of this chapter the design and specification of UrbanSim, using the most recent parcel-based
approach used in the Puget Sound, is presented in more detail.

Table 2.2: Data Inputs and Outputs of UrbanSim

<table>
<thead>
<tr>
<th>UrbanSim Inputs</th>
<th>UrbanSim Outputs (by Building, Parcel or Gridcell), Generally Summarized by Zone</th>
<th>Travel Model Outputs (Zone-to-Zone) Used in UrbanSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Employment data, in the form of geocoded business establishments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Household data, merged from multiple census sources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Parcel database, with acreage, land use, housing units, nonresidential square footage, year built, land value, improvement value, city and county</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• City and County General Plans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• GIS Overlays for environmental features such as wetlands, floodways, steep slopes, or other sensitive or regulated lands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Traffic Analysis Zones</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• GIS Overlays for any other planning boundaries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Travel Model outputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Development Costs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Households by income, age, size, and presence of children</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Employment by industry and land use type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Acreage by land use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Dwelling units by type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Square feet of nonresidential space by type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Real estate prices</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Travel time by mode by time of day by purpose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Trips by mode by time of day by purpose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Composite utility of travel using all modes by purpose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Generalized costs (time + time equivalent of tolls) by purpose</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.3 Policy Scenarios

UrbanSim is designed to simulate and evaluate the potential effects of multiple scenarios. We use the term scenario in the context of UrbanSim in a very specific way: a scenario is a combination of input data and assumptions to the model system, including macroeconomic assumptions regarding the growth of population and employment in the study area, the configuration of the transportation system assumed to be in place in specific future years, and general plans of local jurisdictions that will regulate the types of development allowed at each location.

In order to facilitate comparative analysis, a model user such as a Metropolitan Planning Organization will generally adopt a specific scenario as a base of comparison or all other scenarios. This base scenario is generally referred to as the ‘baseline’ scenario, and this is usually based on the adopted or most likely to be adopted regional transportation plan, accompanied by the most likely assumptions regarding economic growth and land use policies. Once a scenario is created, it determines several inputs to UrbanSim:

- Control Totals: data on the aggregate amount of population and employment, by type, to be assumed for the region.
- Travel Data: data on zone to zone travel characteristics, from the travel model.
- Land Use Plan: data on general plans, assigned to individual parcels.
- Development Constraints: a set of rules that interpret the general plan codes, to indicate the allowed land use types and density ranges on each parcel.

2.4 Discrete Choice Models
UrbanSim makes extensive use of models of individual choice. A pathbreaking approach to modeling individual actions using discrete choice models emerged in the 1970’s, with the pioneering work of McFadden on Random Utility Maximization theory [McFadden, 1974, McFadden, 1981]. This approach derives a model of the probability of choosing among a set of available alternatives based on the characteristics of the chooser and the attributes of the alternative, and proportional to the relative utility that the alternatives generate for the chooser. Maximum likelihood and simulated maximum likelihood methods have been developed to estimate the parameters of these choice models from data on revealed or stated preferences, using a wide range of structural specifications (see [Train, 2003]). Early application of these models were principally in the transportation field, but also included work on residential location choices [Quigley, 1976, Lerman, 1977, McFadden, 1978], and on residential mobility [Clark and Lierop, 1986].

Let us begin with an example of a simple model of households choosing among alternative locations in the housing market, which we index by \( i \). For each agent, we assume that each alternative \( i \) has associated with it a utility \( U_i \) that can be separated into a systematic part and a random part:

\[
U_i = V_i + \epsilon_i, \tag{2.1}
\]

where \( V_i = \beta \cdot x_i \) is a linear-in-parameters function, \( \beta \) is a vector of \( k \) estimable coefficients, \( x_i \) is a vector of observed, exogenous, independent alternative-specific variables that may be interacted with the characteristics of the agent making the choice, and \( \epsilon_i \) is an unobserved random term. Assuming the unobserved term in (2.1) to be distributed with a Gumbel distribution leads to the widely used multinomial logit model [McFadden, 1974, McFadden, 1981]:

\[
P_i = \frac{e^{V_i}}{\sum_j e^{V_j}}, \tag{2.2}
\]

where \( j \) is an index over all possible alternatives. The estimable coefficients of (2.2), \( \beta \), are estimated with the method of maximum likelihood (see for example [Greene, 2002]).

The denominator of the equation for the choice model has a particular significance as an evaluation measure. The log of this denominator is called the logsum, or composite utility, and it summarizes the utility across all the alternatives. In the context of a choice of mode between origins and destinations, for example, it would summarize the utility (disutility) of travel, considering all the modes connecting the origins and destinations. It has theoretical appeal as an evaluation measure for this reason. In fact, the logsum from the mode choice model can be used as a measure of accessibility.

Choice models are implemented in UrbanSim in a modular way, to allow flexible specification of models to reflect a wide variety of choice situations. Figure 2.2 shows the process both in the form of the equations to be computed, and from the perspective of the tasks implemented as methods in software.

For each model component within the UrbanSim model system, the choice process proceeds as shown in Figure 2.2. The first steps of the model read the relevant model specifications and data. Then a choice set is constructed for each chooser. Currently this is done using random sampling of alternatives, which has been shown to generate consistent, though not efficient, estimates of model parameters [Ben-Akiva and Lerman, 1987].

The choice step in this algorithm warrants further explanation. Choice models predict choice probabilities, not choices. In order to predict choices given the predicted probabilities, we require an algorithm to select a specific choice outcome. A tempting approach would be to select the alternative with the maximum probability, but unfortunately this strategy would have the effect of selecting only the dominant outcome, and less frequent alternatives would be completely eliminated. In a mode choice model, for illustration, the transit mode would disappear, since the probability of choosing an auto mode is almost always higher than that of choosing transit. Clearly this is not a desirable or realistic outcome. In order to address this problem, the choice algorithm used for choice models uses a sampling approach. As illustrated in Figure 2.2, a choice outcome can be selected by sampling a random number from the uniform distribution in the range 0 to 1, and comparing this random draw to the cumulative probabilities of the alternatives. Whichever alternative the sampled random number falls within is the alternative that is selected as the ‘chosen’ one. This algorithm has the property that it preserves in the distribution of choice outcomes a close approximation of the original probability distribution, especially as the sample size of choosers becomes larger.

One other choice context is worth noting. In some situations, the availability of alternatives may be constrained. If the limit on availability is entirely predictable, such as a budget constraint eliminating expensive options, or a
Figure 2.2: Computation Process in UrbanSim Choice Models
zero-car household being unable to use the drive-alone mode, this is straightforward to handle, by eliminating the alternatives from the choice set for those choosers. In other situations, however, the solution is not so straightforward. In the case where alternatives may be unavailable because many other agents wish to choose them, the problem is that the alternative may be unavailable due to the endogenous congestion of alternatives. The effect of this is potentially significant, since it may cause parameters estimated in a choice model to confuse, or confound, the effects of preferences with the effects of constraints. Fortunately, an estimation method has been developed to account for this problem [de Palma et al., 2007].
Part II

The Opus Graphical User Interface
CHAPTER
THREE

Introduction to the OPUS GUI

This section of the documentation provides a tutorial approach to using the new The Graphical Interface (GUI) that has been added to the OPUS system as of version 4.2. This represents a substantial initiative to make UrbanSim and OPUS more user-friendly and accessible to modelers and model users — by reducing the need for programming expertise to use the system effectively to build, estimate, and use model systems for a range of applications. We think the GUI offers a substantial advance in usability, and look forward to user feedback, and contributions from collaborators, to continue building on this foundation.

Before starting the tutorial, please install Opus and UrbanSim on your machine if you haven’t already. Installation instructions are in Appendix A.

3.1 Main Features

The GUI is cross-platform compatible, and has been developed using the open source Qt4 library and the PyQt Python interface to it. Screenshots included in this section will be taken from all three platforms, to give a sense of the look and feel of the GUI on each platform. After launching the GUI from any of these platforms, the main Opus GUI window should be displayed as in Figure 3.1 or 3.2.

The organization of the GUI is based on an expectation that the work flow for developing and using models can be effectively organized into tasks that follow an ordering of data management, model management, scenario management, and results management. The main window of the GUI reflects this work flow expectation, by implementing 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.
- 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 once one or more scenarios have been simulated. It integrates an Indicator Framework that makes it possible to generate a variety of indicators, for diagnostic and for 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.
To launch the Opus GUI, you will need to run a python script called ‘opus.py’ in the ‘/opus/src/opus_gui’ directory.\(^1\) If you have used the Windows installer to install Opus, then a Windows Start menu item has been added under the Opus menu item under programs, so launching Opus is simple as selecting the OpusGUI Opus menu item. If you did not use the installer, for example, on OS X, or Linux, then open a command window or shell, change directory to the ‘opus_gui’ directory and type python opus.py. In Windows, you can also double-click on the ‘opus.py’ file in the ‘/opus/src/opus_gui’ directory to launch the GUI.

However it is launched, it will start from a command shell, and this window remains active while Opus is running. Do not attempt to quit this window until after exiting the Opus GUI, or Opus will close also.

Figure 3.1: Opus GUI Main Window - on Ubuntu Linux

3.2 Introduction to XML-based Project Configurations

Notice that at this point there are no contents in any of the tabs. Opus uses XML-based configuration files to flexibly specify the various aspects of a project. In addition, the appearance of the different tabs is also driven from the XML configuration files, so that different parts of the underlying functionality can be dynamically enabled and displayed, or hidden. XML stands for eXtensible Markup Language, and is a generalization of the HTML markup language used to display web pages. It is more flexible, and has become widely used to store content in a structured form.

Let’s add content to the GUI by loading a Project, which is in fact, just an XML file containing configuration information. From the main menu, load a project from ‘eugene_gridcell_default.xml’, which is in the default location of ‘opus/project_configs’. The project name and the file name are shown in the title bar of the window. The Opus

\(^1\)Note the use of forward slashes in the path. On the Macintosh OS X and Linux operating systems, and in Python, forward slashes are used to indicate separations in the path components. On Windows, backward slashes are used instead. Python can actually use forward slashes and translate them appropriately on Windows or other operating systems as needed, so we will use the convention of forward slashes throughout the text, for generality.
window should now appear as in Figure 3.3. In this case the project name is “eugene_gridcell” and the file name is ‘eugene_gridcell_default.xml’. The project name has an asterisk in front of it (as in Figure 3.3) if the project has unsaved changes. Your project may well be marked as having unsaved changes immediately on opening it, since the GUI automatically scans for simulation runs on startup and enters those in a section of the XML configuration. (The GUI will notify you via a popup window if it adds runs to the configuration.)

One important property of XML project configurations is that they can inherit from other project configurations. In practical terms for a user, this means that you can use default projects as templates, or parents, for another project you want to create that is mostly the same as an existing project, but has some changes from it. The new project is called a child project. By default, it inherits all the information contained in the parent project. However, it can override any information inherited from the parent, and add additional information.

Users should create their own projects in the ‘opus/project_configs’ directory. This will allow them to keep projects localized in one place, and to avoid editing and possibly corrupting one of the projects that are in an Opus package in the source code tree.

In fact, the ‘eugene_gridcell_default.xml’ configuration that we just opened initially contains almost no information of its own — virtually everything is inherited from a parent configuration. As you edit and augment this configuration, more information will be stored in the configuration. It can be saved to disk at any time using the “Save” or “Save as …” commands on the “File” menu. We suggest saving the configuration under a new name, say ‘my_eugene_gridcell.xml’, and working from that, so that if you want to start over at some point ‘eugene_gridcell_default.xml’ will still be there in its original state.

There is more information on XML-based project configurations later in this part in Chapter 11, but the above information should be enough to get started!
Figure 3.3: Opus GUI main window with the eugene_gridcell project open
Variable in Opus and UrbanSim represent quantities of interest. These variables can be used in two principal ways: in specifying models and in computing indicators to assess simulation results. For example, a distance_to_highway variable might be used to help predict land values or household location choices; and a population_density variable might be used in computing indicators that are useful for evaluating simulation results. The variable library is a repository for variables defined in the system that are accessible from the GUI. Since it provides a resource that is used throughout the GUI, we access it from the tools menu on the menu bar at the top of the main window, as in Figure 4.1. The screenshot in Figure 4.2 shows a popup window that appears once a user selects the variable library option on the tools menu. Note that the contents of it depend on what project is loaded. In this case, we have the eugene_parcel project loaded, and see the variables that are initially available in this project.

Variables are described in detail later in this manual. Briefly for now, there are three ways to define a variable:

- A variable can be defined using an expression written in a domain-specific programming language. There is a short description of the language later in this chapter, and a more complete description in Chapter 13. In this chapter we’ll only be looking at defining new variables in this way.

- A variable can also be a primary attribute of a dataset (think of these as columns in the input database).

- Finally, a variable can be defined as a Python class. This is an advanced option for complicated variables beyond the scope of the domain-specific programming language — we’ll use variables defined this way that are already in the library, but for now won’t write any new ones.

Note the buttons at the bottom of this window to add new variables or validate all variables. Adding a new variable defined as an expression is straightforward. There are examples of existing variables defined using expressions in the variable library window — look for variables with the entry “expression” in the “Source” column. The corresponding variable definition is in the right-most column. Note that this definition is executable code — it’s not just a description. Expressions are built up as functions and operations applied to existing variables in the library. For example, the expression for wetland is defined as gridcell.percent_wetland>50. This defines the creation of a true/false, or boolean, variable that is interpreted as 1 if the gridcell has more than 50 percent coverage by wetland, 0 otherwise. Variables are array-valued, so we are actually computing an array of true/false values for every gridcell with the single expression.

If you click on the add new variable button at the bottom of the variable library window, it opens a dialog box as shown in Figure 4.3. The top entry is the name you want to use for the variable. Let’s say we want to create a new variable that is a log of population density. We already have a population density variable defined by gridcell, so we can just take the log of this value. Let’s name the variable ln_population_density, leave the middle selection as “expression,” and fill in a simple expression in the definition area: ln(gridcell.population_density). We’re only going to use this variable in defining new models, not as an indicator, so we just check the “Will this variable be used in a model?” box, and leave “Will this variable be used as an indicator?” unchecked. Which boxes you check show up in the “Use” column of the variable library popup window, and also determine whether the new variable appears in lists of variables that you can add to a model specification or use to produce an indicator map; but don’t matter for the underlying definition.
Figure 4.1: Opening the Variable Library from the “Tools” Menu

Figure 4.2: Variable Library Popup Window
The dialog box provides two buttons at the bottom to help you check your new variable. The check syntax button tests whether the expression you have entered passes the Python and expression syntax checkers—in other words, is it syntactically correct. The second allows you to test whether if you apply this expression to your available data, it can successfully compute a result. This is very helpful in determining whether you might have referred to a data element that is not present, or is otherwise not computable with your data. In short, these two tools allow testing whether the variables are in a state that can be computed on the available data.

![Variable editor](image)

**Figure 4.3: Adding a New Variable**

We just saw an example of using an expression to define the `ln_population_density` variable. More generally, an expression for a new variable will be written in terms of existing variables, where these existing variables are referenced as a “qualified name” consisting of the dataset name and the variable name, for example, `gridcell.population_density`. Or our new variable can be used in yet another definition by writing its qualified name: `gridcell.ln_population_density`.

In building up an expression, we can apply various functions to other variables or subexpressions. For subexpressions that are arrays of floating point numbers, the available functions include `ln`, `exp`, and `sqrt`. We can also use the operators `+`, `−`, `∗`, `/`, and `**` (where `**` is exponentiation). The standard operator precedence rules apply: `**` has the highest precedence, then `∗` and `/`, then `+` and `−`.

You can use constants in expressions as well, which are coerced into arrays of the appropriate size, all filled with the constant value. For example, `ln(2∗gridcell.population_density)` is an array, consisting of the logs of 2 times the population density of each grid cell.

Two subexpressions that are arrays of integers or floats can be compared using one of the relational operators `<`, `≤`, `=`, `≥`, `>`, or `≠` (not equal). We saw an example of using a relational operation in the `gridcell.percent_wetland>50` expression. `gridcell.percent_wetland` denotes an array of integers, one for each gridcell. The constant 50 gets coerced to an array of the same length, and then the `>` operator compares the corresponding elements, and returns an array of booleans.
The Menu Bar

The main menu bar at the top of the OPUS GUI main window has three dropdown menus: File, Tools, and Help. The File menu includes the standard operations of opening a project, saving a project, saving a project under a new name, closing an OPUS Project, and finally exiting the OPUS GUI. Help offers an About option which produces a dialog box with information about OPUS and a set of links to the UrbanSim website, online documentation, and the GNU License. The Tools menu provides access to several general tools, described below.

Most of the items in the main menu bar are accessible from a secondary menu bar just above the tabs on the left side of the OPUS GUI window. Hovering over each icon will yield a tooltip with the item’s description.

5.1 Tools

The Tools menu, shown in figure 5.1, enables users to adjust settings and preferences, as well as opening different tabs in the right set of tabs. The items labeled “Log View” and “Result Browser” will each open new tabs on the right. The Result Browser is covered in greater detail in section 10.2.1. The items labeled “Variable Library,” “Preferences,” and “Database Connection Settings” each open a popup when clicked. (On the Macintosh, the “Preferences” item is under “Python” rather than “Tools.”) The Variable Library is further discussed in section 4.

![Tools Menu](image)

Figure 5.1: Tools Menu

5.2 Preferences

The Preferences dialog box changes some user interface related options in the OPUS UI. The dialog box is split into two sections, font preferences and previous project preferences. The font preferences section allows users to adjust font sizes specific to different areas of the GUI. The previous project preferences section contains a checkbox allowing users to open the most recently opened project each time OPUS GUI is started, this is turned off by default. Changes to the user preferences take effect as soon as either the “Apply” or “OK” buttons are clicked.
5.3 Database Server Connections

Database connections can be configured in the Database Server Connections dialog launched from the Tools menu. The Database Server Connections dialog, pictured in Figure 5.2, holds connection information for four database servers. Each connection is used for a specific purpose. While there are four different connections that must be configured, each may be configured to use the same host. Every connection requires a protocol, host name, user name, and password to be entered. Editing the protocol field produces a drop down of database protocols that UrbanSim is able to use. If a server has been setup for UrbanSim’s use choose the protocol that corresponds to the type of SQL server being used. If no SQL server is setup for a particular use, SQLite may be used. SQLite will create a local flat-file database instead of a remote server. UrbanSim currently supports MySQL, Microsoft SQL Server, Postgres, and SQLite.

The Database Connection Settings are saved when the Accept Changes button is pressed, ensuring that all future database connections will be made using the new settings. Database connections that are still in use while the database connection settings are being edited will not be changed until the connection is reestablished, for this reason it may be necessary to reopen any open project after changing the database connection settings.

![Database Connections](image)

Figure 5.2: Database Connections
The General Tab

The General tab is responsible for displaying general information about the current project. Figure 6.1 shows the General tab displaying information for the “eugene_gridcell” project. This information includes a brief description of the project, the project name, the parent project configuration, and a list of available datasets that can be used in computing the values of expressions. The “parent” field identifies the XML project from which the currently open project inherits. Inheritance is described in further detail in Chapter 11. Add a dataset to the “available_datasets” field if you have any extra datasets to be added to the project. The “dataset_pool_configuration” field includes a field named “package_order” that gives the search order when searching for variables implemented as Python classes; it is unlikely that you’ll want to change it.

The fields of the General tab can be edited by simply double clicking the value side of a field that is part of a project. If a field is displayed in blue, this field is being inherited from the parent project, and must first be added to the current project before editing can be done. To add a field to the current project, right-click and select “Add to current project”. Once added, the field may be edited as usual.

If you edit the “parent” field (not a typical operation), immediately save and close the project, and then re-open it, so that the inherited fields from the new parent will be shown.
The Data Manager

The Data Manager has two primary purposes, each reflected in the sub-tabs. One tab, the Opus Data tab, is for browsing, viewing, and exporting data from the Opus data cache. The other tab, the Tools tab, is a place for storing and executing various tools provided by the Opus community or tools you have written.

7.1 Opus Data Tab

The Opus Data tab is a file browser that defaults to the folder in which your project creates data. This folder name is composed from the default location for the Opus files, followed by ‘data’, followed by the project name. The project name for the ‘eugene_gridcell_default.xml’ project is “eugene_gridcell.” (This is given in the “project_name” element in the xml.) Thus, if you installed to ‘c:/opus’, and you are opening the Eugene sample project at ‘c:/opus/project_configs/eugene_gridcell_default.xml’, the data folder for this project is ‘c:/opus/data/eugene_gridcell’. That is the folder that this view starts at. Any subfolders and files are displayed in the tree view. See Figure 7.1.

![Figure 7.1: The Opus Data Tab](image)

There could be any number of subfolders here, but by default you will find a ‘base_year_data’ folder, and a ‘runs’ folder. The base_year_data folder will normally contain an Opus ‘database’ folder. An Opus database folder is any
folder containing Opus ‘datasets’. Often Opus database folders are titled with a year, such as 2000. Opus datasets are folders containing Opus ‘data arrays.’ Opus datasets are equivalent to the tables in a database. Opus data arrays are equivalent to the columns in a table, and are simply numpy arrays that have been written to disk in a binary format. See Figure 7.2.

![Figure 7.2: Opus databases, datasets, and arrays](image)

The Opus data arrays are referred to throughout the documentation as ‘primary attributes.’ Primary attributes are the actual data columns in a dataset. Computed attributes are attributes computed from primary attributes via expressions. For instance, if a parcels dataset contained the primary attributes population and area, a computed attribute called population_density could be computed by using the expression `population_density = population/area`. Once this expression is entered and stored in your project in the Variable Library, it can be used in a model and would be computed as needed. See Chapter 13 for more details on expressions.

### 7.1.1 Viewing and Browsing an Opus Data table

To view and browse the contents of an Opus dataset, right-click a data table, then select 'View Dataset'. This will bring up a new tab on the right-hand side of the Opus GUI window that will display some summary statistics about the dataset, and a table view of the raw data that can be browsed and sorted by clicking the column name. See Figure 7.3 for an example of browsing a data table.

### 7.1.2 Exporting an Opus Data table

An Opus dataset can be exported to another format for use in other applications. By default there are 3 options: ESRI, SQL, and CSV. To export a dataset, right-click a dataset, choose 'Export Opus dataset to,' then click your choice. See Figure 7.4 for the right-click menu. You will then see a pop-up window with the respective export tool with the parameters partially filled in based on which dataset you clicked. These are the same tools that you will find in the Tools tab of the Data Manager. For more information on the individual tools see the help tab on each tool.
Figure 7.3: Viewing and browsing an Opus dataset
7.2 Tools Tab

The Tools tab is an area to collect and execute tools and batches of tools provided with the interface, or it can be extended with tools that you write. A Tool is simply any script that is written in Python and executed by the interface.

7.2.1 Tool Library

The Tool Library is a place to collect and organize your tools. Tools can also be executed directly from the library in a ‘one-off’ manner, meaning that you can supply parameters to the tool and execute it without storing those parameters for future use. To execute a tool from the library, simply right-click it and choose ‘Execute Tool...’, see Figure 7.5. This will pop-up a window in which you can supply parameters to the tool then execute it.

Extending the Tool Library

New tools can be written and added to the tool library fairly easily. The best way to explain this is to use an example. A ‘template_tool’ has been provided so you can see how it works. Feel free to execute the template_tool, it just implements a simple loop and prints the results to the tool’s log window. The template_tool’s code and associated XML display everything that is needed to add a tool to the interface. See the code in the source code tree at /opus_gui/data_manager/run/tools/template_tool.py. A tool also needs XML configuration data in an Opus project. To view the XML configuration data for the template_tool, open urbansim.xml in an XML editor from the source code tree at /urbansim/configs and search for template_tool.

At the time of this writing new tools must be placed in the source tree at /opus_gui/data_manager/run/tools in order to run correctly. There are plans to create an additional ‘user tools’ folder where tools could also be placed. Also, at this moment, the XML must be hand written to ensure that the tools show up properly in the interface and execute correctly.
correctly. There are some right-click functions in the Tool Library to assist with the XML editing (to add a new tool, create parameters for it, etc.) but these functions are in a beta state.

Once a new tool and its associated XML is written properly, the Tool Library will display the tool and dynamically populate the pop-up dialog box with the proper parameters based on the XML configuration. The tools are quite flexible. Although the initial tool must be written using Python, there is no limit placed upon what one can do. For starters, there are tools provided in the interface that make OS calls to external executables (e.g. ogr2ogr.exe), databases, and myriad other libraries to accomplish various tasks (e.g. ESRI geoprocessing). Feel free to browse the source code for any provided tool along with the XML configuration to see some possibilities.

7.2.2 Tool Sets

Tool Sets are simply collections of tools from the Tool Library with parameters stored so they can be executed repeatedly or in order in a batch manner. Tool Sets can contain any number of tools from the Library. A new Tool Set can be created by right-clicking Tool_Sets and choosing 'Add new tool set.' This adds a new Tool Set to the bottom of the list. It can be renamed by double-clicking it and typing in a name, taking care to not use spaces or a leading integer as these are invalid in XML nodes. Once you have a new Tool Set, tools from the Library can be added to it by right-clicking a Tool Set and choosing 'Add Tool to Tool set.' See Figure 7.6 for an example of what this looks like.

From this window, choose a tool from the drop down menu, fill in the parameters, then click 'Add Tool.' The tool is added to the Tool Set and the parameters you entered are stored in the project XML file. This configured tool can now be executed by itself with those parameters, or executed as part of a batch in the Tool Set. Tools in a Tool Set can be re-ordered by right-clicking them and choosing to move them up or down, and all of the tools can be executed in the order they appear by right-clicking a Tool Set and choosing 'Execute Tool Set'.

Figure 7.5: Executing a tool
Figure 7.6: Adding a tool to a Tool Set
The Models Manager

The model manager tab in the GUI provides the functionality to create models of various types, configure them, specify the variables to use in them, and then estimate their parameters if they have a form that needs to be estimated – such as regression models or discrete choice models. A more thorough description of the types of models that can be implemented in OPUS is provided in Chapter 14.

8.1 Creating an Allocation Model

To demonstrate the process of creating models in the GUI, let’s begin with a simple allocation model, which does not have any parameters to estimate and represents a straightforward model to configure. Say we want to create a model that allocates home-based jobs to zones, and lack sufficient data to specify a choice model or regression model for this purpose. Home-based jobs are those jobs that are located in properties that are residential in character. Assume that we have no behavioral information about this problem, other than the insight that home-based jobs are... home-based. So we can infer that we should probably allocate these jobs to places that have housing (or households). In allocating these jobs to zones (traffic analysis zones used in the travel model), we can count how many residential units are in each zone, and use this as the weight to allocate the total home-based jobs to zones. That is, we want to proportionately
allocate home-based jobs to zones, weighted by the number of residential units in the zone. This is equivalent to saying that we want each residential unit to have an equal probability of receiving a home-based job (assuming that we do not have any information to suggest which residential units would be more likely than others to receive such a job).

The next consideration is the capacity of zones to absorb home-based jobs. One simplifying assumption we could make is that there is a maximum of one home-based job per residential unit in a zone. On average, our aggregate information suggests that most residential units do not have a home-based job, so this assumption should not be constraining.

We now have all the information we need to specify a model for home-based jobs. We will name the model allocate_home_based_jobs, to be descriptive. The table below contains the arguments we will need to use in creating this model in the GUI.

<table>
<thead>
<tr>
<th>Configuration Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Name</td>
<td>allocate_home_based_jobs_model</td>
</tr>
<tr>
<td>Dataset</td>
<td>zone</td>
</tr>
<tr>
<td>Outcome Attribute</td>
<td>home_based_jobs</td>
</tr>
<tr>
<td>Weight Attribute</td>
<td>zone.aggregate(building.residential_units)</td>
</tr>
<tr>
<td>Control Totals</td>
<td>annual_employment_control_totals</td>
</tr>
<tr>
<td>Year Attribute</td>
<td>year</td>
</tr>
<tr>
<td>Capacity Attribute</td>
<td>zone.aggregate(building.residential_units)</td>
</tr>
</tbody>
</table>

The create new model dialog box (Figure fig:create-model) contains several kinds of model templates we could create a model from. One of these is Allocation Model. The capacity to create new allocation models, such as this, is now available in the Opus GUI. Select Allocation Model from the list, and a new dialog box appears, with several fields to fill in. Fill in the fields with the contents from Table 8.1, and save it. Once this is done, it will appear in the list of models under the Models section of the Model Manager tab. It is now a fully enabled model, and can be included in a simulation run.

It should go without saying (but doesn’t), that creating models through the GUI, with a few mouse clicks and filling in a few fields in a dialog box, is much, much easier than it has been in the past. One does not need to be an expert software developer in order to create and use interesting and fully functional models in OPUS.

8.2 Creating a Regression Model

Regression models are also simple to create and specify in the Opus GUI, and can be estimated and simulated within the graphical interface. Assume we want to create a model that predicts population density, using the population per gridcell as the dependent variable and other attributes we can observe about gridcells as independent (predictor) variables. Note that this is not a very useful model in this context since we actually have a household location choice model to assign households to gridcells – so this model is for demonstration purposes only.

To create this model in the Opus GUI, right-click again on Models, and select in this case Regression Model to generate a new dialog box for this model template, as shown in Figure 8.3. We just need to provide three arguments in the dialog box - a name to assign to the new model (we will use population_density_model), a dataset to contain the dependent variable (gridcell), and the name of the dependent variable (population_density) - which should exist in the base year, or be an expression to compute it from other attributes already in the data.

Once the values have been assigned to the configuration of the new model, and you click OK on the dialog box, the model is added to the list of models under Models. If you expand this node by clicking on the plus sign to the left of the new land price model entry, you will see that it contains a specification and a structure node. Expand the specification node, and you will find some additional detail, including a reference to submodels, and a variables entry. We will ignore submodels for now – it is a means of specifying that you would like to specify the model differently for
8.2. Creating a Regression Model

Figure 8.2: Creating a New Allocation Model from a Template

Figure 8.3: Creating a New Regression Model from a Template
different subsets of the data. For now we will just apply a single specification to all the data, to keep this a bit simpler. We can now move to the task of specifying and estimating this model.

Right-click on the variables node, and click on Select Variables, as shown in Figure 8.4. At this point a window should appear as shown in Figure 8.5 that is essentially the same as the variables library window you encountered earlier. There is a column of check-boxes at the left hand side of the window which you can use to identify the variables you want to include as independent variables, or predictive variables, for this model. The button at the bottom allows you to accept the selection, which then updates the list of variables in the model specification. Try adding a constant term, since this is a regression and we need an intercept, or a base value. Also add a variable like population density. Now accept the selections.

We can now move to the task of specifying and estimating this model.

![Figure 8.4: Specify the New Population Density Model](image)

Once the model specification has been entered, we can estimate the model parameters using Ordinary Least Squares by right-clicking on the population density model and selecting Run Estimation, as shown in Figure 8.6. Once this has been clicked, a new tab appears on the right hand side of the main window, to interact with the model estimation. Click on the start estimation button, and within a few seconds you should see the estimation results appear in this tab, as shown in Figure 8.7.

We can see from the results that the constant and travel time to the CBD, and also land value were quite statistically significant, and that they explain around 28 percent of the variation in population density in Eugene. Clearly this is a toy model, but adding other variables in this way can increase the explanatory power to a quite useful level, and as you can see modifying the specification and estimating the model is not difficult to do.

One other note at this point is that the specification and estimation results are automatically stored, if you request this,
8.3 Creating a Choice Model

Replace this later, after further testing and cleanup The next type of model we will create is a choice model. This is a very common modeling application, and is used widely. Our example for this demonstration is a model of housing type choice, which for purposes of keeping the example simple, we reduce to two alternatives: single-family housing type, or other. It is likely that households choosing to live in single-family housing may make different trade-offs in other choices, such as travel, and car ownership. By reducing the model to two outcomes, we create a binary choice model specification. We always need to use one alternative as a base of comparison in choice models, so for this model we will use the other housing type as the base of comparison.

Below are the configuration settings for creating a choice model of housing type. In order to create this model for estimation purposes, we will exclude the few households that are in non-residential property types, and only keep those in multi-family, condominium, and single-family housing. These are reflected by building type id values of 4, 12, and 19, respectively, in the seattle parcel data. Filtering the data to include only these three values can be done with the numpy logical_or command, but since it takes only two arguments, we need to create a nested comparison, as shown below. Since there are three housing types represented in this data, and we want to create a binary choice outcome for simplicity, it is necessary to create a dependent variable that is 2 if the household occupies a single family house, and 1 otherwise. In this example, since we will use the entire household table, we draw a small sample of 5% of the agents to use in estimating the model.

Figure 8.9 shows the housing type choice model configuration in progress. For a binary choice model such as this, the specification of the model is quite similar to the specification of the regression model, but there are some subtle differences. The most important one is that, as this model is implemented in Opus now, it requires at least one variable in each equation - that is - per alternative. We typically assign the constant to alternative 1, and all other variables to alternative 2. In a future revision of the code, the base alternative will not take any variables (this is the more standard
Figure 8.6: Estimate the New Population Density Model
Figure 8.7: Estimation Results

8.3. Creating a Choice Model
Figure 8.8: Save Estimation Results Flag
Table 8.2: Creating a Housing Type Choice Model

<table>
<thead>
<tr>
<th>Configuration Entry</th>
<th>Node</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Name</td>
<td></td>
<td>housing_type_choice_model</td>
</tr>
<tr>
<td>Choice Set</td>
<td>Init</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>Choice Attribute</td>
<td>Init</td>
<td>single_family=(household.disaggregate(building.building_type_id)==19)+1</td>
</tr>
<tr>
<td>Estimation Size</td>
<td>Init.Estimation Config</td>
<td>0.05</td>
</tr>
<tr>
<td>Agent Set</td>
<td>Run, Prepare for Run, Estimate, Prepare for Estimate</td>
<td>household</td>
</tr>
<tr>
<td>Agent Filter</td>
<td>Prepare for Run</td>
<td>numpy.logical_or(numpy.logical_or(household.disaggregate(building.building_type_id)==4,household.disaggregate(building.building_type_id)==12),household.disaggregate(building.building_type_id)==19)</td>
</tr>
<tr>
<td>Specification Table</td>
<td>Prepare for Run</td>
<td>housing_type_choice_model Specifications</td>
</tr>
<tr>
<td>Coefficients Table</td>
<td>Prepare for Run, Prepare for Estimate</td>
<td>housing_type_choice_model_coefficients</td>
</tr>
</tbody>
</table>

Figure 8.10 shows the initial specification of the housing type choice model, with a constant for the other housing types, and income and has_children included in the utility specification for the single family housing alternative.

Once the model is specified, it needs to be added to the list of models to estimate, and selected as the model to estimate, as was the case in the preceding regression model example. Once the model has been added and the project saved, the model can be estimated with the normal right-click option on the models to estimate node. The results are shown in Figure 8.11.
Figure 8.9: Configuring the Housing Type Choice Model

Figure 8.10: Specifying the Housing Type Choice Model
Figure 8.11: Estimating the Housing Type Choice Model
The Scenarios Manager

9.1 Running a Simulation

Once a project has been developed, including the data to be used in it, and the model system has been configured and the parameters for the models estimated, the next step is to create and run a scenario. In the eugene_gridcell project, a baseline scenario has already been created and is ready to run. To run this scenario, in the Scenario Manager, right-click with the mouse on the Eugene_baseline entry and select “Run this Scenario.” At this point, a frame should appear in the right hand side of the Opus window, as shown in Figure 9.1.

![Figure 9.1: Starting a simulation on the Eugene baseline scenario](image)

The frame on the right contains an option to start the simulation run and another to cancel it. Start the run with the Start Simulation Run ... button. Note that the button’s name changes to Pause Simulation Run ... The window will now update as the simulation proceeds, with progress bars and labels being updated to show the changing state of the system, such as which year the model is simulating and which model is running. If the simulation completes successfully, the “Total progress:” bar will say “Simulation ran successfully” after it, and the “Year progress” bar will say “Finished” (Figure 9.2).
This simulation run then is entered into the simulation runs database, which can be subsequently inspected via the “Simulation_runs” node in the Results Manager (Section 10.1). Indicator results from the simulation can also be generated using other tools in the Results Manager (Section 10.2).

That’s it for the basics of running a simulation! However, there are various options for selecting, configuring and running a scenario, which are described in the following subsections.

9.1.1 Options for Controlling the Simulation Run

Here are the aspects of controlling the simulation run via controls in the frame on the right. The results of this simulation are entered into the simulation runs database under the “Run Name” (upper right hand part of the pane). This is filled in with a default value consisting of “run_” followed by the date and time; edit the field if you’d like to give the run a custom name. Immediately under that field is a combo box “Indicator Batch:”. If you have defined and named a batch of indicators that can be run, you can select one of these from the list, and the indicators will automatically be run at the conclusion of the simulation. See Section 10.2.2 for information on indicator batches.

As mentioned above, once you start the simulation the Start Simulation Run ... button label changes to Pause Simulation Run. If the pause button is pressed while the model system is running, a request to pause the model is triggered, and once the current model in the model system is finished, the system pauses until you take further action by pressing either the Resume Simulation Run... or the Cancel Simulation Run button. The Cancel Simulation Run button is also available while the simulation is running. (As with “Pause,” “Cancel” generally won’t happen immediately, but only after the current model in the model system is finished.)

9.1.2 Options for Monitoring the Simulation

Moving down the frame, there is a button for selecting views in the bottom of the screen. The default is “Simulation Progress,” which shows progress bars to allow you to track the simulation’s activity. The “Log” option shows a
transcript with detailed simulation output. Finally, the “Diagnostics” button supports generating on-the-fly diagnostic indicators.

Figure 9.3: Running a simulation on Eugene baseline with all progress bars enabled

The default settings for “Simulation Progress” are to show the total progress in running the simulation, and the progress within the current year. For example, if you are running for 10 simulated years, when the first year is completed the progress bar for the current year will reach 100% and the total progress will reach 10%. You can hide the progress bar for the current year for a less cluttered display. Finally, if “Show progress within current year” is enabled, you have the additional option (off by default) to show progress within each model, such as Employment Transition, Household Transition, and so forth. The currently running year, model, and chunk of model will be shown by the corresponding bars. Figure 9.3 shows a running simulation with all three progress bars enabled.

The “Log” option, as shown in Figure 9.4, shows the log output from running the same scenario. This can be monitored as the simulation runs, and also consulted after it has completed.

“Diagnostics,” the last of these three options, supports generating on-the-fly diagnostic indicators. If this option is selected, a series of combo boxes appears immediately below, allowing you to select a map or chart indicator, the level of geography, a specific indicator, and the year. As the given simulated year completes, the corresponding indicator visualization is shown underneath. For example, Figure 9.5 shows a diagnostic map of population for 1981 at the gridcell level for the Eugene baseline scenario.

9.1.3 Selecting and Configuring a Scenario

To select a scenario to run or configure its options, use the XML tree view on the left.

The default Eugene project contains only one scenario (“Eugene_baseline”). However, in general projects can contain multiple scenarios, all of which will be shown in the XML tree in the left-hand pane. Right-click on any one of them and select “Run this Scenario” to run.

To change any of the options in the XML tree you need to be able to edit it. The initial version of the Eugene_gridcell project in ‘opus_home/project_configs/eugene_gridcell_default.xml’ has basically no content of its own — everything
Figure 9.4: Log information from running a scenario

Figure 9.5: Using diagnostic indicators while running a scenario

9.1. Running a Simulation
is inherited from the default Eugene gridcell configuration in the source code. Inherited parts of the XML tree are shown in blue. These can’t be edited directly — they just show inherited information. Suppose that we want to change the last year of the simulation run from 1981 to 1990. To do this, first click on the triangle next to `years_to_run` in the XML tree. Right click on `lastyear` and select “Add to current project.” This copies the inherited information into the local XML tree for the Eugene_gridcell configuration. The `lastyear` entry turns black, and you can now edit the year value to 1990. After the options are set up as you would like, you can then right click on the scenario and run it. Figure 9.6 shows the newly-edited scenario being run for 10 simulated years. (Notice that `lastyear` is now 1990 and shown in black.)

![Figure 9.6: Running a scenario after changing the last year to 1990](image)

The ending year of the scenario is the most likely thing that you’d want to change. Another possibility is to run only some of the component models. To do this, right-click on “Models to run,” make it editable by selecting “Add to current project,” and change the value of the “Run” field to “Skip” to skip that component model.
The Results Manager

The Results Manager, corresponding to the Results tab of the GUI, has two main responsibilities: to manage simulation runs for this project (Section 10.1) and to allow the interrogation of these simulation runs through the use of indicators (Section 10.2). We explore both of these in this chapter.

10.1 Managing simulation runs

The “Simulation runs” node captures all available simulation runs for this project. For example, when a simulation is run from the “Scenarios-manager” (see chapter 9), an entry is made under “Simulation runs”. If for some reason a run that you know exists for this project is not listed, right-click on “Simulation runs” and select “Import run from disk” (See Figure 10.1). The GUI will try to load the missing run and make it available.

![Figure 10.1: Importing a run from disk that is not showing up as a “Simulation run”.

A couple operations can be performed on a simulation run. To view information about a given run, right-click on the run and select “Show details”. To remove all traces of a simulation run, including the data on disk, right-click on the run and select “Remove run and delete from harddrive”.

10.2 Interrogating Results with Indicators
Indicators are variables defined explicitly for use as a meaningful measure (see Chapters 4 and 13). Like model variables, they can be defined using the domain-specific programming language via the “Variable Library” accessible through the Tools menu (see Chapter 4). An indicator can then be visualized as either a map or as a table in a variety of formats. The GUI provides two ways to use indicators to understand what has happened in a simulation run: interactive result exploration (Section 10.2.1) and Batch indicator configuration and execution (Section 10.2.2).

10.2.1 Interactive result exploration

Often, it is desirable to explore simulation results in a lightweight fashion in order to get a basic idea of what happened. You don’t necessarily want to go through the process of exporting results to a GIS mapping tool in order to gain some basic intuitions into spatial patterns.

The Opus GUI’s “Result Browser”, available from the “tools” menu, allows interactive exploration of simulation results. The Result Browser presents a selectable list of available simulation runs, years over which those simulations were run, and available indicators. You can then configure an indicator visualization by selecting a simulation run, a year, and an indicator. To compute and visualize the configured indicator, simply press the “generate results” button (See Figure 10.2). The indicator will then be computed for the year of the selected simulation run. After it is computed, a tab should appear at the bottom of the window with the name of the indicator. Subtabs allow you to see the results as a table or map (using the Matplotlib Python module).

Figure 10.2: Using the “Result browser” for interactive result exploration.
1. Open the Results Browser from the Tools menu. Use the Results Browser to answer the following questions.

2. Just from visual inspection, is there more than one cluster of gridcells with high land value in the Eugene region in 1980 in the baseyear data?

3. Is this cluster(s) in the same general area as the greatest number of jobs in Eugene for the same year of the baseyear data?

Two additional aspects of the Result Browser should be mentioned:

1. If the checkbox “Automatically view indicator” is clicked, everytime you change the indicator configuration (i.e. select a different simulation run, year, or indicator), the indicator will be automatically visualized (as if you pressed the “Generate results” button).

2. The “Export results” button will export the table data of the currently configured indicator to a database. This feature is not yet implemented.

10.2.2 Batch indicator configuration and execution

The “Result Browser” is good for poking around in the data. But often you’ll want to generate the same set of indicators for each of many runs and you don’t want to redefine them every time. Instead, you’d like to configure and save a group of them that can be executed on demand on an arbitrary simulation run. In the Opus GUI, this functionality is supported with indicator batches.

To create a new indicator batch, right-click on the “Indicator_batches” node in the “Results tab” and select “Add new indicator batch...” (See Figure 10.3). A new batch will be created under the Indicator_batches node.

A batch is a collection of “Indicator visualization” definitions. Each indicator visualization is a configuration of the indicator variable to be used, a visualization style (e.g. map or table), and some format options. To add a new indicator visualization to the batch, right-click on the respective batch and select “Add new indicator visualization...”. A dialog box will appear where you can define the visualization. The visualization options for an indicator visualization are discussed in depth later.

You can add as many indicator visualizations to a batch as you want. In order to execute an indicator batch on a simulation run, right-click on the indicator batch and hover over “Run indicator batch on...”. A list of all the available simulations runs will appear as a submenu. You can then select the appropriate simulation. The indicator visualizations in the batch will be executed over all the years of that simulation run. If the resulting indicators are tables or maps stored
in a file, they can then be found on disk in your “OPUSHOME/data/PROJECTNAME/runs/RUNNAME/indicators” directory, where “PROJECTNAME” is the name of your project (e.g. “eugene_gridcell”) and “RUNNAME” is the name of the simulation run that you selected to run the batch on. The indicator visualizations configured to write to a database will have produced tables in the specified database with the name of the respective indicator visualization.

Create, configure, and execute a new indicator batch:

1. Create a new indicator batch by right-clicking on the “Indicator_batches” node in the Results tab and selecting the appropriate option.
2. Add an indicator visualization configuration to that batch. Right-click on your new indicator batch and select “Add new indicator visualization”.
3. Configure a Map visualization that contains the zone_job_density and zone_population_density indicators for the zone dataset.
4. Close the batch visualization configure dialog.
5. Right-click on the batch and execute your indicator batch on the results of a simulation run.

Indicator visualization configuration options

Opus provides a variety of ways to visualize indicators and this functionality is exposed in the “Indicator visualization” dialog box options (e.g. multi-year indicators, exporting to databases). This section describes the range of available options in the Batch indicator visualization dialog box, which is separated into three components: “indicator selection”, “output options”, and “format options” (See Figure 10.4).

**Indicator selection**

The bottom of the dialog box has two list boxes, “available indicators” and “indicators in current visualization”. The indicators here are those variables defined in the “Variable Library” (Chapter 4) whose use has been set to be indicator or both. Note that the set of indicators available is filtered by the currently selected dataset in the “output options” (described later in this section).

By moving an indicator from the “available indicators” box to the “indicators in current visualization” box via the “+” button, you include that indicator in this indicator visualization. Likewise, to remove an indicator from the visualization, select the indicator in the “indicators in current visualization” box and press the “-” button.

**Output options**

*Visualization Name.* The base name of any produced visualizations. Because you might be producing this visualization for different years and different simulation data, more information will be appended to this name to ensure uniqueness of the resulting file or database table when the visualization is run on some data.

*Type.* There are two different types of indicator visualizations that can be produced: maps and tables. Tables are just raw data organized into rows and columns, while maps are spatial projections of this data. The available format options (described later) are fully dependent on the visualization type.

*Dataset name.* The dataset that this visualization corresponds to. When the selected indicator(s) are run, they will be computed over this dataset. Most commonly you are choosing a geographic granularity (e.g. gridcell, zone) that you want to see the results at. Note that when you change the dataset, the set of available indicators changes because a given indicator is valid only for a single dataset.
Figure 10.4: The batch visualization creation dialog.

Format options for maps

Map visualizations will produce a map file for every selected indicator for every available year when it is executed on a simulation run.

The available map format is Mapnik, an open-source toolkit for developing mapping applications (http://mapnik.org/). Note that the Mapnik maps are not intended to replace GIS-based mapping, which allows far more control and the overlay of other features for visual reference. It is merely a quick tool to visualize data to get a sense of the spatial patterns in it. In order to support visualization in a GIS environment such as ArcGIS or QGIS, the results may be exported to a database or geodatabase environment, and the GIS software used to create a more interactive and flexible display of the data. See the following section for a description of how to export indicator results to a SQL database or a DBF file for use in external GIS tools.
Export the results that were found in the previous tutorial inset to a SQL database.

1. Make sure that you have configured a database server. From the Tools menu, select “Database Server Connections”. Check to see that the “indicators_database_server” is correctly set up. If you don’t have a remote database server, make sure that it points to a sqlite connection. Close the connections dialog box.

2. Reconfigure the batch to write to a database. Expand the indicator batch that you defined in the prior step. Right-click on the visualization and select “Configure visualization”. Change the format to “Export to SQL database” and then name a database it should write to. Hit OK and then rerun the batch on the simulation results from before.

3. Launch a database browser and check to see if the proper tables were created.

To set the Mapnik map options, first open the batch visualization creation dialog window (shown above) by clicking on the Results tab, right-clicking “Indicator_batches”, selecting the “Add new indicator batch…” option, right-clicking the new indicator batch, and selecting the “Add new indicator visualization…” option. Select “Map” from the “Type:” drop-down menu, then click the “Mapnik Map Options” button that will appear directly below the “Format:” drop-down menu. In the Mapnik Map Options dialog window, there are two tabs where all of the mapping options can be set: Legend Options and Size Options.
Legend Options tab:

Figure 10.5: The map options dialog window showing default settings (Legend Options).

**Number of Color Ranges.** This sets the number of buckets to split the range of the values being mapped into. The drop-down menu allows you to choose a number between 1 and 10.

*Color Scaling Type.* This drop-down menu allows you to choose between “Linear Scaling”, “Custom Scaling”, “Custom Linear Scaling”, and “Equal Percentage Scaling”. Linear scaling will evenly divide the range of values to be mapped into buckets of the same size. Custom scaling will let you specify how to divide the range of values into the buckets. Custom linear scaling will let you specify a minimum and maximum value and then will evenly divide the range of values into buckets of the same size. Equal Percentage Scaling will divide the range of values up so that (100/number of buckets) percent of the values fall into each category.

*Custom Scale.* If “Custom Scaling” is selected as the Color Scaling Type, then the values in this text field will be used to define the bucket ranges. For example, if the following string is entered in the Custom Scale text field “5, 100, 2000, 40000”, then the bucket ranges will be ‘5 to 100’, ‘100 to 2000’, and ‘2000 to 40000’. Also, if either “min”, “MIN”, “max”, or “MAX” is entered in the Custom Scale text field, they will be replaced with the minimum or maximum values of the values that are being mapped. For example, ”min,100,max” is a valid entry for Custom Scale.

*Label Type.* This drop-down menu allows you to choose between “Range Values” and “Custom Labels” to use as the labels on the color bar that will be drawn in to the map image. Using range values means that the boxes in the color bar will be labeled with the range of values that are being colored with the color contained in the corresponding box. Using custom labels allows you to manually enter the labels for each box.

*Custom Labels.* If “Custom Labels” is selected as the Label Type, then the values in this text field will be used to
label the colored boxes in the color bar. For example, if the following string is entered in the Custom Labels text field “a,b,c,”, then the first box will be labeled “a”, the second will be labeled “b”, and the third will be labeled “c”.

**Color Scheme.** This drop-down menu allows you to choose between “Custom Color Scheme”, “Green”, “Blue”, “Red”, and “Custom Graduated Colors”. The color buttons in the legend are buttons, that when pushed, cause a color chooser dialog window to pop up that can be used to pick the color of the box. If Custom Color Scheme is selected, then all manually-set colors will be saved when the “Apply Changes” button is pushed, whereas all colors will be over-written when the “Apply Changes” button is clicked if any other color scheme option is selected. If Green, Blue, or Red is selected, the boxes will be colored with pre-defined colors. Lastly, if Custom Graduated Colors is selected, then the boxes will be colored in an even, graduated scale where the range of colors starts at the current color of the box at index 1 and ends at the current color of the box at the highest index.

**Diverge Colors On.** This gives you the option to have two diverging colors. You have to option “None” to split on none of the indices to use just one color scale, or you have the option to split on any index from 1 to 10. The color box at the selected index will be set to white and the two color scales above and below it will be set based on what color option is selected in the Color Scheme and Diverging Color drop-down menus. Note that if “Custom Graduated Colors” is selected in the Color Scheme drop-down menu, then the two color scales above and below the box at the diverging index will have graduated color scales and the option selected in the Diverging Color drop-down menu will be ignored.

**Diverging Color.** This drop-down menu allows you to choose between “Red”, “Green”, and “Blue”. This will set the color for the boxes with an index number lower than the index number selected in the Diverge Colors On drop-down menu. Note: any selection made in this menu will be ignored if “Custom Color Scheme” is selected in the Color Scheme drop-down menu.

**Legend.** The legend has three columns: Color, Range, and Label. The Color column has color boxes that display the color currently selected for the corresponding color bucket. Also, clicking a color box will bring up a dialog window that will allow you to select a custom color to display on the color box. The Range column has numerical values that define the range of the corresponding color bucket and these values can be edited within the legend table if “Custom Scaling” is selected in the Color Scaling Type drop-down menu. The Label column has the label that will be applied to each color bucket on the color bar that will be included in the map. These values are editable from within the legend table if “Custom Labels” is selected in the Label Type drop-down menu. Note: values entered in the legend table will only be saved correctly if they appear correctly in the Custom Scale and Custom Labels text fields.
Size Options tab:

Figure 10.6: The map options dialog window showing default settings (Size Options).

Enter Your Computer’s Resolution (DPI). This value is used to convert from inch measurements to pixel measurements (pixels = inches x DPI). The resolution for the computer that the map is being created on should be entered here.

Page Dimensions (inches). Available dimensions for the map image file (map and legend) are listed here in the drop-down menu.

Map Page Placement. Lower left and upper right coordinates for a bounding box on the map image can be entered here. The map image will be placed in the center of the given bounding box and will only be drawn as large as possible without being stretched or skewed. Coordinates should be entered in the form “x,y” and the coordinate (0,0) is at the lower left corner.

Legend Page Placement. Lower left and upper right coordinates for the color bar of the legend can be entered here. The color bar will fill the area defined by the given coordinates and then the labels will be drawn to the right of the color bar.

Mapnik Options Window Buttons:

Apply Changes. This will save the current settings to the project XML file as well as update the Legend showing in this window.

Close Window. This will close the options window and discard any unsaved settings.
Examples of how to create default and custom maps

Create a default map. To create a map with the default settings (using the Eugene gridcell project), create an indicator batch and in the batch indicator visualization dialog window, set Type to “Map”, set Dataset name to “zone”, and select “number_of_jobs” to be included in the current visualization. Then run the indicator batch on a simulation by right-clicking the indicator batch in the results tab and selecting “Run indicator batch on...”. This will produce a map with the default settings of 10 color ranges, colored green, where each color range is the same size and linearly increasing from light green to dark green. For more instructions on how to create and run an indicator batch see section 10.2.2. The map will look like the map shown below.

![Mapnik map colored with the default color scheme.](image)

Figure 10.7: A Mapnik map colored with the default color scheme.
Create a custom map. To create a customized map, open the batch indicator visualization window (see section 10.2.2 on batch indicator configuration for information on how to do this), for the map created in the Create a default map section and click on the Mapnik Map Options button that is directly below the Format drop-down menu.

Suppose that we would like to create the same map again, but this time with 9 color ranges, custom ranges, custom labels, and a diverging color scheme where all values from 0 to 300 are colored with gradually lighter shades of blue and all values from 300 to 4000 are colored with gradually darker shades of red. As a side note, the option to use a diverging color scheme exists primarily to allow the coloring scheme to differentiate between positive and negative values. However, all values in this example are all greater than or equal to zero.

Select “9” in the Number of Color Ranges drop-down menu, select “Custom Scaling” in the Color Scaling Type menu, enter “0,50,100,150,200,300,400,500,1000,4000” in the Custom Scale text field, select “Custom Labels” in the Label Type menu, enter “a,b,c,d,e,f,g,h,i” in the Custom Labels text field, select “Red” in the Color Scheme menu, select “6” in the Diverge Colors On menu, select “Blue” in the Diverging Color menu, and press the Apply Changes button.

The color box at the index specified in the Diverge Colors On menu will be set to white. This color can be changed to a light pink color by clicking on the color box in the row specified in the Diverge Colors On menu to bring up a dialog window that lets you choose a new color. From the color chooser dialog window, select a light pink color and press OK. Check to make sure the selected option in the Color Scheme menu has been changed to “Custom Color Scheme” so that your custom color for the color box in row 6 will be saved when the Apply Changes button is pressed, and then press the Apply Changes button. If “Custom Color Scheme” is not selected, then when the Apply Changes button is pressed, all colors will be reset based on the color scheme options that is currently selected. The mapnik options dialog window should look like the screenshot pictured below. The map that will be created after the indicator batch has been run is also shown below. (see section 10.2.2 for information on how to run an indicator batch)

![Mapnik Options Window](image)

Figure 10.8: The Mapnik map options window showing custom settings.
Figure 10.9: A Mapnik map with a custom color scheme.
Create a custom graduated color scheme. To create a map with a custom graduated color scale, re-open the Mapnik Options dialog window for the map created in the Create a custom map section. Click on the box in row 1 and select a light shade yellow, click on the box in row 9 and select a bright shade of blue, select “Custom Graduated Colors” from the Color Scheme menu, and click Apply Changes. The colors in boxes 2 through 8 will then be automatically set to create a graduated color scheme that spans from light yellow to bright blue. The mapnik options window and resulting map are shown below. (see section 10.2.2 for information on how to run an indicator batch)

![Mapnik Options Window](image)

Figure 10.10: The Mapnik map options window showing a custom graduated color scheme.

10.2. Interrogating Results with Indicators
Figure 10.11: A Mapnik map with a custom graduated color scheme.
Create a diverging custom graduated color scheme. The graduated color schemes can also diverge on an index. Shown below is the Mapnik options dialog window with the same settings as set in the Create a custom graduated color scheme section, but the selected option in the Diverge Colors On has been changed from “None” to “5”.

Figure 10.12: The Mapnik map options window showing a diverging custom graduated color scheme.
Create a map with attributes that are True/False values. To create a map with attributes that are True/False values rather than numerical values (using the Seattle parcel project), create an indicator batch in the batch indicator visualization dialog window, set Type to “Map”, set Dataset name to “parcel”, and select “hwy2000” to be included in the current visualization. (see chapter 4 “The Variable Library” for information on how to use the variable library to add hwy2000 to the list of available indicators.) Then click the Mapnik Map Options button that is directly below the Format drop-down menu.

In the Mapnik Options dialog window, set the options as shown in the figure below. Number of Color Ranges: “2”, Color Scaling Type: “Custom Linear Scaling”, Custom Scale: “0,1”, Label Type: “Custom Labels”, Custom Labels: “False,True”, Color Scheme: “Custom Color Scheme”. Lastly, in the legend, click the color box in row 1 to set the color to red and click the color box in row 2 to set the color to blue. Press the apply changes button to save these settings. The map that will be created after the indicator batch has been run is also shown below. (see section 10.2.2 for information on how to run an indicator batch)

Note: The scale is “0,1” because ‘False’ and ‘True’ are numerically represented as ’0’ and ’1’ respectively.

Figure 10.13: The Mapnik map options window showing settings for a True/False map.
Figure 10.14: A Mapnik map showing a map with True/False attributes.
Format options for animations

Animation visualizations will produce an animation file for every selected indicator that repeatedly loops through map images that correspond to the years the batch was selected to run on when it is executed on a simulation run.

The available animation format is Mapnik, an open-source toolkit for developing mapping applications (http://mapnik.org/).

To set the Mapnik map options, first open the batch visualization creation dialog window by clicking on the Results tab, right-clicking “Indicator_batches”, selecting the “Add new indicator batch...” option, right-clicking the new indicator batch, and selecting the “Add new indicator visualization...” option. Select “Animation” from the “Type:” drop-down menu, then click the “Mapnik Map Options” button that will appear directly below the “Format:” drop-down menu. In the Mapnik Map Options dialog window, all of the animation options can be set.

All of the animated map options are the same as the non-animated map options. See the above section on format options for maps for more info.

Format options for tables

There are four different available formats for tables. Each has its own parameters that need to be set. Note that the id column for the dataset will automatically be included in all outputted tables regardless of format.

Tab-delimited (.tab). This will output a file (or multiple files) where the values are separated by tabs. There are three different modes to choose from that affects how data is split across files when the visualization is executed on a simulation run. “Output in a single file” will create single tab file that has a column for each selected indicator for each year of the simulation run. “Output in a table for every year” will create a tab file for each year of the simulation run, with each column corresponding to a selected indicator. Finally, “Output a table for each indicator” will create a tab file for each selected indicator, where each column corresponds to the indicator values of a year of the simulation run.

Fixed-field. The fixed field format will output a single file whose fields are written with fixed width and no delimiters. The file contains a column for each selected indicator for each year of the simulation run for which the visualization is being created. Format info for each column needs to be specified. To specify the format of the dataset id column, fill in the “id_col” input field. To specify the format of each selected indicator, enter the format in the respective row of the “field format” column in the “indicators in current visualization” box. The field format has two parts, the length of the field and the type. Available types are float ( “f”) and integer ( “i”). Specified field formats follow the pattern “10i” and “5.2f”, where the latter specifies a float with five leading digits with floating point precision carried out to two decimal places.

SQL database. This format option can be used to export to an arbitrary SQL database. The database server used is that specified in the “Database server connections” under “indicators_database” (see Section 5.3). The exported data will take the form of a newly created in the specified database (if the database doesn’t exist, it will be created first). The SQL table will contain a column for every selected indicator for every year of the simulation run that it is being executed against. The name of the table is a combination of the name of the visualization and the name of the simulation run. Additionally, if you are exporting to a PostGRES database and have an existing spatial table corresponding to the dataset of the visualization, a view defining a join over the spatial table and the indicator table will automatically be created. This allows you to instantly view the indicator results in QGIS.

ESRI database. This option exports the indicator data to an ESRI database that can be loaded into ArcMap. Simply specify the path to a geodatabase file (.gdb). It is assumed that the geodatabase contains a Feature Class corresponding to the dataset of the indicator being exported. Once the export is successfully completed, the geodatabase will contain a table that contains the indicator result, with a zone_id and an ArcGIS OBJECTID that corresponds to the internal object ids in the feature class. It is safe to join the indicator table result with the feature class using either the objectid or the zone_id. See Figure 10.15 for an example exported indicator after following this process.
Figure 10.15: Mapping a Population by Zone Indicator in ESRI ArcMap. Shows the result of joining the feature class with the indicator table and generating a thematic map of the population by zone, using the zone.acres field to normalize the population, resulting in a map of population density per acre.
Inheriting XML Project Configuration Information

As previously discussed, Opus uses XML-based configuration files to flexibly specify the various aspects of a project, as well as to specify the appearance of the different tabs in the GUI.

One configuration (the child) can inherit from another configuration (the parent). By default, the child inherits all the information contained in the project. However, it can override any information inherited from the parent, and add additional information. This means that you can use default projects as parents for another project you want to create that is mostly the same as an existing project, but has some changes from it. An XML configuration specifies its parent using the parent entry under the “General” section of the XML. The value of this is the name of the XML file that contains the parent configuration. When searching for this file, Opus first looks in the same directory that holds the child configuration. If it’s not found there, it expects to find a path in the Opus source code tree, starting with the name of a project like eugene or urbansim.

This works well with the convention that users should create their own projects in the ‘opus/project_configs’ directory. You can have several configurations in your ‘opus/project_configs’ directory, one inheriting from the other. Ultimately, though, one or more of these configurations should inherit from a default configuration in the source code tree in ‘opus/src’. For example, Figure 11.1 shows the contents of the ‘eugene_gridcell_default.xml’ project in ‘opus/project_configs’. This has almost no content of its own, but rather inherits everything but the description from the parent configuration in the source code tree at ‘eugene/configs/eugene_gridcell.xml’.

```
<opus_project>
  <xml_version>4.2.0</xml_version>
  <general>
    <description type="string">Minimal user configuration for the Eugene gridcell project</description>
    <parent type="file">eugene/configs/eugene_gridcell.xml</parent>
  </general>
</opus_project>
```

Figure 11.1: Contents of the default eugene_gridcell.xml project

A small section of its parent, ‘eugene/configs/eugene_gridcell.xml’, is shown in Figure 11.2. This is just a text file, but in a structured format, with nodes corresponding to information that is displayed in the GUI. Some of the content of the XML provide data used by the GUI to determine how to display information, or what menu items are appropriate to connect to the node in the GUI. Notice that this project in turn inherits from ‘urbansim_gridcell/configs/urbansim_gridcell.xml’.
To change any of the options in the XML tree you need to be able to edit it. Inherited parts of the XML tree are shown in blue in the left pane in the GUI. These can’t be edited directly — they just show inherited information. You can make an inherited portion of the XML tree editable by right clicking on it and selecting “Add to current project.” This copies the inherited information into the local XML tree, which then is shown in black. We saw an example of doing this in Section 9.1.3. Figure 9.6 in that same section shows an XML tree that is mostly inherited (and so shown in blue).

A few details about the “Add to current project” command: in Section 9.1.3, when we added lastyear to the current project, the containing nodes in the tree (years_to_run and Eugene_baseline) also turned black, since they needed to be added to the current project as well to hold lastyear. It’s also possible to click directly on years_to_run, or even Eugene_baseline, and add the XML tree under that to the current project. However, we recommend adding just the part you’re editing to the current project, and not others. (You can always add other parts later.) The reason is that once a part of the tree is added to the current project, the inheritance relation of that part with the parent disappears, and changes to the parent won’t be reflected in the child XML. For example, if you add all of the Eugene_baseline node to the current project, save your configuration, and then update the source code, changes to some obscure advanced feature in the XML in the parent in the source tree wouldn’t show up in your configuration.

When you first start up the GUI and open a project, we suggested starting with the default ‘eugene_gridcell_default.xml’ configuration. You can use the “Save as” command to save it under a new file name, and in that way keep multiple configurations in your ‘opus/project_configs’ directory. You can change the parent of a configuration by editing its parent field under the “General” tab — if you do this, save the configuration and then re-open it so that the GUI will read in the new parent information (it currently doesn’t do that automatically). Finally, since configurations are just files, you can copy them, save them on a backup directory, or email them as attachments to other users. (You can also edit them directly with a text editor — but only do this if you know what you’re doing, since there won’t be any checks that after your edits the configuration is still well-formed.)
Part III

OPUS Data and Models
Data in Opus

Models written in OPUS (like any other models) operate on data, and OPUS provides extensive support for handling a wide variety of data types and formats as inputs and outputs. The OPUS GUI, as shown in the preceding part of the documentation, provides an interactive interface to process data and models in the OPUS environment. In this chapter we begin to make more explicit the details behind the GUI, beginning with data.

The way most users interact with data is in tables, whether they are stored as spreadsheets, comma-separated ASCII files, binary files, dbf files, or database tables in a DBMS like MySQL or Postgres or MSSQL. A table has rows and columns, with the columns representing attributes and the rows representing observations. A table of households, for example, might contain columns for a unique household_id, income, number of persons, number of workers, etc. The rows would each represent one household, with its values for each attribute.

OPUS has its own internal data management capabilities, built on the Numpy Python library for array processing. This is one of the keys to the relatively high performance of models implemented in OPUS: array-based calculations. OPUS reads data from standard tables in external formats such as databases or ASCII files, and translates them into Numpy arrays. Each column in a table gets translated into a one-dimensional array: think of it like a table with only one column, that can be loaded into memory and operated on very quickly. The OPUS system stores these translated tables, called Datasets in OPUS terminology to distinguish the array-based storage from data outside of the OPUS environment, as a set of files on disk, organized within a directory that corresponds to a table name, and with one file per attribute in the Dataset. For the most part, the inner workings of the data storage and processing in OPUS is not something a user needs to understand in too much detail, but it helps to understand a few aspects of this approach to data handling in OPUS.

12.1 Primary and Computed Attributes

With this preliminary introduction, we proceed to use the term Dataset to represent the OPUS representation of a table.

A Primary Attribute, reflecting the contents of a single columns in the table, is represented in OPUS as a single array, and stored in one binary file in Numpy format on disk. Note that OPUS keeps track of the indexing of these arrays on disk and in memory, to avoid losing the connection across the primary attributes corresponding to each observation (row in a table). A primary attribute is one that is in the data to be loaded into a model. It exists when the models are initialized.

Computed Attributes are also Numpy arrays, one per attribute. We distinguish attributes that are computed as variables, or expressions from primary attributes. The OPUS system has implemented a sophisticated way to avoid redundant calculations by keeping track of what is already computed, and re-using it if it is still available and has not been made obsolete by some other calculation. This process of re-using already computed attributes is called caching, and is very helpful to reducing computation. This distinction and record-keeping also allows the system to write data to disk from
memory, in order to reload and restart a model at a later time, saving space by not storing computed attributes (unless they are predicted future values of primary attributes).

The next chapter will describe in much more depth the facilities offered by OPUS to implement computed attributes using variables and expressions.

### 12.2 Importing and Exporting Data

The GUI contains a set of tools in the data tab to load data into OPUS from external data sources, and to write OPUS to external data targets. These external data storage options include:

- Tab-delimited ASCII
- Comma-separated ASCII
- MySQL
- MS SQL
- Postgres
- SQLite
- PostGIS
- ArcGIS Geodatabases, including SDE, Personal and Flat File Geodatabases
- DBF formatted tables

See also Section 5.3 for information on configuring database server connections in the GUI.
Defining Variables using Expressions

As described in Chapter 12, datasets contain primary and computed attributes. Opus variables are used to represent the required computation for a computed attribute. Underneath, Opus variables are defined as Python classes. However, writing Python code is a significant barrier to using the system, and in many cases Opus variables can be defined instead as simple expressions in a custom domain-specific programming language (called Tekoa). When this can be done, it is much simpler than defining them in Python itself.¹

### 13.1 Informal Introduction

An expression consists of the name of an attribute, or a function or operation applied to other expressions. Thus, a more complex expression can be built up from simpler expressions. The functions and operations are ones from the Numpy numerical Python package, and operate on arrays. Here are some examples.

- gridcell.distance_to_highway<300 generates a dummy variable whose value is 1 for gridcells that have an attribute value of distance to highway less than 300 meters, and otherwise 0. In this case, we are operating on a primary attribute of gridcells, namely gridcell.distance_to_highway<300. This means that this attribute is part of the data that we initially load into the model, as opposed to data we compute within the model (computed attributes).
- \( \ln(\text{urbansim.gridcell.population}) \) computes the log of an existing variable, \text{population}, which is in the \text{urbansim} package and applies to the gridcell dataset.
- \( \text{urbansim.gridcell.population} / \text{urbansim.gridcell.number_of_jobs} \) computes the ratio of population to employment, using variables stored in the \text{urbansim} package, associated with the gridcell dataset.

Two useful methods in the language are aggregate and disaggregate. Aggregation allows an expression to compute a result on one dataset and aggregate the results to assign to another dataset, such as summing the population in households that live in a gridcell. For example, this variable defines the population in each zone, by aggregating the number of people in each gridcell contained in that zone:

\[
\text{zone.aggregate(urbansim.gridcell.population)}
\]

Sometimes multiple levels of geography are required in the aggregation. For example, the following variable computes the population for each parcel:

¹See [Borning et al., 2008] for a discussion of the impact on code size reduction and other benefits of using Tekoa.
parcel.aggregate(household.persons, intermediates = [building])

In the parcel-based models (e.g. the Seattle_parcel sample model and data), households are assigned to buildings, and buildings are assigned to parcels — so to find the number of people living on a parcel we first aggregate to the building level, then to the parcel level.

Working in the opposite direction, the **disaggregate** method assigns values from one dataset to another, in a one to many relationship. (In contrast the aggregate method is many to one.) An example is to assign the zonal population density to all households living in the zone. In this example, we use the `population_density` variable in the zone dataset of the package urbansim_parcel package, and assign it to households, which are connected to zones indirectly through buildings and then parcels: household $\rightarrow$ building $\rightarrow$ parcel $\rightarrow$ zone. So the expression is

```python
disaggregate(household, urbansim_parcel.zone.population_density, intermediates = [building, parcel])
```

Note that for both aggregate and disaggregate methods, the first element, preceding the method name, is the name of the dataset to which the result should be assigned. `zone.aggregate(...) should generate some result and assign it to zone, whereas household.disaggregate(...) should assign some value from a larger geography to the household dataset.

The Tekoa expression language has been added to Opus fairly recently, and there are many variables in the existing urbansim package that could be rewritten as expressions rather than using Python classes. (We plan to replace these with expressions in specifications in the future; but in case you run across some Python variable classes that look like they might as well be simple expressions, that’s the reason.)

### 13.2 Syntax

The syntax of Tekoa is a subset of Python syntax, but the semantics are somewhat different — for example, a name like `gridcell.distance_to_cbd` is a reference to the value of that variable. (If you just evaluated this in a Python shell you’d get an error, saying that the `gridcell` package didn’t have a `distance_to_cbd` attribute.) Further, expressions are evaluated lazily (in other words, values are computed only when they are needed).

An expression consists of an attribute name, or a function or operation applied to other expressions. Thus, a more complex expression can be built up from simpler expressions. For example, here are some legal expressions:

- `gridcell.population`
- `ln(gridcell.population)+1`

### 13.3 Variable Names in Expressions

The attribute names used in an expression can be:

- a fully-qualified variable name (for a variable defined as a Python class). Example: `urbansim.gridcell.population`.
- a dataset-qualified variable name (for variables in the XML expression library) or primary attribute. Example: `household.income`.
- an attribute of a constant dataset. (Constant datasets are identified syntactically — their names end in the string `_constant`.) Example: `urbansim_constant.near_highway_threshold`. 
The variable names can include arguments (Section 23.3.4). For example, `is_near_highway_if_-threshold_is_2` matches the variable definition for `is_near_SSS_if_threshold_is_DDD`.2

13.4 Unary Functions for Opus Expressions

There is a set of unary functions defined to use in expressions, as follows. These all operate on numpy arrays.

- **clip_to_zero** Returns the input values with all negative values clipped to 0.
- **exp** Returns an array consisting of \(e^x\) raised to the input values.
- **ln** Returns an array of natural logarithms. Input values of 0 result in 0. (The intent is that this function be used on arrays of values, where 0 denotes a missing value. However, be cautious — as you approach 0 from the positive side, the result value becomes more and more negative, and then suddenly returns to 0 at 0.)
- **ln_bounded** Returns an array of natural logarithms. Values less than 1 result in 0.
- **ln_shifted** Returns an array of natural logarithms. The input values are shifted by the second argument before taking the log. (The default shift value is 1.)
- **ln_shifted_auto** If the input values includes values that are less than or equal to 0, they are shifted so that the minimum of the shifted values is 1 before taking the log. Otherwise the log is taken on the original values.
- **sqrt** Returns an array of square roots. Values less than 0 result in 0.
- **safe_array_divide** The first three arguments are the nominator, denominator and a constant. The function returns an array of numerator / denominator for all values where denominator is not 0, otherwise the constant (default value is 0). An optional fourth argument controls the type of the resulting array. The default value is ‘float32’.

In addition, all of the functions in Numpy are available. To avoid name collisions, the function name in an expression must include the package name `numpy`. For example, this expression gives you the reciprocals of all the values in a variable \(v\):

\[\text{numpy.reciprocal}(v)\]

13.5 Operators for Opus Expressions

All of the numpy operators can be used in Opus expressions, including `+` `-` `*` `/` `**` (** denotes exponentiation). Note the numpy semantics for these — for example, `*` does elementwise multiplication of two numpy arrays, or with a scalar argument, scales all the elements in an array, e.g. `1.2*\text{household.income}`.

13.6 Casting the Value of an Expression

Normally the value of a variable defined by an expressions will be a `float64` (this is a numpy type). For large datasets this may use too much space. You can cast the result of any expression to a different type using the numpy `astype` method. For example, the type of this expression is `float32`:

\[\text{numpy.reciprocal}(v)\]

---

2This syntax for arguments to variables is not ideal — we hope to replace it with a more standard syntax in the future.
ln(urbansim.gridcell.population).astype(float32)

The following numpy types can be used as the argument to the astype method: bool8, int8, uint8, int16, uint16, int32, uint32, int64, uint64 float32, float64, complex64, complex128, and longlong.

13.7 Interaction Sets

If you access an attribute of a component of an interaction set in the context of that interaction set, the result is converted into a 2-d array and returned. These 2-d arrays can then be multiplied, divided, compared, and so forth, using the numpy functions and operators. For example, suppose we have an interaction set household_x_gridcell. The component household set has an attribute income, and the gridcell component has an attribute cost. Then

urbansim.gridcell.cost * urbansim.household.income

evaluates to a 2-d array of cost times income. See Section 23.3.6 for sample data illustrating this.

Both the arguments to the operation and the result can be used in more complex expressions. For example, if we wanted to give everyone a $5000 income boost, and also scale the result, this could be done using (household.income+5000) * gridcell.cost * 1.2.

13.8 Aggregation and Disaggregation

The methods aggregate and disaggregate are used to aggregate and disaggregate variable values over two or more datasets. The aggregate method associates information from one dataset to another along a many-to-one relationship, while the disaggregate method does the same along a one-to-many relationship. Some examples are:

- zone.aggregate(gridcell.population)
- zone.aggregate(2.5*gridcell.population)
- zone.aggregate(urbansim.gridcell.population)
- neighborhood.aggregate(gridcell.population, intermediates=[zone, faz])
- neighborhood.aggregate(gridcell.population, intermediates=[zone, faz], function=mean)
- zone.aggregate(gridcell.population, function=mean)
- region.aggregate_all(zone.my_variable)
- region.aggregate_all(zone.my_variable, function=mean)
- faz.disaggregate(neighborhood.population)
• `gridcell.disaggregate(neighborhood.population, intermediates=[zone, faz])`

Here is a description of the syntax for these two methods. Also see Section 23.3.7 for sample data and results for the example computations below.

**Aggregation**

Suppose we have three different geographical units: gridcells, zones and neighborhoods. We have information available on the gridcell level and would like to aggregate this information for zones and neighborhoods. We know the assignments of gridcells to zones and of zones to neighborhoods.

An aggregation over one geographical level for the `locations` attribute ‘capacity’ can be done by:

```
zone.aggregate(gridcell.capacity)
```

By default, the aggregation function applied to the aggregated data is the ‘sum’ function. This can be changed by passing the desired function as second argument:

```
zone.aggregate(urbansim.gridcell.is_near_cbd_if_threshold_is_2, function=maximum)
```

The `aggregate` method accepts the following aggregation functions: sum, mean, variance, standard_deviation, minimum, maximum, center_of_mass. These are functions of the scipy package `ndimage`.

An aggregation over two or more levels of geography is done by passing a third argument into the `aggregate` method. It is a list of dataset names over which it is aggregated, excluding datasets for the lowest and highest level. For example, aggregating the gridcell attribute ‘capacity’ for the neighborhood set can be done by:

```
neighborhood.aggregate(gridcell.capacity, function=sum, intermediates=[zone])
```

**Disaggregation**

Disaggregation is done analogously. The `disaggregate` method takes information from a coarse set of entities and allocates it to a finer set of entities, in the manner of a one-to-many relationship. By default, the function for allocating data is to simply replicate the data on the parent entity for each inheriting entity. The method takes one required argument, an attribute/variable name, and one optional argument, a list of dataset names. Here we distribute an attribute “is_cbd” from the neighborhood set across gridcells:

```
gridcell.disaggregate(neighborhood.is_cbd, intermediates=[zone])
```

To aggregate over all members of one dataset, one can use the built-in method `aggregate_all`. It must be used with a dataset that has one element which is the case of the `opus_core` dataset `AllDataDataset` implemented in the directory ‘datasets’. For example, the total capacity for all gridcells can be determined by:
In addition to sum, the aggregate_all class accepts all functions that are accepted by the aggregate class; the default is sum.

If the attribute being aggregated or disaggregated is a simple variable, it should be either dataset-qualified or fully-qualified, i.e. always including the dataset name and optionally including the package name. The attribute being aggregated can also be an expression. (In this case, behind the scenes the system generates a new variable for that expression, and then uses the new variable in the aggregation or disaggregation operations. However, this isn’t visible to the user.) The result of an aggregation or disaggregation can also be used in more complex expressions, e.g.

\[ \ln(2 \times \text{aggregate(gridcell.population)}) \].

### 13.9 Number of Agents

A common task in modeling is to determine a number of agents of one dataset that are assigned to another dataset. For this purpose, Opus contains a built-in method number_of_agents, which takes as an argument the name of the agent dataset. For example, the number of households in each location can be determined by:

```python
gridcell.number_of_agents(household)
```

Similarly, the number of zones in neighborhoods is computed by:

```python
neighborhood.number_of_agents(zone)
```

As in the case of aggregate and disaggregate, the number_of_agents method must be preceded by the ‘owner’ dataset name, e.g. neighborhood.number_of_agents for computing on the neighborhood dataset.

### 13.10 Principal Dataset of an Expression

Every expression has a single principal dataset to which it applies. For example, the principal dataset for ln(gridcell.population) is gridcell. Generally, it wouldn’t make sense for an expression to have more than one principal dataset — for example, the expression gridcell.population+zone.population doesn’t make sense semantically.

There are a few additional aspects of this rule:

- A constant dataset doesn’t count as a principal dataset — for example, gridcell.distance_to_highway<urbansim_constant.near_highway_threshold is a meaningful expression, whose principal dataset is gridcell. (urbansim_constant.near_highway_threshold returns an array of length 1, and numpy replicates that value for each gridcell, so that the value of the whole expression is an array of booleans of the same length as the gridcell dataset.)

- For an aggregate or disaggregate expression, the principal dataset is the dataset being aggregated or disaggregated to, i.e. the dataset on which the aggregate or disaggregate method is being called. For example, zone is the principal dataset for zone.aggregate(gridcell.population).
• For expressions involving interaction sets, different subexpressions can reference one or the other of the datasets being interacted. For example, the principal dataset for `urbansim.gridcell.cost * urbansim.household.income` is `household_x_gridcell`.

13.11 Equality of Expressions

Two expressions are equal if their defining strings are identical, ignoring spaces. Thus these two expressions are equivalent:

```
urbansim.gridcell.population + 1
urbansim.gridcell.population + 1
```

However, two textually different expressions are not equivalent, even if they are algebraically equal, and the system will treat them as two different variables. For example, `1 + urbansim.gridcell.population` is different from the previous expressions. In many cases this doesn’t matter. Reasons it may matter are: (1) if the resulting value uses a lot of memory or takes a long time to compute, having a second copy of the value will waste memory or computation time; and (2) if the variable defined by the expression is used in a specification, you could inadvertently end up with two variables. For this reason, good practice is to put each expression that you’ll need for a given package and dataset in the expression library. Elsewhere use its name. (Also see Section 13.12.)

13.12 Names and Aliasing

The expression library includes a name field for each expression. This is used to bind the name to the corresponding expression. When using such expressions intermixed with Python code, another mechanism is needed to give a name (or alias) to an expression – this is done with something much like an assignment statement, for example:

```
lnpop = ln(urbansim.gridcell.population)
```

Since this functionality is only relevant when writing expressions as part of Python code, rather than using the GUI and the expression library, further discussion of this aspect of the Tekoa language is deferred to Section 23.3.5.
14.1 Overview

Opus provides infrastructure to develop, specify, estimate, diagnose and predict with a variety of model types. The Opus GUI currently supports the creation of several types of models by providing templates that can be copied and configured. More will be added in the future. These initial types are:

- Simple Models
- Sampling Models
- Allocation Models
- Regression Models
- Choice Models
- Location Choice Models

14.2 Simple Models

The simplest form of a model in Opus is called, for lack of imagination, Simple Model. It is about as simple as a model can get: compute a variable and write the results to a dataset. Here are some examples of what could be done with a simple model:

- Aging Model: add one to the age of each person, each year
- Walkability Model: write the result of an expression that evaluates the amount of retail activity within walking distance
- Redevelopment Potential Model: compute the ratio of improvement to total value for each parcel and write this to the parcel dataset

A Simple Model template is available in the Model Manager, and can be copied and configured in order to create a new Simple Model like the examples above. It takes only three arguments in its configuration:

- Dataset: the Opus Dataset that the result will be written to
- Outcome Attribute: the name of the attribute on the Dataset that will contain the predicted values
- Expression: the Opus Expression that computes the result to be assigned to the outcome attribute
14.3 Sampling Models

The second type of model template is a Sampling Model. This generic model takes a probability (a rate), compares it to a random number, and if the random number is larger than the given probability (rate), it assigns the outcome as being chosen. Some examples will make the use of this model clearer. Say we want to build a household evolution model. We need to deal with aging, which we can do with a Simple Model. We also models that predict:

- Births
- Deaths
- Children leaving home as they age
- Divorces
- Entering the labor market
- Retiring

For all of these examples – assuming that we want to base our predictions on expected rates that vary by person or household attributes – we need a more sophisticated model that we shall call a Sampling Model. Since we need to assign a tangible outcome rather than a probability, we use a sampling method to assign the outcome in proportion to the probability. This method is also occasionally referred to as a Monte Carlo Sampling algorithm.

The algorithm is simple. Let’s say we have a probability of a coin toss, heads or tails each having a probability of 0.5. A sampling model to predict an outcome attribute of Heads, would take the expected probability of a fair coin toss resulting in an outcome of Heads as being 0.5. We then draw a random number from a univariate distribution between 0 and 1, and compare it to the expected probability. If the random draw is greater than the expected probability, then we set the choice outcome to Heads. If it is less than 0.5, then we set the choice outcome to Tails. Since we are drawing from a univariate random distribution between 0 and 1, we would expect that around half of the draws would be less than 0.5 and half would be greater than this value. Larger numbers of draws will tend to converge towards the expected probability by the law of large numbers. A very large number of draws should match the expected probability to a very high degree of precision.

To make the model useful for practical applications, we can add a means to apply different probabilities to different subsets of the data. For example, death rates or birth rates vary by gender, age, and race/ethnicity, and to some extent by income. We might want to stratify our probabilities by one or more of these attributes, and then use the sampling model to sample outcomes using the expected probabilities for each subgroup.

The Sampling Model takes the following arguments:

- Outcome Dataset: the name of the dataset to receive the predicted values
- Outcome Attribute: the name of the attribute to contain the predicted outcomes
- Probability Dataset: the name of the dataset containing the probabilities
- Probability Attribute: the name of the attribute containing the probability values (or rates)
- List of Classification Attributes: attributes of Outcome Dataset that will be used to index different Probabilities (e.g. age and income in household relocation)

14.4 Allocation Models

Another simple generic model supported in Opus is the Allocation Model, which does not require estimating model parameters. This model proportionately allocates some aggregate quantity to a smaller unit of analysis using a weight. This model could be configured, for example, to allocate visitor population estimates, military population, nursing home population, and other quantities to traffic analysis zones for use in the travel model system. Or it could be
used to build up a simplistic incremental land use allocation model (though this would not contain much behavioral content).

The algorithm for this type of model is quite simple. To create an Allocation Model, we need to specify six arguments:

- Dataset to contain the new computed variable
- Name of the new computed variable $Y$, which will be indexed by the ids of the dataset it is being allocated to, $Y_i$.
- Dataset containing the total quantity to be allocated (this can contain a geographic identifier, and will include a year column).
- Variable containing the control total to be allocated, $T$
- Variable containing the (optional) capacity value $C$, indexed as $C_i$
- Variable containing the weight to use in the allocation $w$, indexed as $w_i$, with a sum across all $i$ as $W$

The algorithm is then just:

$$Y_i = \min(\text{round}(TW_i/W), C_i)$$

(14.1)

If a capacity variable is specified, we add an iterative loop, from $m$ to $M$, to allocate any excess above the capacity to other destinations that still have remaining capacity:

$$T^m = \sum(\text{round}(TW_i/W) - C_i)$$

(14.2)

In each iteration, we exclude alternatives where $Y >= C$, and repeat the allocation with the remaining unallocated total:

$$Y^m_i = Y^{m-1}_i + (T^m_w/W)$$

(14.3)

We then iterate over $m$ until $T^m = 0$

This simple algorithm is fairly versatile, and can be used in two modes: as incremental growth or as total values. If used in incremental mode, it adds the allocated quantity to the existing quantities. The alternative, total, mode for this model replaces the quantities with the new predicted values.

### 14.5 Regression Models

Regression models are available to address problems in which the dependent variable is continuous, and a linear equation can be specified to predict it. The primary use of this model in a core model in UrbanSim is the prediction of property values. In the context of predicting property values, the model is referred to as a hedonic regression [Waddell et al., 1993], but the Opus regression model is general enough to address any standard multiple regression specification. Other examples of applications for this basic class of models would be to predict water or energy consumption per household, or parking prices.

The basic form is:

$$Y_i = \alpha + \beta X_i + \epsilon_i$$

(14.4)
where $X_i$ is a matrix of explanatory, or independent, variables, and $\beta$ is a vector of estimated parameters. Opus provides an estimation method using Ordinary Least Squares, and additional estimation methods are available by interfacing Opus with the R statistical package. For the current discussion, we focus on working with the built-in estimation capacity.

14.6 Choice Models

Many modeling problems do not have a continuous outcome, or dependent variable. It is common to have modeling problems in which the outcome is the selection of one of a set of possible discrete outcomes, like which mode to take to work, or whether to buy or rent a property. This class of problem we will refer to as discrete choice situations, and we develop choice models to address them.

Recall from Section 2.4 that the standard multinomial logit model [McFadden, 1974, McFadden, 1981] can be specified as:

$$P_i = \frac{e^{V_i}}{\sum_j e^{V_j}},$$

where $j$ is an index over all possible alternatives, $V_i = \beta \cdot x_i$ is a linear-in-parameters function, $x_i$ is a vector of observed, exogenous, independent alternative-specific variables that may be interacted with the characteristics of the agent making the choice, and $\beta$ is a vector of $k$ coefficients estimated with the method of maximum likelihood [Greene, 2002].

The multinomial logit model is a very robust and widely used model in practical applications in transportation planning, marketing, and many other fields. It is easy to compute and is therefore fast enough to use on large-scale computational problems such as residential location choice. For explanatory purposes, we will focus initially on choice problems with small numbers of alternatives, such as the choice to rent or own a house, or the number of vehicles a household will choose to own.

Note that there are limitations to the MNL model, and assumptions a user should be aware of. The most important of these is the Independence of Irrelevant Alternatives (IIA) property, which implies that adding or eliminating an alternative from a choice set will affect all of the remaining alternatives proportionately. Stated another way, the relative probabilities of any two alternatives will be unaffected by adding or removing another alternative. See [Train, 2003] for a thorough introduction to discrete choice modeling using MNL and other choice model specifications.

We now turn to a tutorial for creating models of some of these types using the Opus GUI. In the following sections, we provide a worked example of creating a new model of each type. The other model types follow the same pattern in the Opus GUI.
Creating Synthetic Households for OPUS Modeling Applications

Disclaimer: The household synthesizer described in this chapter is functional, but has not been fully integrated into the OPUS GUI, and some of the tools to support pre-processing data to use in the synthesizer, and to post-process synthesizer results to make them fully usable, are not done yet. Stay tuned for this to be completed in the next maintenance release, 4.2.1.

UrbanSim and other models such as activity-based travel model are microsimulation models that simulate the choices of individual households and persons. In order to operationalize such models, a synthetic household population is needed. Although there are now several household synthesizers in existence and use, it was deemed important to tightly integrate a household synthesizer into OPUS in order to support UrbanSim and activity-based travel modeling within the OPUS environment.

A new household synthesizer has been developed and contributed to the OPUS user community through a collaboration with Ram Pendyala and Karthic Konduri at Arizona State University. A paper submitted for presentation to TRB 2010 describes the algorithm and results on a test application. The paper is *A methodology to match distributions of both household and person attributes in the generation of synthetic populations*, by Xin Ye, Karthic Konduri, Ram Pendyala, Bhargava Sana, and Paul Waddell.

The UrbanSim group at UW has begun the process of fully integrating the synthesizer into OPUS, by adding an interface in the GUI, and adding supporting tools to create the inputs for the synthesizer, and to post process outputs from the synthesizer. The synthesizer uses inputs from the U.S. Census by census block group, and from the Public Use Microdata sample. Tools have been coded to create these inputs from standard census data files, and to set up and launch the synthesizer. These tools are visible in the tools section of the data tab in the GUI. They are now in testable condition but have not been extensively tested.

The synthesizer creates a household table and a persons table, with categories for several characteristics and a geographic identifier of a census block group. Post-processing tools allow the imputation of point values for categorical characteristics such as income, and also for assigning synthetic households to specific buildings. The latter step is currently being integrated into the GUI, and depends on having an estimated household location choice model, since this model is used to place the households that are assigned to a block group into a specific building.

The table below provides an example of the household and person characteristics used in the initial development and testing of the synthesizer. See the paper for details of the algorithm and for validation results.
<table>
<thead>
<tr>
<th>Household Attributes</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Household Type</td>
<td>Family: Married Couple</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Family: Male Householder, No Wife</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Family: Female Householder, No Husband</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Non-family: Householder Alone</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Non-family: Householder Not Alone</td>
<td>5</td>
</tr>
<tr>
<td>Household Size</td>
<td>1 Person</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2 Persons</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3 Persons</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4 Persons</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5 Persons</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6 Persons</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7 or more Persons</td>
<td>7</td>
</tr>
<tr>
<td>Household Income</td>
<td>$0 - $14,999</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$15,000 - $24,999</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$25,000 - $34,999</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>$35,000 - $44,999</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$45,000 - $59,999</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>$60,000 - $99,999</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>$100,000 - $149,999</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Over $150,000</td>
<td>8</td>
</tr>
<tr>
<td>Person attributes</td>
<td>Gender</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Male</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Female</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Age</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Under 5 years</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5 to 14 years</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>15 to 24 years</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>25 to 34 years</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>35 to 44 years</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>45 to 54 years</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>55 to 64 years</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>65 to 74 years</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>75 to 84 years</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>85 and more</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Ethnicity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>White alone</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Black or African American alone</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>American Indian and Alask Native alone</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Asian alone</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Native Hawaiian and Other Pacific Islander alone</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Some other race alone</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Two or more races</td>
<td>7</td>
</tr>
</tbody>
</table>
Part IV

UrbanSim Applications
Sample UrbanSim Projects

To facilitate learning how UrbanSim works, sample projects and data have been made available for download and use. Note that these are not in any way operational, but are for demonstration purposes only. A zone-based sample application is in development, and will be added once it is complete.

16.1 Eugene Gridcell Project

Project name: eugene-gridcell-default

The Eugene Gridcell project is one of the two initial sample projects that are available. It is based on data compiled by the Lane Council of Governments, and approximates conditions in 1980 for the Eugene-Springfield, Oregon metropolitan area. The project is an example of an UrbanSim application of the first generation, using gridcells as the unit of geography, with 150 by 150 meter resolution. The models were developed as part of the development of the initial prototype of UrbanSim approximately during 1996 - 1998. Many of the models have been modified extensively since that time, for example, as shown in the Seattle-parcel project.

The Eugene gridcell project can be loaded in the GUI, and a baseline simulation can be run using the models as specified previously. At this time the specification of the models has not been added into the GUI, and therefore the models cannot easily be re-specified or estimated within the GUI. As time permits, this will be added for demonstration purposes.

The main advantage and use of the Eugene project is that it is small and runs quickly, so it provides a quick means to test an installation of OPUS and UrbanSim by running a simulation. Note that little time or effort has been available to fine tune the Eugene application, and the specifications and coefficients are from the initial calibration of the model using a base year of 1994. So one should not rely heavily on the results, but use the project as intended: a simple demonstration. It also can be of some use for exploring the data used in the gridcell-based applications.

16.2 Seattle Parcel Project

Project name: seattle-parcel-default

Recent developments in UrbanSim have included the development of substantial flexibility in the use of geographic units of analysis. A new model system has been developed for the Puget Sound Regional Council (PSRC), and the previous version of UrbanSim based on gridcells (like the Eugene application) has been converted to use parcels and buildings. The Seattle parcel project has been generated as a second example project, by taking a snapshot of the
PSRC model system and data, and extracting a subset of the database containing only information within the City of Seattle. In addition, the employment data, which in the PSRC application is confidential, has been replaced with synthetic employment data derived from County Business Patterns by Zip Code for use in the Seattle parcel project.

The main objective in making the Seattle parcel project available to the user community is the same as the original sample project: to make the models and data available for examination by users that may wish to create an application based on the configuration of this project. Note that the simulation results are not likely to be robust, since the original specifications were from the calibration of the model on the full Puget Sound four county region.

One last caveat for this project: the specifications included in the GUI for models that can be estimated are simply place-holders, and allow a user to experiment with specifying and estimating models in the GUI. Note that specifying and estimating models in the GUI, if the save results flag is set to true, will overwrite stored specifications and modify simulation results. This is not a significant concern since this is a sample, experimental project, and not for production use.
UrbanSim Models and Data Structures

17.1 Geographic Units of Analysis in UrbanSim Models

UrbanSim has evolved over several years, and so have the data structures used in it. Until 2005, most applications used a geographic framework based on a grid overlaid on the study area. Most UrbanSim applications to date have used this approach to structuring the data. Various limitations of the gridcell approach led to the more recent adoption of a parcel-based data and spatial structure. This section explains the fundamental differences in these data structures, and assesses their relative merits. Both are still used and supported approaches, though the advantages of the parcel approach are fairly significant. We begin with an overview of the three different data structures that UrbanSim now supports: gridcells, parcels, and zones.

17.1.1 Gridcells

The gridcell-based approach to developing the data for UrbanSim, as the name suggests, begins with the decision of a resolution to use for a grid to overlay the study area. There is no definitively correct grid cell resolution, and a pragmatic choice of 150 meters by 150 meters was chosen in early UrbanSim applications, mainly as a compromise between the high level of resolution desired, and the increased computational demands made by higher resolution data.

Figure 17.1 depicts a portion of the study area in the Puget Sound, and shows the 150 meter grid initially used for the PSRC model application superimposed on other planning geographies for relative comparison. It is obvious that grid cells bisect parcels, or to put it another way, that it is not possible to aggregate parcel information neatly into gridcells. This was an unfortunate but obvious outcome of imposing a completely regular shape (a grid) on a polygonal layer of parcels that vary in size in shape. The main advantage of using a grid is that is makes it possible to use raster processing efficiently, as is done in image processing or raster GIS spatial analysis. It is possible to compute quite efficiently, for example, how much population or employment is within a fixed radius of each cell. This computational efficiency was the main initial motivation for using the gridcell approach to structuring the input data for UrbanSim.

Each gridcell contains approximately 5 1/2 acres, at a resolution of 150 meters. In order to prepare the data for UrbanSim, parcel maps were overlaid in the GIS on a vector representation of gridcells, and the contents of the parcel (housing, etc) allocated to the gridcells in proportion to the amount of its land area falling within each gridcell. The fragments of the real estate components created in this way were then aggregated into a composite at the cell level. UrbanSim then operates on the gridcell-level data. In order to better reflect the contents of the grid cells, which are clearly not homogeneous in their composition, building objects were created to allow at least different types of real estate in a cell to be represented by different types of buildings. Households and jobs were then associated with buildings, and buildings with gridcells, as shown in Figure 17.2.

The main disadvantage of the gridcell data structure has already been mentioned: it requires unnaturally splitting
the underlying parcel information and recombining it in ways that create artificial representations of the data. This problem also makes it difficult to apply information on development regulations from general plans, since those are also based on polygons, and in fact, apply to parcels.

17.1.2 Parcels

To address some of the limitations of the gridcell-based data structure, recent development of UrbanSim has adopted a data structure based on parcels. The parcel-based UrbanSim application uses a data model that reflects parcels, buildings, households and jobs as the primary objects and units of analysis. Households and jobs choose locations by selecting a specific building, which is associated with a specific parcel. Real estate development is based on development projects occurring on specific parcels. In the most recent extensions of the Puget Sound model, persons have been added to the data model, and workers are associated with specific jobs. These data relationships are shown in Figure 17.2, and make clear that only the link of buildings to a geographic unit is changed.

17.1.3 Zones

Given the available flexibility in configuring models in OPUS, an alternative data structure can be readily substituted for parcel or gridcell based data. For example, locations could be defined by zones used in the travel model, to make them consistent with current-generation zone-based travel models. This approach can also be used to create a rather simple model system using less geographic detail. For example, an application in Paris used Communes, or administrative areas roughly equivalent to traffic analysis zones in number (approximately 1300).

By using the same data structure for households, jobs and buildings, the only change needed for a zone-based model system is to assign locations to buildings at a coarser level of detail. This retains all of the accounting systems in the
UrbanSim model: households and jobs are still located in buildings, and buildings can be linked spatially to zones. But this approach would give up considerable detail for analyzing development locations and capacity constraints due to zoning or land use plans. For testing purposes, a zone-based model system has been implemented in UrbanSim and is being prepared for access via the GUI.

17.2 UrbanSim Model Components

The overall architecture of the UrbanSim model system is depicted in Figure 17.3. While most of the early applications of UrbanSim used gridcells, and more recent ones have adopted the use of parcels and buildings, the overall logic contained in Figure 17.3 remains intact. What differs is the configuration of specific models. The models used in the parcel version of UrbanSim differ in some obvious respects from the earlier gridcell versions, and these are summarized in Table 17.1. In addition to the substitution of parcels for gridcells as the unit of analysis, the real estate development model was completely restructured to take advantage of the availability of parcel geography in representing actual development projects - which do vary in size and shape in the real world, in ways that were difficult to reconcile with gridcell geography. The parcel based model specifications also have recently added models to predict the choice of workers to be home-based (normally work from home), and a workplace choice model for workers who are not home-based. This allows for the first time the more appropriate handling of the prediction of commuting behavior as a long-term outcome of where a household chooses to live, and where the workers in the household have jobs, and allows the removal of the home-based-work trip distribution model from the set of behaviors predicted by the travel model on a daily basis.

In the remainder of this chapter, the components of the current version of UrbanSim are described in some detail in terms of their structure and algorithms. Since many (but not all) of these models are based on a discrete choice framework, a brief explanation of the common basis for these models is presented first. Where model specifications vary by geographic basis (e.g. gridcell or parcel), deviations are described. First, the common underpinnings of many of the models, discrete choice models, is presented as an overview.

17.2.1 Economic Transition Model

Employment is classified by the user into employment sectors based on aggregations of Standard Industrial Classification (SIC) codes, or more recently, North American Industry Classification (NAICS) codes. Typically 10 to 20 sectors are defined based on the local economic structure. Aggregate forecasts of economic activity and sectoral employment are exogenous to UrbanSim, and are used as inputs to the model. These forecasts may be obtained from state economic forecasts or from commercial or in-house sources.
Figure 17.3: Overview of UrbanSim Model System
Table 17.1: Specification of UrbanSim Model Components Using Parcel Data Structure

<table>
<thead>
<tr>
<th>Model</th>
<th>Agent</th>
<th>Dependent Variable</th>
<th>Functional Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Household Location Choice</td>
<td>Household (New or Moving)</td>
<td>Residential Building</td>
<td>Multinomial Logit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Vacant Unit</td>
<td></td>
</tr>
<tr>
<td>Employment Location Choice</td>
<td>Job (New or Moving)</td>
<td>Non-residential Building</td>
<td>Multinomial Logit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With Vacant Space</td>
<td></td>
</tr>
<tr>
<td>Home-based Job Choice</td>
<td>Worker (Without Job)</td>
<td>Binary Choice (Work at Home)</td>
<td>Binary Logit</td>
</tr>
<tr>
<td>Workplace Choice</td>
<td>Non Home Based Worker (Without Job)</td>
<td>Vacant Job</td>
<td>Multinomial Logit</td>
</tr>
<tr>
<td>Real Estate Development</td>
<td>Development Proposal</td>
<td>Parcel (With Vacant Land)</td>
<td>Multinomial Logit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vacant</td>
<td>Sampler</td>
</tr>
<tr>
<td>Real Estate Price</td>
<td>Parcel</td>
<td>Price Per Square Foot</td>
<td>Multiple Regression</td>
</tr>
</tbody>
</table>

The base year UrbanSim employment data for the Puget Sound application are derived from data from the State of Washington Employment Securities Division, generally referred to as ES202 data, containing unemployment insurance administrative records with information on the location and size of business establishments. InfoUSA is a commonly used private source for similar data, compiled from credit reports and other sources.

The Economic Transition Model integrates exogenous forecasts of aggregate employment by sector with the UrbanSim database by computing the sectoral growth or decline from the preceding year, and either removing jobs from the database in sectors that are declining, or queuing jobs to be placed in the employment location choice model for sectors that experience growth. If the user supplies only total employment control totals, rather than totals by sector, the sectoral distribution is assumed consistent with the current sectoral distribution. In cases of employment loss, the probability that a job will be removed is assumed proportional to the spatial distribution of jobs in the sector. The jobs that are removed vacate the space they were occupying, and this space becomes available to the pool of vacant space for other jobs to occupy in the location component of the model. This procedure keeps the accounting of land, structures, and occupants up to date.

New jobs are not immediately assigned a location. Instead, new jobs are added to the database and assigned a null location, to be resolved by the Employment Location Choice Model.

17.2.2 Demographic Transition Model

The Demographic Transition Model accounts for changes in the distribution of households by type over time, using an algorithm analogous to that used in the Economic Transition Model. In reality, these changes result from a complex set of social and demographic changes that include aging, household formation, divorce and household dissolution, mortality, birth of children, migration into and from the region, changes in household size, and changes in income, among others. The data (and theory) required to represent all of these components and their interactions adequately are complex, and this set of behaviors remain to be implemented in UrbanSim. Instead, the Demographic Transition Model, like the Economic Transition Model described above, uses external control totals of population and households by type (the latter only if available) to provide a mechanism for the user to approximate the net results of these changes. Analysis by the user of local demographic trends may inform the construction of control totals with distributions of household size, age of head, and income. If only total population is provided in the control totals, the model assumes that the distribution of households by type remains static.

As in the economic transition case, household births are added to a list of movers that will be located by the Household Location Choice Model. Household deaths, on the other hand, are accounted for by this model by removing those
households from the housing stock, and by properly accounting for the vacancies created by their departure. The demographic transition model is analogous in form to the employment transition model described above.

### 17.2.3 Employment Relocation Model

Employment relocation and location choices are made by firms. However, in the current version of UrbanSim, we use individual jobs as the units of analysis. This is equivalent to assuming that businesses are making individual choices about the location of each job, and are not constrained to moving an entire establishment.

The Employment Relocation Model predicts the probability that jobs of each type will move from their current location or stay during a particular year. This is a transitional change that could reflect job turnover by employees, layoffs, business relocations or closures. Similar to the economic transition model when handling job losses in declining sectors, the model assumes that the hazard of moving is proportional to the spatial distribution of jobs in the sector. All placement of jobs is managed through the employment location model.

As in the case of job losses predicted in the economic transition component, the application of this model requires subtracting jobs by sector from the buildings they currently occupy, and the updating of the accounting to make this space available as vacant space. These counts will be added to the unallocated new jobs by sector calculated in the economic transition model. The combination of new and moving jobs serve as a pool to be located in the employment location choice model. Vacancy of nonresidential space will be updated, making space available for allocation in the employment location choice model.

Since it is possible that the relative attractiveness of commercial space in other locations when compared with an establishment’s current location may influence its decision to move, an alternative structure for the mobility model could use the marginal choice in a nested logit model with a conditional choice of location. In this way, the model would use information about the relative utility of alternative locations compared to the utility of the current location in predicting whether jobs will move. While this might be more theoretically appealing than the specification given, it is generally not supported by the data available for calibration. Instead, the mobility decision is treated as an independent choice, and the probabilities estimated by annual mobility rates directly observed over a recent period for each sector.

### 17.2.4 Household Relocation Model

The Household Relocation Model is similar in form to the Employment Relocation Model described above. The same algorithm is used, but with rates or coefficients applicable to each household type. For households, mobility probabilities are estimated from the Census Current Population Survey, which provides a national database on which annual mobility rates are computed by type of household. This will reflect differential mobility rates for renters and owners, and households at different life stages.

Application of the Household Relocation Model requires subtracting mover households by type from the housing stock by building, and adding them to the pool of new households by type estimated in the Demographic Transition Model. The combination of new and moving households serves as a population of households to be located by the Household Location Choice Model. Housing vacancy is updated as movers are subtracted, making the housing available for occupation in the household location and housing type choice model.

An alternative approach would be to implement a relocation model as a choice model, and specify and estimate it using a combination of household and location characteristics. This could be linked with the location choice model, or done as a joint model. This remains to be done in future development.
17.2.5 Employment Location Choice Model

In this model, we predict the probability that a job that is either new (from the Economic Transition Model), or has moved within the region (from the Employment Mobility Model), will be located at a particular site. Buildings are used as the basic geographic unit of analysis in the current model implementation. Each job has an attribute of space it needs, and this provides a simple accounting framework for space utilization within buildings. The number of locations available for a job to locate within a building will depend mainly on the total square footage of nonresidential floorspace in the building, and on the density of the use of space (square feet per employee).

Given the possibility that some jobs will be located in residential units, however, housing as well as nonresidential floorspace must be considered in job location. We have allowed the user to specify the control totals for employment by sector in two categories: home-based and non-home-based. The model is specified as a multinomial logit model, with separate equations estimated for each employment sector.

For both the employment location and household location models, we take the stock of available space as fixed in the short run of the intra-year period of the simulation, and assume that locators are price takers. That is, a single locating job or household does not have enough market power to influence the transaction price, and must accept the current market price as given.

The variables included in the employment location choice model are drawn from the literature in urban economics. We expect that accessibility to population, particularly high-income population, increases bids for retail and service businesses. We also expect that two forms of agglomeration economies influence location choices: localization economies and inter-industry linkages.

Localization economies represent positive externalities associated with locations that have other firms in the same industry nearby. The basis for the attraction may be some combination of a shared skilled labor pool, comparison shopping in the case of retail, co-location at a site with highly desirable characteristics, or other factors that cause the costs of production to decline as greater concentration of businesses in the industry occurs. The classic example of localization economies is Silicon Valley. Inter-industry linkages refer to agglomeration economies associated with location at a site that has greater access to businesses in strategically related, but different, industries. Examples include manufacturers locating near concentrations of suppliers in different industries, or distribution companies locating where they can readily service retail outlets.

One complication in measuring localization economies and inter-industry linkages is determining the relevant distance for agglomeration economies to influence location choices. At one level, agglomeration economies are likely to affect business location choices between states, or between metropolitan areas within a state. Within a single metropolitan area, we are concerned more with agglomeration economies at a scale relevant to the formation of employment centers. The influence of proximity to related employment may be measured using two scales: a regional scale effect using zone-to-zone accessibilities from the travel model, or highly localized accessibilities using queries of the area immediately around the given parcel. Most of the spatial queries used in the model are of the latter type, because the regional accessibility variables tend to be very highly correlated, and because agglomerations are expected to be very localized.

Age of buildings is included in the model to estimate the influence of age depreciation of commercial buildings, with the expectation that businesses prefer newer buildings and discount their bids for older ones. This reflects the deterioration of older buildings, changing architecture, and preferences, as is the case in residential housing. There is the possibility that significant renovation will make the actual year built less relevant, and we would expect that this would dampen the coefficient for age depreciation. We do not at this point attempt to model maintenance and renovation investments and the quality of buildings.

Density, the inverse of lot size, is included in the location choice model. We expect businesses, like households, to reveal different preferences for land based on their production functions and the role of amenities such as green space and parking area. As manufacturing production continues to shift to more horizontal, land-intensive technology, we expect the discounting for density to be relatively high. Retail, with its concentration in shopping strips and malls, still requires substantial surface land for parking, and is likely to discount bids less for density. We expect service firms
to discount for density the least, since in the traditional urban economics models of bid-rent, service firms generally
outbid other firms for sites with higher accessibility, land cost, and density.

We might expect that certain sectors, particularly retail, show some preference for locations near a major highway,
and are willing to bid higher for those locations. Distance to a highway is measured in meters, using grid spatial
queries. We also test for the residual influence of the classic monocentric model, measured by travel time to the CBD,
after controlling for population access and agglomeration economies. We expect that, for most regions, the CBD
accessibility influence will be insignificant or the reverse of that in the traditional monocentric model, after accounting
for these other effects.

Estimation of the parameters of the model is based on a geocoded establishment file (matched to the parcel file to link
employment by type to land use by type). A sample of geocoded jobs in each sector is used to estimate the coefficients
of the location choice model. As with the Household Location Choice Model, the application of the model produces
demand by each employment type for building locations.

The independent variables used in the employment location choice model can be grouped into the categories of real
estate characteristics, regional accessibility, and urban-design scale effects as shown below:

- **Real Estate Characteristics**
  - Prices
  - Development type (land use mix, density)

- **Regional accessibility**
  - Access to population
  - Travel time to CBD, airport

- **Urban design-scale**
  - Proximity to highway, arterials

- **Local agglomeration economies within and between sectors: center formation**

17.2.6 Household Location Choice Model

In this model, as in the employment location model, we predict the probability that a household that is either new
(from the transition component), or has decided to move within the region (from the mobility component), will choose
a particular location defined by a residential building. As before, the form of the model is specified as multinomial
logit, with random sampling of alternatives from the universe of available (vacant) housing units, including those units
vacated by movers in the current year.

The model architecture allows location choice models to be estimated for households stratified by income level, the
presence or absence of children, and other life cycle characteristics. Alternatively, these effects can be included in a
single model estimation through interactions of the household characteristics with the characteristics of the alternative
locations. The current implementation is based on the latter but is general enough to accommodate stratified estimation,
for example by household income. The variables used in the model are drawn from the literature in urban economics,
urban geography, and urban sociology. An initial feature of the model specification is the incorporation of the classical
urban economic trade-off between transportation and land cost. This has been generalized to account not only for travel
time to the classical monocentric center, the CBD, but also to more generalized access to employment opportunities
and to shopping. These accessibilities to work and shopping are measured by weighting the opportunities at each
destination zone with a composite utility of travel across all modes to the destination, based on the logsum from the
mode choice travel model.

These measures of accessibility should negate the traditional pull of the CBD, and, for some population segments,
potentially reverse it. In addition to these accessibility variables, we include in the model a net building density, to
measure the input-substitution effect of land and capital. To the extent that land near high accessibility locations is bid
up in price, we should expect that builders will substitute capital for land and build at higher densities. Consumers for whom land is a more important amenity will choose larger lot housing with less accessibility, and the converse should hold for households that value accessibility more than land, such as higher income childless households.

The age of housing is considered for two reasons. First, we should expect that housing depreciates with age, since the expected life of a building is finite, and a consistent stream of maintenance investments are required to slow the deterioration of the structure once it is built. Second, due to changing architectural styles, amenities, and tastes, we should expect that the wealthiest households prefer newer housing, all else being equal. The exception to this pattern is likely to be older, architecturally interesting, high quality housing in historically wealthy neighborhoods. The preference for these alternatives are accommodated through a combination of nonlinear or dummy variable treatment for this type of housing and neighborhood.

A related hypothesis from urban economics is that, since housing is considered a normal good, it has a positive income elasticity of demand. This implies that as incomes rise, households will spend a portion of the gains in income to purchase housing that is more expensive, and that provides more amenities (structural and neighborhood) than their prior dwelling. A similar hypothesis is articulated in urban sociology in which upward social mobility is associated with spatial proximity to higher status households. Both of these hypotheses predict that households of any given income level prefer, all else being equal, to locate in neighborhoods that have higher average incomes. (UrbanSim does not attempt to operationalize the concepts of social status or social assimilation, but does consider income in the location choice.)

The age hypothesis and the two income-related hypotheses are consistent with the housing filtering model, which explains the dynamic of new housing construction for wealthy households that sets in motion a chain of vacancies. The vacancy chain causes households to move into higher status neighborhoods than the ones they leave, and housing units to be successively occupied by lower and lower status occupants. At the end of the vacancy chain, in the least desirable housing stock and the least desirable neighborhoods, there can be insufficient demand to sustain the housing stock and vacancies go unsatisfied, leading ultimately to housing abandonment. We include in the model an age depreciation variable, along with a neighborhood income composition set of variables, to collectively test the housing filtering and related hypotheses.

Housing type is included in the model as a set of dummy variables for alternative housing types. These are discussed further in Section 17.2.8 describing the real estate development model.

One of the features that households prefer is a compatible land use mix within the neighborhood. It is likely that residential land use, as a proxy for land uses that are compatible with residential use, positively influences housing bids. On the other hand, industrial land use, as a proxy for less desirable land use characteristics, would lower bids.

The model parameters are estimated using a random sample of alternative locations, which has been shown to provide consistent estimates of the coefficients. In application for forecasting, each locating household is modeled individually, and a sample of alternative cell locations is generated in proportion to the available (vacant) housing. Monte carlo simulation is used to select the specific alternative to be assigned to the household, and vacant and occupied housing units are updated in the cell.

The market allocation mechanism used to assign households and jobs to available space, then, is not done through a general equilibrium solution in which we assume consumers and suppliers optimize across all alternatives based on perfect information, and zero transaction costs, with prices on all buildings at each location adjusting to the general equilibrium solution that perfectly matches consumers and suppliers to clear the market. Rather, the solution is based on an expectation of incomplete information and nontrivial transactions and search costs, so that movers obtain the highest satisfactory location that is available, and prices respond at the end of the year to the balance of demand and supply at each location.

The independent variables can be organized into the three categories of housing characteristics, regional accessibility, and urban-design scale effects as shown below.

- Housing Characteristics
Prices (interacted with income)
Development types (density, land use mix) Housing age

• Regional accessibility
  Job accessibility by auto-ownership group
  Travel time to CBD and airport

• Urban design-scale (local accessibility)
  Neighborhood land use mix and density
  Neighborhood Employment

17.2.7 Real Estate Price Model

UrbanSim uses real estate prices as the indicator of the match between demand and supply of land at different locations and with different land use types, and of the relative market valuations for attributes of housing, nonresidential space, and location. This role is important to the rationing of land and buildings to consumers based on preferences and ability to pay, as a reflection of the operation of actual real estate markets. Since prices enter the location choice utility functions for jobs and households, an adjustment in prices will alter location preferences. All else being equal, this will in turn cause higher price alternatives to become more likely to be chosen by occupants who have lower price elasticity of demand. Similarly, any adjustment in land prices alters the preferences of developers to build new construction by type of space, and the density of the construction.

We make the following assumptions:

1. Households, businesses, and developers are all price-takers, and market adjustments are made by the market in response to aggregate demand and supply relationships. Each responds, therefore, to previous period price information.

2. Location preferences and demand-supply imbalances are capitalized into land values. Building value reflects building replacement costs only, and can include variations in development costs due to terrain, environmental constraints or development policy.

Real estate prices are modeled using a hedonic regression of the log-transformed property value per square foot on attributes of the parcel and its environment, including land use mix, density of development, proximity of highways and other infrastructure, land use plan or zoning constraints, and neighborhood effects. The hedonic regression may be estimated from sales transactions if there are sufficient transactions on all property types, and if there is sufficient information on the lot and its location. An alternative is to use tax assessor records on land values, which are part of the database typically assembled to implement the model. Although assessor records may contain biases in their assessment, they do provide virtually complete coverage of the land (with notable exceptions and gaps for exempt or publicly owned property).

The hedonic regression equation encapsulates interactions between market demand and supply, revealing an envelope of implicit valuations for location and structural characteristics [DiPasquale and Wheaton, 1996]. Prices are updated by UrbanSim annually, after all construction and market activity is completed. These end of year prices are then used as the values of reference for market activities in the subsequent year.

The independent variables influencing land prices can be organized into site characteristics, regional accessibility, and urban-design scale effects, as shown below:

• Site characteristics
  Development type
  Land use plan
  Environmental constraints

Chapter 17. UrbanSim Models and Data Structures
• Regional accessibility
  Access to population and employment

• Urban design-scale
  Land use mix and density
  Proximity to highway and arterials

17.2.8 Real Estate Development Model

Development of residential and non-residential real estate property is modeled by Real Estate Development Model. There are two different versions of Real Estate Development Model: the Development Project Location Choice Model and the Development Project Proposal Sampling Model, used by UrbanSim in the gridcell-based and parcel-based model system respectively.

Development Project Location Choice Model

The Development Project Location Choice Model models real estate development as a process where real estate property developers seek the best available site for their real estate projects. A collection of real estate projects are sampled from recent real estate development history according to the market condition and are then assigned to the most favorable locations available. Similar to the employment location choice model and household location choice model, we predict the probability that a project will choose a particular location. The form of the model is specified as multinomial logit, with random sampling of alternatives from the universe of feasible locations (gridcells where such development is allowed by development constraints).

The independent variables used in the development project location choice model can be organized into categories of site characteristics, urban design-scale effects, regional accessibility, and market conditions, as shown below:

• Site characteristics
  Existing development characteristics
  Land use plan
  Environmental constraints

• Urban design-scale
  Proximity to highway and arterials
  Proximity to existing development
  Neighborhood land use mix and property values
  Recent development in neighborhood

• Regional accessibility
  Access to population and employment
  Travel time to CBD, airport

• Market Conditions
  Vacancy rates

Development Project Proposal Sampling Model

The Development Project Proposal Sampling Model models real estate development as real estate project proposals competing for financing. A project proposal proposes developing a predefined real estate template (or prototype) on a given site, and a group of profit-maximization financial agents then pick proposals to finance with random utility maximization process. The concept of development template is borrowed from that of “standard” product type in real
We use development templates to describe the mix of building types and intensities of development, and other characteristics such as approximate scale of project, and fraction of land lost as ‘overhead’, for example for parking or for internal streets and rights of way in a subdivision.

We assume the financial agents’ behavior is motivated by profit (they attempt to maximize their profits), within constraints imposed by the physical environment and public land use regulations. The main factors on their choices will then be factors influencing revenue from proposed projects, the costs of producing the projects, and the constraints relevant at those sites.

We derive a sampling process that is consistent with random utility maximization framework [Wang and Waddell, 2008].

The urban area is divided into infinite number of non-overlapping development sites, indexed by \( i = 1, \ldots, I \). Each site has a bundle of characteristics \( x_i \), such as the market value of the property, lot size, and development regulation. A predefined set of real estate development templates \( j = 1, \ldots, J \), which categorized the development patterns in an urban area and can be as detailed as needed (though with some computational cost for more detail). Each development template is characterized by attributes \( y_j \).

Define a proposed development project \( D_{ij} \) as the combination of development site \( i \) and development template \( j \) when development constraints allow such a development project at this site:

\[
D = \{(i, j) | q_i \geq q_{ij}, \quad i = 1, \ldots, I; \quad j = 1, \ldots, J\},
\]

where \( Q_i(\cdot) \) maps the development site characteristics to a \( K \)-dimension vector \( q_i \) that constrains development, while \( Q_j(\cdot) \) calculates this \( K \)-dimension measure \( q_{ij} \) for the proposed development. A combination of development site \( i \) and development template \( j \) is in the proposed development project set \( D \) only when the development template is allowed at the site for each dimension of the constraints \( q_{ik} \geq q_{jk} \), for \( k = 1, \ldots, K \).

Given a combination of development site \( i \) and development template \( j \) in \( D \), we can calculate the expected hedonic price and construction cost of the proposed development:

\[
p_{ij} = P(x_i, y_j), \tag{17.2}
\]
\[
c_{ij} = C(x_i, y_j), \tag{17.3}
\]

for \( i, j \in D \)

Assuming the utility of selecting a proposed development project equals a function of \( p_{ij} \) and \( c_{ij} \) plus an unobserved random component:

\[
u_{ij} = V(p_{ij}, c_{ij}) + \varepsilon_{ij}, \quad \text{subject to} \quad p_{ij} - c_{ij} > 0 \tag{17.4}
\]

With assumption that \( \varepsilon_{ij} \) is i.i.d Gumbel distributed, we have a specification that is consistent with random utility maximization theory:

\[
Pr(ij|D) = Pr(u_{ij} \geq u_{ij'}, \forall i', j' \in D) = \frac{\exp(V(p_{ij}, c_{ij}))}{\sum_{i', j' \in D} \exp(V(p_{ij'}, c_{ij'}))}, \tag{17.5}
\]

If we simplify the model by assuming the systematic utility of a proposed project is the rate of return on investment, that is, \( V(p_{ij}, c_{ij}) = (p_{ij} - c_{ij})/c_{ij} \), equation 17.5 reduces to random sampling process when \( p_{ij} \) and \( c_{ij} \) are known.

We use factors laid out in [Waddell, 2001b] in the revenue and costs calculation. The calculation of revenue or expected sale price of a proposal involves evaluating the price of the proposal with the real estate price model, and multiplying
it by the anticipated size of the new development. The costs are more complex, and involve the land cost, structure construction (hard) cost, site preparation, and policy-based costs such as impact fees (soft costs). In addition, in order to consider redevelopment, the developed parcels with relatively low improvement-to-land-value ratios are sampled and compared to greenfield development. In order to equalize these and make them comparable, the costs of acquiring any existing buildings on a parcel and demolishing them are included in the case of redevelopment.

<table>
<thead>
<tr>
<th>Table 17.2: Profitability Calculation and Filter for Real Estate Development and Redevelopment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revenue:</td>
</tr>
<tr>
<td>Expected sale price</td>
</tr>
<tr>
<td>Costs:</td>
</tr>
<tr>
<td>Land cost</td>
</tr>
<tr>
<td>Demolition cost</td>
</tr>
<tr>
<td>Soft costs</td>
</tr>
<tr>
<td>Filter:</td>
</tr>
<tr>
<td>Developable</td>
</tr>
<tr>
<td>Re-developable</td>
</tr>
</tbody>
</table>

Development Constraints

Constraints on development outcomes are included through a combination of user-specified spatial overlays and decision rules about specific types of development allowed in different situations. First, each parcel is assigned a series of overlays through spatial preprocessing using GIS overlay techniques. These overlays include features such as the following:

- Land use plan designation
- City
- County
- Wetland designation
- Floodplain/floodway
- Stream or riparian buffer
- High slope areas
- Urban Growth Boundary

These overlays can be used to assign user-specified constraints on the type of development that is allowed to occur within each of these overlay designations. These constraints are interpreted as ‘binding’ constraints, and not subject to market pressure. Currently, if users wish to examine the impact of these constraints, they would need to ‘relax’ a particular constraint within one scenario and compare the scenario results to a more restrictive policy.
Development Templates

It has been shown in the literature that home builders and developers are employing a portfolio of “standard housing type” ex ante in the development process and series of development pattern can be classified ex post through existing built environment components [Song and Knaap, 2007].

We adapt a procedure similar to [Song and Knaap, 2007] to classify the historical development pattern into development templates. Cluster analysis can be used to identify the development pattern for each major property type. Table 17.2.8 shows the residential development templates in the Puget Sound from 1990 to 2005.
<table>
<thead>
<tr>
<th>ID</th>
<th>Template Type</th>
<th>Characteristics</th>
<th>Property type</th>
<th>Lot size range (min)</th>
<th>Lot size range (max)</th>
<th>Units per acre</th>
<th>Sample size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Subdivision</td>
<td>10- to 20-acre lots</td>
<td>SFH</td>
<td>37.00</td>
<td>150.00</td>
<td>0.08</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Subdivision</td>
<td>3.3- to 10-acre lots</td>
<td>SFH</td>
<td>20.00</td>
<td>58.00</td>
<td>0.20</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>Subdivision</td>
<td>1.3- to 3.3-acre lots</td>
<td>SFH</td>
<td>12.00</td>
<td>35.00</td>
<td>0.46</td>
<td>86</td>
</tr>
<tr>
<td>4</td>
<td>Subdivision</td>
<td>0.67- to 1.3-acre lots</td>
<td>SFH</td>
<td>6.50</td>
<td>23.00</td>
<td>1.12</td>
<td>261</td>
</tr>
<tr>
<td>5</td>
<td>Subdivision</td>
<td>0.40- to 0.67-acre lots</td>
<td>SFH</td>
<td>4.00</td>
<td>14.00</td>
<td>2.05</td>
<td>325</td>
</tr>
<tr>
<td>6</td>
<td>Subdivision</td>
<td>0.25- to 0.40-acre lots</td>
<td>SFH</td>
<td>3.00</td>
<td>12.00</td>
<td>3.13</td>
<td>436</td>
</tr>
<tr>
<td>7</td>
<td>Subdivision</td>
<td>0.20- to 0.25-acre lots</td>
<td>SFH</td>
<td>2.50</td>
<td>10.00</td>
<td>4.45</td>
<td>715</td>
</tr>
<tr>
<td>8</td>
<td>Subdivision</td>
<td>0.15- to 0.20-acre lots</td>
<td>SFH</td>
<td>2.00</td>
<td>8.00</td>
<td>5.96</td>
<td>964</td>
</tr>
<tr>
<td>9</td>
<td>Subdivision</td>
<td>0.10- to 0.15-acre lots</td>
<td>SFH</td>
<td>2.00</td>
<td>6.50</td>
<td>8.41</td>
<td>585</td>
</tr>
<tr>
<td>10</td>
<td>Subdivision</td>
<td>0.050- to -0.10-acre lots</td>
<td>SFH</td>
<td>1.00</td>
<td>3.50</td>
<td>13.42</td>
<td>185</td>
</tr>
<tr>
<td>11</td>
<td>Single property</td>
<td>0.050- to 0.12-acre lots</td>
<td>SFH</td>
<td>0.05</td>
<td>0.12</td>
<td>10.61</td>
<td>11115</td>
</tr>
<tr>
<td>12</td>
<td>Single property</td>
<td>0.12- to 0.20-acre lots</td>
<td>SFH</td>
<td>0.12</td>
<td>0.20</td>
<td>6.20</td>
<td>21359</td>
</tr>
<tr>
<td>13</td>
<td>Single property</td>
<td>0.20- to 0.28-acre lots</td>
<td>SFH</td>
<td>0.20</td>
<td>0.28</td>
<td>4.37</td>
<td>16329</td>
</tr>
<tr>
<td>14</td>
<td>Single property</td>
<td>0.28- to 0.40-acre lots</td>
<td>SFH</td>
<td>0.28</td>
<td>0.40</td>
<td>3.07</td>
<td>11216</td>
</tr>
<tr>
<td>15</td>
<td>Single property</td>
<td>0.40- to 0.60-acre lots</td>
<td>SFH</td>
<td>0.40</td>
<td>0.60</td>
<td>2.09</td>
<td>8194</td>
</tr>
<tr>
<td>16</td>
<td>Single property</td>
<td>0.60- to 0.90-acre lots</td>
<td>SFH</td>
<td>0.60</td>
<td>0.90</td>
<td>1.37</td>
<td>5466</td>
</tr>
<tr>
<td>17</td>
<td>Single property</td>
<td>0.90- to 1.3-acre lots</td>
<td>SFH</td>
<td>0.90</td>
<td>1.30</td>
<td>0.92</td>
<td>8009</td>
</tr>
<tr>
<td>18</td>
<td>Single property</td>
<td>1.3- to 1.8-acre lots</td>
<td>SFH</td>
<td>1.30</td>
<td>1.80</td>
<td>0.64</td>
<td>3407</td>
</tr>
<tr>
<td>19</td>
<td>Single property</td>
<td>1.8- to 2.8-acre lots</td>
<td>SFH</td>
<td>1.80</td>
<td>2.80</td>
<td>0.42</td>
<td>6316</td>
</tr>
<tr>
<td>20</td>
<td>Single property</td>
<td>2.8- to 4.0-acre lots</td>
<td>SFH</td>
<td>2.80</td>
<td>4.00</td>
<td>0.30</td>
<td>1917</td>
</tr>
<tr>
<td>21</td>
<td>Single property</td>
<td>4.0- to 5.8-acre lots</td>
<td>SFH</td>
<td>4.00</td>
<td>5.80</td>
<td>0.20</td>
<td>7653</td>
</tr>
<tr>
<td>22</td>
<td>Single property</td>
<td>5.8- to 8.0-acre lots</td>
<td>SFH</td>
<td>5.80</td>
<td>8.00</td>
<td>0.15</td>
<td>1310</td>
</tr>
<tr>
<td>23</td>
<td>Single property</td>
<td>8.0- to 12-acre lots</td>
<td>SFH</td>
<td>8.00</td>
<td>12.00</td>
<td>0.10</td>
<td>1231</td>
</tr>
<tr>
<td>24</td>
<td>Single property</td>
<td>12- to 18-acre lots</td>
<td>SFH</td>
<td>12.00</td>
<td>18.00</td>
<td>0.07</td>
<td>294</td>
</tr>
<tr>
<td>25</td>
<td>Single property</td>
<td>18- to 50-acre lots</td>
<td>SFH</td>
<td>18.00</td>
<td>50.00</td>
<td>0.04</td>
<td>439</td>
</tr>
<tr>
<td>26</td>
<td>Single property</td>
<td>townhouse extra large lots</td>
<td>Condo</td>
<td>0.67</td>
<td>61.18</td>
<td>2.38</td>
<td>35</td>
</tr>
<tr>
<td>27</td>
<td>Single property</td>
<td>townhouse large lots</td>
<td>Condo</td>
<td>0.38</td>
<td>23.89</td>
<td>4.35</td>
<td>143</td>
</tr>
<tr>
<td>28</td>
<td>Single property</td>
<td>townhouse medium lots</td>
<td>Condo</td>
<td>0.26</td>
<td>21.98</td>
<td>6.60</td>
<td>151</td>
</tr>
<tr>
<td>29</td>
<td>Single property</td>
<td>townhouse small lots</td>
<td>Condo</td>
<td>0.19</td>
<td>26.64</td>
<td>9.34</td>
<td>173</td>
</tr>
<tr>
<td>30</td>
<td>Single property</td>
<td>low-rise large units</td>
<td>Condo</td>
<td>0.14</td>
<td>20.61</td>
<td>12.79</td>
<td>181</td>
</tr>
<tr>
<td>31</td>
<td>Single property</td>
<td>low-rise medium units</td>
<td>Condo</td>
<td>0.11</td>
<td>13.64</td>
<td>17.06</td>
<td>161</td>
</tr>
<tr>
<td>32</td>
<td>Single property</td>
<td>low-rise small units</td>
<td>Condo</td>
<td>0.09</td>
<td>11.01</td>
<td>22.02</td>
<td>110</td>
</tr>
<tr>
<td>33</td>
<td>Single property</td>
<td>med-rise large units</td>
<td>Condo</td>
<td>0.07</td>
<td>3.58</td>
<td>27.95</td>
<td>103</td>
</tr>
<tr>
<td>34</td>
<td>Single property</td>
<td>med-rise medium units</td>
<td>Condo</td>
<td>0.06</td>
<td>4.55</td>
<td>37.68</td>
<td>104</td>
</tr>
<tr>
<td>ID</td>
<td>Template Type</td>
<td>Characteristics</td>
<td>Property type</td>
<td>Lot size range (min)</td>
<td>Lot size range (max)</td>
<td>Units per acre</td>
<td>Sample size</td>
</tr>
<tr>
<td>----</td>
<td>-------------------------------------</td>
<td>--------------------------</td>
<td>---------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>69</td>
<td>Single property</td>
<td>med-rise small units</td>
<td>Condo</td>
<td>0.06</td>
<td>1.22</td>
<td>54.51</td>
<td>108</td>
</tr>
<tr>
<td>71</td>
<td>Single property</td>
<td>high-rise large units</td>
<td>Condo</td>
<td>0.11</td>
<td>2.72</td>
<td>102.20</td>
<td>43</td>
</tr>
<tr>
<td>73</td>
<td>Single property</td>
<td>high-rise small units</td>
<td>Condo</td>
<td>0.25</td>
<td>0.72</td>
<td>206.30</td>
<td>8</td>
</tr>
<tr>
<td>75</td>
<td>Single property</td>
<td>duplex/triplex extra large lots</td>
<td>MFA</td>
<td>3.22</td>
<td>8.05</td>
<td>0.44</td>
<td>37</td>
</tr>
<tr>
<td>77</td>
<td>Single property</td>
<td>duplex/triplex large lots</td>
<td>MFA</td>
<td>1.49</td>
<td>3.14</td>
<td>0.98</td>
<td>60</td>
</tr>
<tr>
<td>79</td>
<td>Single property</td>
<td>duplex/triplex medium lots</td>
<td>MFA</td>
<td>0.79</td>
<td>2.16</td>
<td>2.02</td>
<td>154</td>
</tr>
<tr>
<td>81</td>
<td>Single property</td>
<td>duplex/triplex &amp; low-rise</td>
<td>MFA</td>
<td>0.54</td>
<td>1.10</td>
<td>3.24</td>
<td>283</td>
</tr>
<tr>
<td>83</td>
<td>Single property</td>
<td>low-rise</td>
<td>MFA</td>
<td>0.40</td>
<td>2.14</td>
<td>4.39</td>
<td>415</td>
</tr>
<tr>
<td>85</td>
<td>Single property</td>
<td>low-rise</td>
<td>MFA</td>
<td>0.29</td>
<td>1.09</td>
<td>5.98</td>
<td>553</td>
</tr>
<tr>
<td>87</td>
<td>Single property</td>
<td>low-rise large units</td>
<td>MFA</td>
<td>0.22</td>
<td>0.99</td>
<td>8.11</td>
<td>350</td>
</tr>
<tr>
<td>89</td>
<td>Single property</td>
<td>low-rise medium units</td>
<td>MFA</td>
<td>0.16</td>
<td>0.80</td>
<td>10.90</td>
<td>361</td>
</tr>
<tr>
<td>91</td>
<td>Single property</td>
<td>low-rise small units</td>
<td>MFA</td>
<td>0.11</td>
<td>0.80</td>
<td>15.18</td>
<td>223</td>
</tr>
<tr>
<td>93</td>
<td>Single property</td>
<td>med-rise large units</td>
<td>MFA</td>
<td>0.07</td>
<td>1.01</td>
<td>24.82</td>
<td>147</td>
</tr>
<tr>
<td>95</td>
<td>Single property</td>
<td>med-rise small units</td>
<td>MFA</td>
<td>0.05</td>
<td>0.15</td>
<td>34.83</td>
<td>67</td>
</tr>
<tr>
<td>97</td>
<td>Single property</td>
<td>high-rise</td>
<td>MFA</td>
<td>0.03</td>
<td>0.15</td>
<td>59.05</td>
<td>46</td>
</tr>
</tbody>
</table>
17.2.9 The Role of Accessibility

Accessibility is a very important influence in urban space, and it similarly plays an important role in UrbanSim. Almost all models in UrbanSim consider the effects of accessibility. But unlike the monocentric or spatial interaction models, in which the choice of workplace is exogenous and residential locations are chosen principally on the basis of commute to the city center or to a predetermined workplace, we deal with accessibility in a more general framework. Accessibility is considered a normal good, like other positive attributes of housing, which consumers place a positive economic value on. We therefore expect that consumers value access to workplaces and shopping opportunities, among the many other attributes they consider in their housing preferences. However, not all households respond to accessibility in the same way. Retired persons would be less influenced by accessibility to job opportunities than would working age households, for instance.

We operationalize the concept of accessibility for a given location as the distribution of opportunities weighted by the travel impedance, or alternatively the utility of travel to those destinations. A number of alternative accessibility measures have been developed in UrbanSim. The utility of travel is measured as the composite utility across all modes of travel for each zone pair, obtained as the logsum of the mode choice for each origin-destination pair.

The accessibility model reads the logsum matrix from the travel model and the land use distribution for a given year, and creates accessibility indices for use in the household and business location choice models. The general framework is to summarize the accessibility from each zone to various activities for which accessibility is considered important in household or business location choice.

Since UrbanSim operates annually, but travel model updates are likely to be executed for two to three of the years within the forecasting horizon, travel utilities remain constant from one travel model run until they are replaced by the next travel model result. Although travel utilities remain constant, the activity distribution in these accessibility indices is updated annually, so that the accessibility indices change from one year to the next to reflect the evolving spatial distribution of activities.

17.3 Interface with Travel Model

UrbanSim takes several key inputs as exogenous, meaning that these are input assumptions that are not predicted directly by UrbanSim. Two of these are from external model systems: a macroeconomic model to predict future macroeconomic conditions such as population and employment by sector, and a travel demand model system to predict travel conditions such as congested times and composite utilities of travel between each interchange. The latter is loosely coupled to UrbanSim, with land use predictions input to the external travel models, and travel conditions input to subsequent annual iterations of the UrbanSim land use model system.

The travel models in widespread use in Metropolitan Planning Organizations are almost all traditional four-step travel models. The first model in the process is the trip generation model, and this uses zonal population and employment characteristics. When UrbanSim is connected to a travel model system, it generates a summary of the household and job data to a zone level, in order to create the summary input data needed by the travel model.

When the travel model completes the fourth step of traffic assignment to a transportation network, it can produce ‘skims’ from zone to zone that summarize key model predictions, such as:

- Travel time by mode by time of day by purpose
- Trips by mode by time of day by purpose
- Composite utility of travel using all modes by purpose
- Generalized costs (time + time equivalent of tolls) by purpose
- Logsums, or composite utilities\(^1\), from the mode choice model, by purpose and time of day

\(^1\)See section 2.4 for an explanation of this measure.
These skims can be combined with the spatial information in UrbanSim regarding the location of households and jobs, to produce a variety of accessibility measures, which in turn can influence UrbanSim models of residential location, workplace location, employment location, real estate prices, and real estate development. Figure 17.4 summarizes the interactions between UrbanSim and the travel model system.

In some applications, such as the San Francisco model, the travel model system is based on a more sophisticated activity-based framework, and UrbanSim is interfaced in a similar way. In the future, potential exists to more closely integrate UrbanSim with Activity-based models that are moving from research into practice.

17.3.1 User-Specified Events

Given our current understanding, no model will be able to simulate accurately the timing, location and nature of major events such as a major corporate relocation into or out of a metropolitan area, or a major development project such as a regional shopping mall. In addition, major policy events, such as a change in the land use plan or in an Urban Growth Boundary, are outside the range of predictions of our simulation. (At least in its current form, UrbanSim is intended as a tool to aid planning and civic deliberation, not as a tool to model the behavior of voters or governments. We want it to be used to say “if you adopt the following policy, here are the likely consequences,” but not to say “UrbanSim predicts that in 5 years the county will adopt the following policy.”)

However, planners and decision-makers often have information about precisely these kinds of major events, and there is a need to integrate such information into the use of the model system. It is useful, for example, to explore the potential effects of a planned corporate relocation by introducing user-specified events to reflect the construction of the corporate building, and the relocation into the region (and to the specific site) of a substantial number of jobs, and examine the cumulative or secondary effects of the relocation on further residential and employment location and real estate development choices. Inability to represent such events, in the presence of knowledge about developments that may be ‘in the pipeline,’ amounts to less than full use of the available information about the future, and could undermine the validity and credibility of the planning process. For these reasons, support for three kinds of events has been incorporated into the system: development events, employment events, and policy events.
Data for UrbanSim Applications

This chapter describes what UrbanSim needs in its baseyear database, ways in which the baseyear may be structured, how to create a set of scenario databases that share much of their data, and the use of output databases.

UrbanSim currently uses three types of databases, each of which is described in more details in following sections:

- **baseyear database** – defining the initial state of a simulation in a particular base year.
- **scenario database** – defining changes to a baseyear (or another scenario) database.
- **output database** – optional repository for simulation results.

At the moment, UrbanSim supports several database servers. The one that has been most thoroughly used and tested is the MySQL database server. Support has been added for Postgres, SQLite, and Microsoft SQL Server, though the Microsoft SQL Server interface has some differences from the other database platforms that limit it in some ways. We advise using MySQL or Postgres as DBMS options for production use with OPUS and UrbanSim.

Note that OPUS can now also read and write data in ESRI Geodatabases, which is quite valuable for data preparation and geoprocessing. At this point, however, the ESRI proprietary interface does not appear to have robust performance, and we recommend that large projects consider moving data into MySQL or Postgres for operational use. We are beginning to provide more extensive support for using PostGIS, a spatial extension to Postgres (and similar in some ways to the role of SDE for ESRI Geodatabases), as a means of obtaining rapid data access, editing, geoprocessing and visualization. PostGIS data can also be accessed from ESRI Geodatabases, making this a more universally viable option.

### 18.1 Data Requirements for OPUS and UrbanSim

It is important to recognize that the data needed for a model system are dependent on the models and their specifications. OPUS is a general software platform for implementing models. One could implement a wide array of models in OPUS, and their needs for data would be dictated by those models. For example, one could create an OPUS project with one model, that implements a simple gravity model for locating households, and uses only a zonal table with constraints, a zone-to-zone travel time table, and a set of control totals as inputs. In such a case, the data requirements would be only those tables required by the gravity model.

UrbanSim is an evolving set of models, some of which have been adapted to different data structures and geographic units of analysis, such as gridcells, parcels, buildings and zones. Each of these models, depending on how the user specifies the model, creates its own data requirements. Documenting a universal set of data requirements for all
UrbanSim users is therefore not possible. Over the years, examples of data used in an existing UrbanSim example project has been used by new users as a blueprint for developing their own databases using local data. But it became clear that the boundary between data that was essential and data that was optional was not at all clear to users.

Some data in a standard UrbanSim application database is generic, and some contains data used to store overall system information for an application. The *urbansim_constants* table is an example of the latter. This table, developed for use with the gridcell versions of UrbanSim, contains information such as the point of origin of the grid, its cell size, thresholds to be considered for spatial queries of what is to be considered 'within-walking-distance'. In the more recent parcel application of UrbanSim, this table is still retained, mainly because a grid can still be used with a parcel model system, by cross-referencing parcels and gridcells, and some variables still make use of the spatial queries. So in this case, the table is needed, even though it may not be used specifically by a model. Dependencies of this sort will gradually be eliminated from the system, as all of these kinds of configurations will be accommodated within the GUI.

In the sections that follow we attempt to organize a presentation of tables that are commonly used in UrbanSim applications, and to cluster or identify those tables that are more specific to one or another configuration of UrbanSim, such as a gridcell-based application, or one based on parcels.

### 18.2 Input Database Design: Basyear and Scenario Databases

UrbanSim gets its input data from either a *basyear* database or a *scenario* database. The UrbanSim simulator treats *basyear* and *scenario* databases as read-only databases, however other data preparation applications such as the estimators or the household synthesizer may write to them. In fact, when the run manager starts a new simulation, the first step is to copy the basyear data into the basyear cache. The simulation then reads all of its basyear information from the basyear cache and writes all results to the simulation cache. Data is only written to the output database when specified (currently done manually).

A *basyear* database contains a snapshot of the base information defining the initial state before the UrbanSim simulation. Most of the data typically is about a particular year, e.g., geographic information, initial household and job information, etc., for a given year.

A *scenario* database contains additional and augmenting information to alter the base year data when simulating a particular scenario e.g., new transportation links, an expanded urban growth boundary, etc. Any of the table may be placed in either of the database, although typically most are placed in the basyear database. The scenario databases typically only contains tables specifying different possible futures, e.g. tables of exogenous events scheduled for future years.

The way that the scenario can modify the information in the basyear database or another scenario database is determined by the scenario linking.

#### 18.2.1 Scenario Linking

The scenario databases are linked to each other and, eventually, to a basyear database via a tree structure: each scenario can refer to exactly one parent database; that parent can be either another scenario or a basyear database. The basyear database is the root of the tree. In this way, multiple scenarios may share the same basyear.

When UrbanSim looks for a particular input database table, it traverses this chain looking for that table. The first table matching that name is used. In this way, any tables contained in the scenario database “shadow” or “hide” the same-named tables in the scenario’s parent database(s).

Consider these example of how to create derivative scenarios:
- Scenario 1 is the base year plus a larger urban growth boundary, thus scenario 1’s parent is the base year database. The UrbanSim scenario file will specify the scenario 1 database as the “scenario-data”.

- Scenario 2 is the base year plus a major employer leaving the municipality. Scenario 2’s parent is also the base year database. The UrbanSim scenario file will specify the scenario 2 database as the “scenario-data”.

- Scenario 3 is the same as scenario 1, but with additional changes in the zoning laws to compensate for the larger UGB. Scenario 3’s parent is scenario 1 and the UrbanSim scenario file will specify the scenario 3 database as the “scenario-data”.

- Scenario 4 is the same as scenario 1, but with changes in the population demographics as a result of the larger UGB. Scenario 4’s parent is scenario 1 and the UrbanSim scenario file will specify the scenario 4 database as the “scenario-data”.

Any table in a scenario database “hides” the same-named table in the scenario’s parent database.

**18.2.2 Scenario Database Design**

The only required table in the scenario database is the `scenario_information` table (see Sec. 18.10.4). This table points to its parent, i.e. to a baseyear database, or to another scenario database.

In addition, the scenario database may include any other tables for data that is different in this scenario. For example, if the scenario is simulating a large retail development in the suburbs, the `development_events` table would be included in the scenario database. In this way, a scenario database may change any of the information contained in the baseyear.
18.3 Output Database

An output database may contain the results of an UrbanSim simulation. This database is optional; the urbansim cache is the primary storage location for simulation inputs and results.

18.4 General Database Design

18.4.1 Guidelines

Here are some guidelines on database design:

- Use only lower-case letters, digits, and underscores for the names of databases, database tables, and database columns. This avoids problems when moving databases between different operating systems (e.g. between Windows and Linux).

- Avoid overly abbreviated names. While very short names were required by some other systems, UrbanSim itself has no limit on the length of names. Most database systems allow database names, table names and column names to be 32 characters, 64 characters (e.g. MySQL), or more.

- Unique identifiers must be larger than 0.

- Avoid Null values in tables. The Python conversion tool cannot deal with Nulls, and thus, converting tables to a simulation cache would crash if Nulls are present.

18.4.2 Data Types

When Opus reads data from a database table, it stores the data in a Python type that is close to the type of the corresponding column in the database. The particular mapping between database types and Python types currently is defined for MySQL and should be re-defined for each additional type of database. The conversion for MySQL is:

<table>
<thead>
<tr>
<th>MySQLdb FIELD_TYPE</th>
<th>Python/numpy type</th>
</tr>
</thead>
<tbody>
<tr>
<td>tinyint(1)</td>
<td>bool8</td>
</tr>
<tr>
<td>short</td>
<td>int16</td>
</tr>
<tr>
<td>int24</td>
<td>int32</td>
</tr>
<tr>
<td>long</td>
<td>int32</td>
</tr>
<tr>
<td>longlong</td>
<td>int64</td>
</tr>
<tr>
<td>float</td>
<td>float32</td>
</tr>
<tr>
<td>double</td>
<td>float64</td>
</tr>
<tr>
<td>decimal</td>
<td>float64</td>
</tr>
</tbody>
</table>

Similarly, when writing from Python to a database, Opus converts from Python types to database-specific data types. The conversion for MySQL is:

<table>
<thead>
<tr>
<th>Python/numpy type</th>
<th>MySQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool8</td>
<td>int(1)</td>
</tr>
<tr>
<td>int8</td>
<td>int(8)</td>
</tr>
<tr>
<td>int16</td>
<td>int(16)</td>
</tr>
<tr>
<td>int32</td>
<td>int(32)</td>
</tr>
<tr>
<td>int64</td>
<td>int(64)</td>
</tr>
<tr>
<td>float32</td>
<td>float</td>
</tr>
<tr>
<td>float64</td>
<td>double</td>
</tr>
</tbody>
</table>
Note that these conversions are not symmetrical, since multiple database types map onto a single Python type. The result is that when written back to the database, the column types may change from that of the input database table.

In the versions of UrbanSim after 4.0, we have integrated a database interface library, SQLAlchemy, which standardizes the interfaces to multiple database platforms (MySQL, Postgres, SQLite, MS SQL, etc), providing a more consistent translation of data and queries to the platform-specific requirements.

18.5 What Tables are Used in UrbanSim?

Most database tables are optional. The required set of tables is determined by the set of models configured for a run. Details can be found in the description of the particular models in Section 25.4.

Additionally, some tables and various attributes of tables are determined by the variables used by the models. This can be found by looking at the models’ specification tables.

The tables_to_cache argument of urbansim.configs.cache_baseyear_configuration lists the tables required for the standard set of models in a production run of UrbanSim. Additional tables can be used in post-processing, for example for creating indicators.

18.6 Coefficients and Specification Tables

The UrbanSim models are configured through user specified variables and coefficients. The coefficients should be estimated separately for each region to be modeled by UrbanSim. The art and science of estimating the coefficients is a matter for a series of college courses so this description assumes that appropriate variables for each model have been chosen, and the appropriate coefficients have been estimated for those variables.

Each of the regression models and Logit models have two associated tables: a table to store the specification of what variables to use for that model, and a table to store the estimated coefficients to use for those variables. The names of these tables are composed by appending either _coefficients or _specification to the model name. The tables for the land price model, for instance, are land_price_model_coefficients and land_price_model_specification.

All coefficient tables share the same schema, as do all specification tables. The schemas are:

18.6.1 Specification table

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable_name</td>
<td>varchar</td>
<td>A legitimate specification for an Opus variable (see below).</td>
</tr>
<tr>
<td>coefficient_name</td>
<td>varchar</td>
<td>Name of a coefficient connected to this variable.</td>
</tr>
<tr>
<td>sub_model_id</td>
<td>integer</td>
<td>(optional) Defines the submodel, if the model contains submodels. If the model does not have multiple submodels, use “-2” for this field, or leave it out.</td>
</tr>
<tr>
<td>equation_id</td>
<td>integer</td>
<td>(optional) If a submodel has multiple equations, this field contains an id identifying which equation this row applies to. If a model does not have multiple equations, use “-2” for this field, or leave it out.</td>
</tr>
<tr>
<td>fixed_value</td>
<td>double</td>
<td>(optional) If a coefficient should have a fixed value for an estimation, it should be set in this column. All values that are not equal to 0 are considered as fixed values.</td>
</tr>
</tbody>
</table>

• Values of the sub_model_id column are determined by the submodel_string parameter of the model, see
e.g. initialization of ChoiceModel (24.4.3) or RegressionModel (24.4.4). Specifically, the values of the sub_model_id column must exist in the submodel_string attribute of the model’s dataset. The employment location choice models, for instance, define submodels by employment sectors, so the values of this field are the sector_id values of the jobs dataset.

- Each combination of (sub_model_id, coefficient_name) must exist in the model’s coefficients table.
- Each combination of (sub_model_id, equation_id, variable_name) must be unique.
- If fixed_value is non-zero in at least one row of the table, set the remaining values to 0. Currently, fixed_value are considered only in the estimation of ChoiceModel.

A legitimate specification for an Opus variable may be one of the following:

- The word constant indicating a value specific to this combination of (sub_model_id, equation_id).
- The name of a primary attribute of a dataset, specified as a period-separated tuple of dataset name, attribute name, e.g. gridcell.percent_slope.
- The name of a dataset attribute, specified as a period-separated triple of Opus package name, dataset name, attribute name, e.g. urbansim.gridcell.population.
- Any Opus expression as described in 13.
- Any word preceeded by ‘__’ will be considered as a special parameter without a relation to an Opus variable. It can be used for estimating additional parameters.

### 18.6.2 Coefficient table

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coefficient_name</td>
<td>varchar</td>
<td>Unique name of a coefficient</td>
</tr>
<tr>
<td>estimate</td>
<td>double</td>
<td>The estimated value of this coefficient</td>
</tr>
<tr>
<td>sub_model_id</td>
<td>integer</td>
<td>(optional) Identifier for a submodel, or -2 if not used.</td>
</tr>
<tr>
<td>standard_error</td>
<td>double</td>
<td>(optional) The standard error of this estimated value. This is for reference only and is not used by UrbanSim.</td>
</tr>
<tr>
<td>t_statistic</td>
<td>double</td>
<td>(optional) The t-statistic of this coefficient for the test of significance from 0. This is for reference only and is not used by UrbanSim.</td>
</tr>
<tr>
<td>p_value</td>
<td>double</td>
<td>(optional) The p-value of this t-statistic, gives the Prob(</td>
</tr>
</tbody>
</table>

- Each combination of (sub_model_id, coefficient_name) must be unique and must exist in the model’s specification table.

### 18.7 Database Tables about Employment

This section contains table descriptions that relate to employment. They should be general whether using a gridcell or a parcel or a zone-based application, with the exception of what the location-id is that the jobs table links to.
18.7.1 The annual_employment_control_totals table

This table gives total target quantities of employment, by sector, by home-based, and by year for each simulated year. It is used by the Employment Transition Model.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sector_id</td>
<td>integer</td>
<td>Index into the employment_sectors table</td>
</tr>
<tr>
<td>year</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>total_home_based_employment</td>
<td>integer</td>
<td>Target home based employment for this sector and year</td>
</tr>
<tr>
<td>total_non_home_based_employment</td>
<td>integer</td>
<td>Target non-home based employment for this sector and year</td>
</tr>
</tbody>
</table>

- sector_id must be a valid sector_id from the employment_sectors table
- total_home_based_employment and total_non_home_based_employment must be greater than or equal to zero
- A control total must be provided for each sector in employment_sectors for every year in the scenario.

18.7.2 The annual_relocation_rates_for_jobs table

This table is only used by the Employment Relocation Model.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sector_id</td>
<td>integer</td>
<td>Index into the employment_sectors table</td>
</tr>
<tr>
<td>job_relocation_probability</td>
<td>float</td>
<td>Probability that a job in this sector will relocate within the time span of one year</td>
</tr>
</tbody>
</table>

- There must be a single entry for every employment sector in the employment_sectors table.
- job_relocation_probability must be between 0 and 1, inclusive.

18.7.3 The employment_sectors table

An EmploymentSector is a logical category of employment, such as “automobile_sales” or “shipping”. Each row defines one EmploymentSector.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sector_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Unique name of the Sector</td>
</tr>
</tbody>
</table>

- sector_id must be unique and greater than zero.
- name must be unique. We recommend that names follow the style guide.

18.7.4 The employment_adhoc_sector_groups table

Each row defines one EmploymentAdHocSectorGroup, but not the group’s membership - the memberships are defined in the employment_adhoc_sector_group_definitions table.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Unique name of the Group</td>
</tr>
</tbody>
</table>

18.7. Database Tables about Employment
• group_id must be unique and greater than zero
• name must be unique. The required employment ad hoc sector groups must be lower case with underscores between words, e.g. lower_case_with_underscores_between_words. We recommend that all names follow this style.

For example, if we had an ad-hoc group “retail” that included sectors id=56 “automobile_sales”, id=29 “department_store_sales”, and id=38 “wireless_phone_sales”, and another group “transportation” that included sectors “automobile_sales” and id=6 “trucking”, these tables would be:

<table>
<thead>
<tr>
<th>employment_adhoc_sector_groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_id</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>employment_adhoc_sector_group_definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sector_id</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Here are example of groups used in specification of various models:

• Employment Location Choice Model
  – basic
  – retail
  – service

• Employment Non-Home-Based Location Choice Model
  – basic
  – retail
  – service
  – elc_sector

• Scaling Procedure for Jobs Model
  – scalable_sectors

• Household Location Choice Model
  – retail

18.7.5 The employment_adhoc_sector_group_definitions table

This table defines the set of employment_sectors in each EmploymentSectorAdHocGroup. Each row defines one “belongs to” relationship (a particular EmploymentSector “belongs to” a particular EmploymentSectorAdHocGroup).

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sector_id</td>
<td>integer</td>
<td>Index into the employment_sectors table</td>
</tr>
<tr>
<td>group_id</td>
<td>integer</td>
<td>Index into the employment_adhoc_sector_groups table</td>
</tr>
</tbody>
</table>
• sector_id must be a valid index into the employment_sectors table
• group_id must be a valid index into the employment_adhoc_sector_groups table
• The combination of sector_id+group_id must be unique

18.7.6 The jobs table used in Gridcell-based Applications

One row per job in the region.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>grid_id</td>
<td>integer</td>
<td>Grid cell this job exists in; zero if currently not assigned to a grid cell</td>
</tr>
<tr>
<td>home_based</td>
<td>boolean</td>
<td>True if home-based</td>
</tr>
<tr>
<td>sector_id</td>
<td>integer</td>
<td>Sector this job belongs to</td>
</tr>
<tr>
<td>building_type</td>
<td>integer</td>
<td>building type code</td>
</tr>
</tbody>
</table>

• grid_id must be a valid id in the gridcells table
• job_id must be unique and greater than zero
• sector_id must be a valid id in the employment_sectors table
• Building type code must be a valid id in the job_building_types table.

18.7.7 The jobs table used in Parcel-based Applications

One row per job in the region.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>building_id</td>
<td>integer</td>
<td>Building this job exists in; zero if currently not assigned to a building</td>
</tr>
<tr>
<td>sector_id</td>
<td>integer</td>
<td>Sector this job belongs to</td>
</tr>
<tr>
<td>building_type</td>
<td>integer</td>
<td>building type code</td>
</tr>
<tr>
<td>sqft</td>
<td>integer</td>
<td>Square Feet used by job</td>
</tr>
</tbody>
</table>

• building_id must be a valid id in the building table
• job_id must be unique and greater than zero
• sector_id must be a valid id in the employment_sectors table
• Building type code must be a valid id in the job_building_types table.

18.7.8 The job_building_types table

A table of building types for jobs. It is used to determine the members of the Employment Location Choice Model group, and is used by the Employment Transition Model.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>name</td>
<td>string</td>
<td>Name of type, e.g. “commercial”</td>
</tr>
<tr>
<td>home_based</td>
<td>boolean</td>
<td>True if home-based</td>
</tr>
</tbody>
</table>
• id must be unique and greater than zero
• name must be unique
• home_based is either 0 (zero) or 1 (one)

Example:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>home_based</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>commercial</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>governmental</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>industrial</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>home_based</td>
<td>1</td>
</tr>
</tbody>
</table>

18.8 Database Tables about Households

The following tables should be general whether using a gridcell or a parcel or a zone-based application, with the exception of what the location-id is that the households table links to.

18.8.1 The annual_household_control_totals table

This table is used by the Household Tranition Model. It gives target quantities of households classified by year and an optional set of other user-defined attributes, such as race of head, or size of household. Each attribute is a column. The table’s key is a combination of all attributes other than total_number_of_households. The table must contain a row for each attribute and each simulated year.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>integer</td>
<td>Year for the total</td>
</tr>
<tr>
<td>age_of_head</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of age of head of household</td>
</tr>
<tr>
<td>cars</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of number of cars in household</td>
</tr>
<tr>
<td>children</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of number of children in household</td>
</tr>
<tr>
<td>income</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of household income</td>
</tr>
<tr>
<td>persons</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of size of household in number of people</td>
</tr>
<tr>
<td>race_id</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of race of head of household</td>
</tr>
<tr>
<td>workers</td>
<td>integer</td>
<td>(optional) Household characteristic bin number of employed people in household</td>
</tr>
<tr>
<td>total_number_of_households</td>
<td>integer</td>
<td>Target number of households of this household type and year</td>
</tr>
</tbody>
</table>

• The optional attributes above are households attributes. Thus, the names must match attribute names in the households table (18.8.3). Any other attributes are allowed if they are found in the households table.

• The bins for each attribute are defined in the table household_characteristics_for_ht (18.8.4). The bin number is an index of such bin, starting at 0. If there is no bin definition in household_characteristics_for_ht for an attribute, the following default bins are assumed: [0, 1), [1, 2), [2, 3), ....

• total_number_of_households must be greater than or equal to zero.
• For each year in the scenario, the entries should be complete. This means that there is a row for that year for the cross product of all specified bins.
• If a year is not complete, the household types that are not specified will not be modified.

As an example, the following table is valid for a simulation of year 2005, with two races (id=1 and id=2) and households with number of persons = 1, 2 and 3.

<table>
<thead>
<tr>
<th>year</th>
<th>race_id</th>
<th>persons</th>
<th>total_number_of_households</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>1</td>
<td>1</td>
<td>2500</td>
</tr>
<tr>
<td>2005</td>
<td>1</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>2005</td>
<td>1</td>
<td>3</td>
<td>8000</td>
</tr>
<tr>
<td>2005</td>
<td>2</td>
<td>1</td>
<td>1200</td>
</tr>
<tr>
<td>2005</td>
<td>2</td>
<td>2</td>
<td>1300</td>
</tr>
<tr>
<td>2005</td>
<td>2</td>
<td>3</td>
<td>2500</td>
</tr>
</tbody>
</table>

18.8.2 The `annual_relocation_rates_for_households` table

The annual relocation rates for households, by combination of age and income of household. These values are the probabilities that a household with the given characteristics will relocate within the time span of one year. They do not alter from year to year. This table is only used by the Household Relocation Model.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>age_min</td>
<td>integer</td>
<td>The minimum age for which this probability is valid.</td>
</tr>
<tr>
<td>age_max</td>
<td>integer</td>
<td>The maximum age for which this probability is valid, -1 means no maximum.</td>
</tr>
<tr>
<td>income_min</td>
<td>integer</td>
<td>The minimum income for which this probability is valid.</td>
</tr>
<tr>
<td>income_max</td>
<td>integer</td>
<td>The maximum income for which this probability is valid, -1 means no maximum.</td>
</tr>
<tr>
<td>probability_of_relocating</td>
<td>float</td>
<td>The probability of relocating in a year.</td>
</tr>
</tbody>
</table>

• age_min must be >= 0
• age_max must be > age_min or else -1
• income_min must be >= 0 and must be a multiple of 10.
• income_max must be > income_min and a multiple of 10 -1 (e.g. 200,999) or else -1
• probability_of_relocating must be >= 0.0 and <= 1.0
• The ranges must be disjoint and cover the entire space (from zero to infinity in the two-dimensional space produced by age and income).

As an example, this table:

<table>
<thead>
<tr>
<th>age_min</th>
<th>age_max</th>
<th>income_min</th>
<th>income_max</th>
<th>probability_of_relocating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4999</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>0</td>
<td>3999</td>
<td>0.6</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>5000</td>
<td>-1</td>
<td>0.7</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>4000</td>
<td>4999</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Would produce a space like this:
18.8.3 The households table

One row per household in the region. All people in the region belong to exactly one household.

Note that the table below, which is from a gridcell-based application, also works for a parcel-based application with only one exception: the gridcell identifier column should be replaced by a building id. Also, the household synthesizer that has been recently contributed by ASU is being integrated into the GUI, and will provide more flexibility in deciding on the specific household attributes to be generated in this table.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>household_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>grid_id</td>
<td>integer</td>
<td>Grid cell this household resides in; zero if currently not residing in a housing unit</td>
</tr>
<tr>
<td>persons</td>
<td>integer</td>
<td>Total number of people living in this household.</td>
</tr>
<tr>
<td>workers</td>
<td>integer</td>
<td>Total number of workers living in this household.</td>
</tr>
<tr>
<td>age_of_head</td>
<td>integer</td>
<td>Age of head of the household</td>
</tr>
<tr>
<td>income</td>
<td>integer</td>
<td>Income of this household</td>
</tr>
<tr>
<td>children</td>
<td>integer</td>
<td>Number of children living in this household</td>
</tr>
<tr>
<td>race_id</td>
<td>integer</td>
<td>Race of head of household</td>
</tr>
<tr>
<td>cars</td>
<td>integer</td>
<td>Number of cars in this household</td>
</tr>
</tbody>
</table>

- household_id must be unique and greater than zero
- grid_id must be a valid id in the gridcells table or zero
- persons must be greater than zero
- workers must be between zero and persons
- age_of_head greater than zero
- income must be greater than zero and less than or equal to absolute_max_income which can be defined in urbansim_constants table (default is 2,000,000,000).
• children must be greater than or equal to zero and less than or equal to persons
• race_id must be a valid id in the race_names table
• cars must be greater than or equal to zero
• The total number of households in a single grid cell should be no greater than that cell’s residential units.

households_for_estimation

The schema and structure of this table is identical to the basic households table. It contains data on actual households, and their actual placements in gridcells, buildings or zones (depending on the type of the particular application) typically from survey data. It is used in the household location model estimation process for determining model coefficients.

18.8.4 The household_characteristics_for_ht table

Bin definitions for the characterizing households used by the Household Transition Model (25.4.12) to produce an N-dimensional partitioning of the households. Thus, the table is only needed if the Household Transition Model is enabled.

The names of the characteristics must match attribute names in the households table (18.8.3). If a characteristic is used in the table annual_household_control_totals (18.8.1), the names in both tables must also match. For example the table can contain the following characteristics:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Characteristic Definition</th>
<th>Bin Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>age_of_head</td>
<td>Age, in years, of head of household</td>
<td>user-configurable</td>
</tr>
<tr>
<td>cars</td>
<td>Number of cars in household</td>
<td>user-configurable</td>
</tr>
<tr>
<td>children</td>
<td>Number of children in the household</td>
<td>user-configurable</td>
</tr>
<tr>
<td>income</td>
<td>Household income</td>
<td>user-configurable</td>
</tr>
<tr>
<td>persons</td>
<td>Number people in household</td>
<td>user-configurable</td>
</tr>
<tr>
<td>race_id</td>
<td>race_id of head of household</td>
<td>user-configurable</td>
</tr>
<tr>
<td>workers</td>
<td>Number of employed people in household</td>
<td>user-configurable</td>
</tr>
</tbody>
</table>

The table has the following structure:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>characteristic</td>
<td>varchar</td>
<td>See above for examples</td>
</tr>
<tr>
<td>min</td>
<td>integer</td>
<td>Minimum value for this bin for this characteristic. Values are placed in a bin iff min &lt;= value &lt;= max</td>
</tr>
<tr>
<td>max</td>
<td>integer</td>
<td>Maximum value for this bin for this characteristic; -1 means infinity / no maximum</td>
</tr>
</tbody>
</table>

• min must be greater than or equal to zero.
• max must be greater or equal than min or else -1.
• Bins for each characteristic may not overlap.
• Bins for each characteristic should cover all values contained in the data.

As an example, this table:
defines these bins:

- income: 0..4,999, 5,000..14,999, 15,000..+infinity
- age_of_head: 0..+infinity
- children: 0..+infinity
- workers: 0..+infinity
- cars: 0..2, 3..+infinity

The index of these bins within a characteristic is used as a bin number in `annual_household_control_totals`. This index starts at 0. Thus, using the bins above, a household with two children, a 27 year old head, an income of $6,345, one worker and no cars would be characterized into these bin numbers:

- Income: 1
- age_of_head: 0
- children: 1
- workers: 1
- cars: 0

### 18.8.5 The `race_names` table

This table is only needed if you include race related variables to model specifications. It has one row per race.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>race_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Name of the race</td>
</tr>
<tr>
<td>minority</td>
<td>boolean</td>
<td>True if the race is a minority</td>
</tr>
</tbody>
</table>

- race_id must be unique and greater than zero
- There must be at least one non-minority group listed

### 18.9 Database Tables about Transportation Analysis Zones

#### 18.9.1 The `zones` table

Traffic analysis zones are geographic regions. In UrbanSim, these zones are rasterized by the grid cells (the zones are distorted to fit to cell boundaries and thus will have rough or stair-stepped edges).
In practice, the zones table often includes other columns, depending upon the needs for your models. These data should be updated with the results of any travel model run with whatever attributes are needed.

### Column Name | Data Type | Description
--- | --- | ---
zone_id | integer | Unique identifier
travel_time_to_airport | integer | (optional) Units: Minutes
travel_time_to_cbd | integer | (optional) Units: Minutes
faz_id | integer | (optional) Foreign key of the FAZ (forecast analysis zone) containing this zone.

- zone_id must be unique and greater than zero
- travel_time_to_airport and travel_time_to_cbd must be >= 0. The attributes are required if there are variables in any model specification that access these attributes.

#### 18.9.2 The travel_data table

The travel data can be interpreted as the composite utility of going from one place to another given the available travel modes for that household type. (Negative values reflect the fact that the time required gives the trip negative utility.)

Intrazonal travel may have less utility than interzonal travel if mass transit routes or highway options allow for easier travel to an adjacent zone than within a zone. logsum3 often shows lower utility than logsum2 because the logsums represent composite utilities for different household types. So, for example, it may be that 2 car households tend to have a more favorable person-car ratio than 3+ car households. Or it may be that 2 car households are more frequently able to combine trips, decreasing the disutility of any individual trip.

These data should be updated with the results of any travel model run.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_zone_id</td>
<td>integer</td>
<td>“From” traffic analysis zone</td>
</tr>
<tr>
<td>to_zone_id</td>
<td>integer</td>
<td>“To” traffic analysis zone</td>
</tr>
<tr>
<td>logsum0</td>
<td>float</td>
<td>(optional) Logsum value for 0 vehicle households, transit logsum</td>
</tr>
<tr>
<td>logsum1</td>
<td>float</td>
<td>(optional) Logsum value for 1 vehicle households, transit logsum</td>
</tr>
<tr>
<td>...</td>
<td>float</td>
<td>...</td>
</tr>
<tr>
<td>logsumN</td>
<td>float</td>
<td>(optional) Logsum value for N+ vehicle households, transit logsum</td>
</tr>
</tbody>
</table>

- There must be a row for each combination of from_zone_id and to_zone_id for all zones in the zones table. For instance, if the table zones contains 3 zones (1, 2, and 5) there must be at least the following 9 entries in travel_data: (1,1), (1,2), (1,5), (2,1), (2,2), (2,5), (5,1), (5,2), (5,5).
- All logsum* values must be less than or equal to zero. If you have positive logsum values, subtract the maximum logsum value from all logsums in your table. This will correctly shift the logsums so that no are greater than zero.
- Other attributes can be included, depending on the travel model used and model specifications.

#### 18.10 Other Database Tables

##### 18.10.1 The base_year table

This table is optional. It is only used if the base year is not defined in the configuration. It has one row only.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>integer</td>
<td>Year of base data</td>
</tr>
</tbody>
</table>

18.10.2 The **cities** table

The table is only needed if you want to create indicators on city level.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>city_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>city_name</td>
<td>varchar</td>
<td></td>
</tr>
</tbody>
</table>

- city_id must be unique and greater than zero.

18.10.3 The **counties** table

The table is only needed if you want to create indicators on county level.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>county_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>county_name</td>
<td>varchar</td>
<td></td>
</tr>
</tbody>
</table>

- county_id must be unique and greater than zero.

18.10.4 The **scenario_information** table

Description of the scenario. It has one row only.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>varchar</td>
<td>(optional) Human readable description</td>
</tr>
<tr>
<td>parent_database_url</td>
<td>varchar</td>
<td>The name of the next database in the chain of scenario databases.</td>
</tr>
</tbody>
</table>

- parent_database_url must be empty in the baseyear database.

18.11 UrbanSim Constants

18.11.1 The **urbansim_constants** table

Constants needed for calculations made by the various models. It has a single row with one column per constant.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell_size</td>
<td>float</td>
<td>Width and height of each grid cell in units</td>
</tr>
<tr>
<td>units</td>
<td>varchar</td>
<td>Units of measurement, e.g. “meters” or “feet”</td>
</tr>
<tr>
<td>walking_distance_circle_radius</td>
<td>float</td>
<td>Walking distance in meters, e.g., 600 m</td>
</tr>
<tr>
<td>young_age</td>
<td>integer</td>
<td>Max age for a person to be considered young</td>
</tr>
<tr>
<td>property_value_to_annual_cost_ratio</td>
<td>float</td>
<td>Ratio of the total property value to an annual rent for that property</td>
</tr>
<tr>
<td>low_income_fraction</td>
<td>float</td>
<td>Fraction of the total number of households considered to have low incomes, e.g., 0.1</td>
</tr>
<tr>
<td>mid_income_fraction</td>
<td>float</td>
<td>Fraction of the total number of households considered to have mid-level incomes, e.g., 0.5</td>
</tr>
<tr>
<td>near_arterial_threshold</td>
<td>float</td>
<td>Line distance from the centroid of a cell to an arterial for it to be considered nearby, e.g., 300</td>
</tr>
<tr>
<td>near_highway_threshold</td>
<td>float</td>
<td>Line distance from the centroid to a highway for it to be considered nearby, e.g., 300</td>
</tr>
<tr>
<td>percent_coverage_threshold</td>
<td>integer</td>
<td>The threshold above which a grid cell’s percent_*, e.g. percent_wetland, must be considered “covered” for that attribute. So, if percent_coverage_threshold is 50 percent and percent_wetland is 60 percent, the grid cell would be considered “covered” by wetland.</td>
</tr>
<tr>
<td>recent_years</td>
<td>integer</td>
<td>Maximum number of years to look back when considering recent transitions. For example, if recent_years = 3, then the value commercial_sqft_recently_added in the gridcells table would refer to the number of square feet of commercial space built in the last 3 years.</td>
</tr>
</tbody>
</table>

- cell_size must be greater than zero
- units must be one of the following: meters, feet, miles, kilometers.
- walking_distance_circle_radius must be greater than zero
- young_age must be greater than zero
- property_value_to_annual_cost_ratio must be greater than zero
- low_income_fraction must be between 0 and 1
- mid_income_fraction must be between 0 and 1
- mid_income_fraction + low_income_fraction must be at most 1.
- near_arterial_threshold must be greater than zero
- near_highway_threshold must be greater than zero
- percent_coverage_threshold must be between zero and 100. Note that this value is exclusive; for example, if the value is set to 45 and a grid cell is 45% covered by roads, the cell will not be considered to be “covered” by roads.
- recent_years must be > 0
- Other constants can be included.
- Which constants are required and which are not depends on the selection of models and model specifications.
Data for Gridcell-based Applications

19.1 Database Tables about Grid Cells

19.1.1 The gridcells table

Geographic information partitioned into a rectangular grid of rectangular cells.

The “improvement_value” fields, below, indicate the value (e.g., dollars) of all buildings of a particular type that are in this grid cell. For instance, commercial_improvement_value is the total value of all commercial buildings in this grid cell. The use of “improvement” indicates that buildings are considered “improvements” over the grid cell’s land value.

Attributes that are marked as optional are only required by specific variables. Thus, if they are needed or not depends on model specifications. Attributes that are NOT marked as optional are used by various models. In some situations, (e.g. by skipping specific models) some of those attributes may not be required.
<table>
<thead>
<tr>
<th><strong>Column Name</strong></th>
<th><strong>Data Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>grid_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>commercial_sqft</td>
<td>integer</td>
<td>The sum of the square footage of buildings that are classified as commercial (generally including retail and office land uses). This is not a measure of land area.</td>
</tr>
<tr>
<td>development_type_id</td>
<td>integer</td>
<td>Index into the Development Types table</td>
</tr>
<tr>
<td>distance_to_arterial</td>
<td>float</td>
<td><em>(optional)</em> Units: urbansim_constants.units</td>
</tr>
<tr>
<td>distance_to_highway</td>
<td>float</td>
<td><em>(optional)</em> Units: urbansim_constants.units</td>
</tr>
<tr>
<td>governmental_sqft</td>
<td>integer</td>
<td>The sum of the square footage of buildings that are classified as governmental</td>
</tr>
<tr>
<td>industrial_sqft</td>
<td>integer</td>
<td>The sum of the square footage of buildings that are classified as industrial</td>
</tr>
<tr>
<td>commercial_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>industrial_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>governmental_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>nonresidential_land_value</td>
<td>integer</td>
<td>Units, e.g. dollars</td>
</tr>
<tr>
<td>residential_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>residential_land_value</td>
<td>integer</td>
<td>Units, e.g. dollars</td>
</tr>
<tr>
<td>residential_units</td>
<td>integer</td>
<td>Number of residential units</td>
</tr>
<tr>
<td>relative_x</td>
<td>integer</td>
<td>X coordinate in grid coordinate system</td>
</tr>
<tr>
<td>relative_y</td>
<td>integer</td>
<td>Y coordinate in grid coordinate system</td>
</tr>
<tr>
<td>year_built</td>
<td>integer</td>
<td>e.g. 2002</td>
</tr>
<tr>
<td>plan_type_id</td>
<td>integer</td>
<td>An id indicating the plan type of the grid cell</td>
</tr>
<tr>
<td>percent_agricultural_protected_land</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by agricultural protected land</td>
</tr>
<tr>
<td>percent_water</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by water</td>
</tr>
<tr>
<td>percent_stream_buffer</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by stream buffer</td>
</tr>
<tr>
<td>percent_floodplain</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by flood plain</td>
</tr>
<tr>
<td>percent_wetland</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by wetland</td>
</tr>
<tr>
<td>percent_slope</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by slope</td>
</tr>
<tr>
<td>percent_open_space</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by open space</td>
</tr>
<tr>
<td>percent_public_space</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by public space</td>
</tr>
<tr>
<td>percent Roads</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by roads</td>
</tr>
<tr>
<td>percent_undevelopable</td>
<td>integer</td>
<td><em>(optional)</em> Percentage of this cell covered by undevelopable</td>
</tr>
<tr>
<td>is_outside_urban_growth_boundary</td>
<td>boolean</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>is_state_land</td>
<td>boolean</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>is_federal_land</td>
<td>boolean</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>is_inside_military_base</td>
<td>boolean</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>is_inside_national_forest</td>
<td>boolean</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>is_inside_tribal_land</td>
<td>boolean</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>zone_id</td>
<td>integer</td>
<td>Traffic analysis zone that contains this grid cell’s centroid</td>
</tr>
<tr>
<td>city_id</td>
<td>integer</td>
<td><em>(optional)</em> City this grid cell belongs to</td>
</tr>
<tr>
<td>county_id</td>
<td>integer</td>
<td><em>(optional)</em> County this grid cell belongs to</td>
</tr>
<tr>
<td>fraction_residential_land</td>
<td>float</td>
<td>Fraction of residential land in this cell</td>
</tr>
<tr>
<td>total_nonres_sqft</td>
<td>integer</td>
<td><em>(optional)</em></td>
</tr>
<tr>
<td>total_undevelopable_sqft</td>
<td>integer</td>
<td><em>(optional)</em></td>
</tr>
</tbody>
</table>

- fraction_residential_land must be between 0 and 1
- commercial_sqft, governmental_sqft and residential_units must be >= 0
- development_type_id must be a valid index in the development_types table
- distance_to_arterial, distance_to_highway must be >= 0
• grid_id must be unique and > 0
• commercial_improvement_value, industrial_improvement_value, residential_improvement_value, and governmental_improvement_value must be >= 0
• nonresidential_land_value and residential_land_value must be >= 0
• relative_x, relative_y coordinate pairs must be unique, and >= 1.
• The relative_x and relative_y columns are measured in grid cell units. They are specifically not latitude/longitude or any other universal measurement system. For example this sparse grid (6 cells in a 3x3 grid; cells are labeled with grid_id, relative_x, relative_y):
  
  (1,1,1)  (2,1,2)  - 
  (3,1,2)  (4,2,2)  (5,3,2) 
  - - (6,3,3)  

• year_built must be less than or equal to the start date of the scenario and larger than absolute_min_year which can be defined in the urbansim_constants table (default is 1800).
• plan_type must be a valid index in the plan_types table
• All percent_* attributes must be between 0 and 100
• zone_id must be a valid id in the zones table
• city_id must be a valid index into the cities table or zero if there is no city
• county_id must be a valid index into the counties table or zero if there is no county
• For gridcells with any households on them (i.e., household.grid_id = gridcell.grid_id), the gridcell.residential_units must be greater than 0.

19.1.2 The plan_types table

plan_types are synonymous with Zoning types: for example “residential2”. Also synonymous with Planned Land Use (PLU) types. The distinction is arbitrary and is to be made by the user.

One row per plan type.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plan_type_id</td>
<td>integer</td>
<td>Unique identifier</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Unique name of the Plan Type</td>
</tr>
</tbody>
</table>

• plan_type_id must be unique, greater than zero, and less than or equal to 9999. We recommend that plan_type_id start at 1 and be sequential.
• name must be unique.

19.2 Database Tables about Development Types

Development types are used to classify a grid cell according to the “type” of development currently in the grid cell. For instance, grid cells with only a few residential units and no other square footage might be classified as “low density residential” which may be abbreviated as “R1”. Other grid cells may be classified as mixed use, commercial, etc. The set of development types to use is arbitrary.

Development types are grouped by two nested mechanisms: groups and non-overlapping-groups. Each development type may be a member of multiple groups. Each group may be a member of multiple non-overlapping-groups. All of
the groups in a non-overlapping-group must be disjoint (i.e., may not share any development types); in other words, each development type must belong to at most one group in each non-overlapping-group.

Groups and non-overlapping-groups are used in the computation of the variables in the models, so to fully understand them requires understanding the model definitions.

19.2.1 The development_types table

This table is used by the Events Coordinator (see Section 25.4.18). Each row defines one development type.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_type_id</td>
<td>integer</td>
<td>Unique identifier for this row.</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Name of the development type.</td>
</tr>
<tr>
<td>min_units</td>
<td>integer</td>
<td>Minimum number of units to be in this development type.</td>
</tr>
<tr>
<td>max_units</td>
<td>integer</td>
<td>Maximum number of units to be in this development type.</td>
</tr>
<tr>
<td>min_sqft</td>
<td>integer</td>
<td>Minimum square feet to be in this development type.</td>
</tr>
<tr>
<td>max_sqft</td>
<td>integer</td>
<td>Maximum square feet to be in this development type.</td>
</tr>
</tbody>
</table>

- development_type_id must be unique and greater than zero. We recommend that it starts at 1 and is sequential.
- min_units must be >= 0.
- max_units must be >= min_units.
- min_sqft must be >= 0.
- max_sqft must be >= min_sqft.
- The development types should not overlap, and should completely cover the space. A grid cell should only be able to be in a single development type.

19.2.2 The development_type_groups table

Each row defines one development type group, but not the group’s membership - the memberships are defined in the development_type_group_definitions table.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_id</td>
<td>integer</td>
<td>Unique identifier for this row.</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>Unique name of the development type group.</td>
</tr>
<tr>
<td>non_overlapping_groups</td>
<td>varchar</td>
<td>Name of the non-overlapping-group or empty for no non-overlapping-group.</td>
</tr>
</tbody>
</table>

- group_id must be unique, and greater than zero.
- name must be unique. The required development type groups must be lower case with underscores between words e.g. high_density_residential. We recommend that all names follow this style.
- names and non_overlapping_groups names must not contain spaces must be lower-case.
- Development types must not overlap across the groups in the same non_overlapping_groups.

The set of required development type groups and non-overlapping-groups is determined by the set of variables used by the models being estimated or simulated. Thus, there is no way to a-priori specify which development type groups will be needed for your application of UrbanSim. There are two exceptions: First, the model Events Coordinator (25.4.18) is internally using groups ‘residential’, ‘mixed_use’, ‘commercial’, ‘industrial’, and ‘governmental’. Second, the Land Price Model (25.4.1) is using by default a filter that requires a group called ‘developable’. Therefore, if you do not change this settings, make sure your table contain these entries.
19.2.3 The `development_type_group_definitions` table

This table defines the set of `development_types` in each development type group. Each row defines one “belongs to” relationship (a particular development type that “belongs to” a particular development type group).

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_type_id</td>
<td>integer</td>
<td>Index into the <code>development_types</code> table</td>
</tr>
<tr>
<td>group_id</td>
<td>integer</td>
<td>Index into the <code>development_type_groups</code> table</td>
</tr>
</tbody>
</table>

- development_type_id must be a valid index into the `development_types` table
- group_id must be a valid index into the `development_type_groups` table
- The combination of development_type_id and group_id must be unique

19.3 Database Tables about Development Events

These tables represent events in the real estate development. Events that are scheduled to take place in the future are stored in the `development_events` table, events that occurred prior to the base year are stored in the `development_event_history` table.

Both tables can contain columns of the pattern “units_change_type”. Each value determines a type of change for that type of units. Possible values are:

- “A” for Add
- “R” for Replace
- “D” for Delete

If this column is missing for a certain type of units, the default value is “A” for all events.

19.3.1 The `development_events` table

These development events are changes to grid cells which are scheduled to take place in the future. For any given year, it is possible to schedule any number of changes to the attributes of any number of gridcells. Each change represents that addition, subtraction or replacement of the specified number of sqft, residential units, and improvement values. For example, if grid cell 23 is to grow by 200 residential units in 2008 (an apartment building is built), the table would include a row with scheduled_year = 2008, grid_id = 23, residential_units = 200, and residential_units_change_type = 'A'.

The value in the “improvement_value” fields, below, are used to indicate how to change the associated improvement_value for this grid cell. Each event will add/subtract/replace (improvement_value * (number of units [or sqft] being built by this event)) to the current improvement value in this grid cell. The units of the improvement is currency value, e.g. dollars.

The described procedure is implemented in Events Coordinator (see Section 25.4.18).
### The development_event_history table

The development event history records the development events that occurred prior to the base year. It is used by the development project transition model (25.4.14), and for “unrolling” the baseyear to create versions of the gridcell data for prior years.

This table uses a subset of the schema used for development_events. It can be considered an extension back in time of the development_events table, though with additional constraints, specified below.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid_id</td>
<td>integer</td>
<td>Grid cell where the event takes place</td>
</tr>
<tr>
<td>scheduled_year</td>
<td>short</td>
<td>Year in which the event was implemented</td>
</tr>
<tr>
<td>starting_development_type_id</td>
<td>integer</td>
<td><em>(optional)</em> This will be the value of the development_type for this gridcell after “unrolling” this development event.</td>
</tr>
<tr>
<td>residential_units</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>commercial_sqft</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>industrial_sqft</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>governmental_sqft</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>residential_units_change_type</td>
<td>char</td>
<td><em>(optional) see 19.3</em></td>
</tr>
<tr>
<td>commercial_sqft_change_type</td>
<td>char</td>
<td><em>(optional) see 19.3</em></td>
</tr>
<tr>
<td>industrial_sqft_change_type</td>
<td>char</td>
<td><em>(optional) see 19.3</em></td>
</tr>
<tr>
<td>governmental_sqft_change_type</td>
<td>char</td>
<td><em>(optional) see 19.3</em></td>
</tr>
<tr>
<td>residential_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>commercial_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>industrial_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
<tr>
<td>governmental_improvement_value</td>
<td>integer</td>
<td>See description, above</td>
</tr>
</tbody>
</table>

- grid_id must be a valid id in the gridcells table
- starting_development_type_id must be a valid index into the development_types table
- starting_development_type_id is only required if the process of unrolling gridcells is activated.
- Entries for which scheduled_year is greater than or equal to the base year will not be used.
19.4 Database Tables About Development Constraints

19.4.1 The `development_constraints` table

This table defines rules that restrict the possible development types a developer can create on a particular grid cell. Each row defines one rule. Development is not allowed on any grid cell that matches any of these rules. A grid cell matches a rule if the attribute values for the grid cell match all of the values in the rule (rule columns with the value “-1” are ignored when determining a match). The table is used by the Development Project Location Choice Model (see page 103 and Section 25.4.6) when computing the amount of allowed development in grid cells.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_id</td>
<td>integer</td>
<td>Unique rule identification number</td>
</tr>
<tr>
<td><code>name-of-a-gridcell-attribute-1</code></td>
<td>integer or float</td>
<td>Value for this attribute, or “-1” if this attribute is not part of the constraint (e.g. don’t care)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>name-of-a-gridcell-attribute-N</code></td>
<td>integer or float</td>
<td>...</td>
</tr>
<tr>
<td>min_units</td>
<td>integer</td>
<td>Minimum number of residential units for a grid cell. A development project may only be placed on this grid cell if it will result in this grid cell containing at least this number of residential units.</td>
</tr>
<tr>
<td>max_units</td>
<td>integer</td>
<td>Maximum number of residential units for a grid cell. A development project may only be placed on this grid cell if it will result in this grid cell containing at most this number of residential units.</td>
</tr>
<tr>
<td>min_commercial_sqft</td>
<td>integer</td>
<td>Minimum number of commercial sqft. for a grid cell. A development project may only be placed on this grid cell if it will result in this grid cell containing at least this number of commercial sqft.</td>
</tr>
<tr>
<td>max_commercial_sqft</td>
<td>integer</td>
<td>Maximum number of commercial sqft. for a grid cell. A development project may only be placed on this grid cell if it will result in this grid cell containing at most this number of commercial sqft.</td>
</tr>
<tr>
<td>min_industrial_sqft</td>
<td>integer</td>
<td>Minimum number of industrial sqft. for a grid cell. A development project may only be placed on this grid cell if it will result in this grid cell containing at least this number of industrial sqft.</td>
</tr>
<tr>
<td>max_industrial_sqft</td>
<td>integer</td>
<td>Maximum number of industrial sqft. for a grid cell. A development project may only be placed on this grid cell if it will result in this grid cell containing at most this number of industrial sqft.</td>
</tr>
</tbody>
</table>

- `constraint_id` must be a unique positive integer.
- `name-of-a-gridcell-attribute-[1...N]` are names of gridcell attributes, e.g. `city_id` or `is_in_wetland`. The set of available attribute names is determined by the set of numeric column names on the grid cell table.
- Within each `min/max` pair, the max must be greater than or equal to the min, e.g., `min_units <= max_units`.

19.5 Database Tables About Target Vacancies

19.5.1 The `target_vacancies` table
This version of `target_vacancies` table is used by the development project transition model (25.4.14) in gridcell-based applications. It gives the model information about acceptable vacancy rates. The table has one row for each year the simulation runs. Each row gives target values for the residential and nonresidential vacancies for that year, which are defined below. Only data after the base year is used.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>integer</td>
<td>Year of the simulation for which the vacancy targets apply</td>
</tr>
<tr>
<td>target_total_residential_vacancy</td>
<td>float</td>
<td>Ratio of unused residential units to total residential units</td>
</tr>
<tr>
<td>target_total_non_residential_vac</td>
<td>float</td>
<td>Ratio of unused nonresidential sqft to total nonresidential sqft</td>
</tr>
</tbody>
</table>

- There must be exactly one row for each year to be simulated.
- `target_total_residential_vacancy` and `target_total_non_residential_vacancy` must be between 0 and 1, inclusive.
20.1 Database Tables About Parcels

20.1.1 The parcels table

This table contains attributes about parcels. In general, there will be an identifier in this table for every other level of geography that you may want to aggregate up to. In this example, there are attributes for zones, cities, counties, census blocks, etc. Having these identifiers on the parcel makes it easier to aggregate indicators up to higher level geographies. Any other attributes that one may want to restrict development by, or update throughout a simulation could be stored here as well.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parcel_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>zone_id</td>
<td>integer</td>
<td>id number for the zone that the parcel’s centroid falls within</td>
</tr>
<tr>
<td>land_use_type_id</td>
<td>integer</td>
<td>identifies the land use of the parcel</td>
</tr>
<tr>
<td>city_id</td>
<td>integer</td>
<td>id number for the city that the parcel’s centroid falls within</td>
</tr>
<tr>
<td>county_id</td>
<td>integer</td>
<td>id number for the county that the parcel’s centroid falls within</td>
</tr>
<tr>
<td>plan_type_id</td>
<td>integer</td>
<td>id number that identifies the parcel’s plan type</td>
</tr>
<tr>
<td>parcel_sqft</td>
<td>integer</td>
<td>square feet of the parcel as an integer</td>
</tr>
<tr>
<td>assessor_parcel_id</td>
<td>integer</td>
<td>(optional) original tax assessor’s id number</td>
</tr>
<tr>
<td>tax_exempt_flag</td>
<td>integer</td>
<td>(optional) identifies parcel as tax exempt or not</td>
</tr>
<tr>
<td>land_value</td>
<td>long</td>
<td>value of the land from the assessor</td>
</tr>
<tr>
<td>is_in_flood_plain</td>
<td>integer</td>
<td>(optional) indicates whether or not a parcel is in a flood plain</td>
</tr>
<tr>
<td>is_on_steep_slope</td>
<td>integer</td>
<td>(optional) indicates whether or not a parcel is on a steep slope</td>
</tr>
<tr>
<td>is_in_fault_zone</td>
<td>integer</td>
<td>(optional) indicates whether or not a parcel is in a fault zone</td>
</tr>
<tr>
<td>centroid_x</td>
<td>long</td>
<td>state plane x coordinate of parcel centroid</td>
</tr>
<tr>
<td>centroid_y</td>
<td>long</td>
<td>state plane y coordinate of parcel centroid</td>
</tr>
<tr>
<td>census_block_id</td>
<td>integer</td>
<td>(optional) id number for the census block that the parcel’s centroid falls within</td>
</tr>
<tr>
<td>raz_id</td>
<td>integer</td>
<td>(optional) id number for the raz that the parcel’s centroid falls within</td>
</tr>
</tbody>
</table>
20.2 Database Tables about Buildings

20.2.1 The buildings table

In all recently developed UrbanSim applications, buildings of all types have been represented in their own table, and linked to the unit of geography used for location choice: gridcell, parcel, or zone. This configuration provides a simple and flexible means of organizing the data for UrbanSim. The buildings table is similar for each of the types of applications, whether gridcell, parcel or zone – the only significant difference is the location identifier. In the table below, parcel id is included, but for gridcell or zone applications, the user should substitute gridcell id or zone id.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>building_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>building_quality_id</td>
<td>integer</td>
<td>(optional) identified for building quality</td>
</tr>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>identifier for building type; valid id in the building_types table</td>
</tr>
<tr>
<td>improvement_value</td>
<td>long</td>
<td>value of building (replacement cost)</td>
</tr>
<tr>
<td>land_area</td>
<td>long</td>
<td>land area (usually in sqft) associated with building, includes footprint plus associated area such as landscaping and parking.</td>
</tr>
<tr>
<td>non_residential_sqft</td>
<td>long</td>
<td>non-residential square footage of building</td>
</tr>
<tr>
<td>parcel_id</td>
<td>integer</td>
<td>identifier of parcel in which building is located</td>
</tr>
<tr>
<td>residential_units</td>
<td>integer</td>
<td>number of residential units in the building</td>
</tr>
<tr>
<td>sqft_per_unit</td>
<td>integer</td>
<td>number of residential square feet per unit in the building</td>
</tr>
<tr>
<td>stories</td>
<td>integer</td>
<td>(optional) number of stories in the building</td>
</tr>
<tr>
<td>tax_exempt</td>
<td>integer</td>
<td>(optional) indicator for whether building is tax-exempt</td>
</tr>
<tr>
<td>year_built</td>
<td>integer</td>
<td>year of construction of the building</td>
</tr>
</tbody>
</table>

20.2.2 The building_types table

This is a table about available types of buildings.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>building_type_name</td>
<td>varchar</td>
<td>name of the building type</td>
</tr>
<tr>
<td>description</td>
<td>varchar</td>
<td>(optional) description of the building type</td>
</tr>
<tr>
<td>generic_building_type_id</td>
<td>integer</td>
<td>(optional) identifier for generic building type</td>
</tr>
<tr>
<td>generic_building_type_desc</td>
<td>varchar</td>
<td>(optional)</td>
</tr>
<tr>
<td>is_residential</td>
<td>boolean</td>
<td>1 if this building type is residential, 0 otherwise</td>
</tr>
<tr>
<td>unit_name</td>
<td>varchar</td>
<td>name of units for this building type, e.g. ‘commercial_sqft’ or ‘residential_units’</td>
</tr>
</tbody>
</table>

20.3 Database Tables About Development Projects

20.3.1 The development_project_proposals table

A record in this table, when combined with one or more records in the development_project_components table, represents a "known" development project. This table would be populated with projects known to be coming in the future. This table would also be populated during a simulation run for projects that are not yet complete, in other words, projects that are in the middle of developing according to their velocity function. It is entirely possible for a simulation
run to happen without pre-populating this table with records.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_project_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>development_template_id</td>
<td>integer</td>
<td>indicates the development template that represents the project</td>
</tr>
<tr>
<td>far</td>
<td>float</td>
<td>floor to area ratio of the project</td>
</tr>
<tr>
<td>percent_open</td>
<td>integer</td>
<td>the percent of the land area of the project accounted for by &quot;overhead&quot; uses such as rights of way or open space</td>
</tr>
<tr>
<td>status_id</td>
<td>integer</td>
<td>this represents active, proposed, or planned developments with the following codes: 1: in active development, 2: proposed for development, 3: planned and will be developed, 4: tentative, 5: not available (already developed, 6: refused</td>
</tr>
<tr>
<td>parcel_id</td>
<td>integer</td>
<td>indicates the parcel_id on which the development occurs</td>
</tr>
<tr>
<td>start_year</td>
<td>integer</td>
<td>the year in which this project is expected to begin building</td>
</tr>
<tr>
<td>built_sqft_to_date</td>
<td>integer</td>
<td>the number of non-residential sqft built in the current simulation year</td>
</tr>
<tr>
<td>built_units_to_date</td>
<td>integer</td>
<td>the number of residential units built in the current simulation year</td>
</tr>
</tbody>
</table>

**20.3.2 The development_project_proposal_components table**

A record in this table represents a portion of a development project identified in the development_project_proposals table. In some sense a single record here is meant to represent a single building, or part of a building. Therefore individual records here do not necessarily represent single free-standing buildings, although they are mostly treated that way. This table allows for the flexible representation of mixed uses to occur on a parcel. Examples include multiple free-standing buildings with different uses, a single building with multiple uses inside of it (a single record for each use), or further complex representations of mixed use. This table is not required by UrbanSim, but it is created by the developer model and cached every simulation year.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_project_component_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>development_project_id</td>
<td>integer</td>
<td>identifies which development the project belongs to</td>
</tr>
<tr>
<td>percent_of_building_sqft</td>
<td>integer</td>
<td>identifies the percentage of the building that this component takes up. 100% would indicate a free-standing building with a single use, and several records with percent_of_building_sqft adding up to 100% would indicate a multiple use single building.</td>
</tr>
<tr>
<td>construction_cost_per_unit</td>
<td>integer</td>
<td>the per unit construction cost for residential uses only</td>
</tr>
<tr>
<td>sqft_per_unit</td>
<td>integer</td>
<td>the square footage per residential unit</td>
</tr>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>indicates the building type of this particular component</td>
</tr>
<tr>
<td>land_area</td>
<td>integer</td>
<td>the land area &quot;claimed&quot; by the building component, including not only the building footprint but also additional land used such as yards, parking lots, etc.</td>
</tr>
<tr>
<td>residential_units</td>
<td>integer</td>
<td>the number of residential units in the building component</td>
</tr>
</tbody>
</table>

**20.3.3 The development_templates table**

This table, along with corresponding records in the development_template_components table, represents development templates that can be used to define virtually any size and configuration of a development project, from a single house on an infill lot to a large subdivision, to a mixed use project with retail on the first floor and condominiums above. The contents of this table are roughly comparable to the development_projects table, since development templates become
proposals once they are determined to fit within a parcel and are allowed by development constraints, and then become projects if they are chosen to be constructed. See also page 106.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_template_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>percent_open</td>
<td>integer</td>
<td>the percent of the land area of the project accounted for by &quot;overhead&quot; uses such as rights of way or open space</td>
</tr>
<tr>
<td>min_land</td>
<td>integer</td>
<td>minimum amount of land in square feet to be utilized for this development</td>
</tr>
<tr>
<td>max_land</td>
<td>integer</td>
<td>maximum amount of land in square feet to be utilized for this development</td>
</tr>
<tr>
<td>density_type</td>
<td>integer</td>
<td>a readable name that describes the ‘density’ field: units per acre, FAR</td>
</tr>
<tr>
<td>density</td>
<td>integer</td>
<td>indicates the density of the development</td>
</tr>
<tr>
<td>land_use_type_id</td>
<td>integer</td>
<td>specifies the land use type for the development template</td>
</tr>
<tr>
<td>development_type</td>
<td>integer</td>
<td>a readable name that describes the type of development this record represents (e.g. SFR-parcel, MFR-apartment, MFR-condo, etc.), this field is not used by the model and is there to make the table more readable</td>
</tr>
</tbody>
</table>

20.3.4 The development_template_components table

This table is roughly equivalent to the development_project_proposal_components table and represents buildings or parts of buildings to be included in a particular development template. By breaking development templates into components, development project templates can be configured as hierarchies or combinations of building blocks, providing a very flexible means of representing a wide variety of development types. Note that the templates can be generated using real or hypothetical data, since they will be compared to regulatory constraints and the size constraints of parcels.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_template_component_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>development_template_id</td>
<td>integer</td>
<td>indicates which development template this component belongs to</td>
</tr>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>indicates the building type of this particular component</td>
</tr>
<tr>
<td>percent_of_building_sqft</td>
<td>integer</td>
<td>identifies the percentage of the building that this component takes up</td>
</tr>
<tr>
<td>construction_cost_per_unit</td>
<td>integer</td>
<td>the per unit construction cost</td>
</tr>
<tr>
<td>building_sqft_per_unit</td>
<td>integer</td>
<td>the square footage per residential unit</td>
</tr>
</tbody>
</table>

20.3.5 The velocity_functions table

This table is designed to hold the velocity functions that specify the rate at which development is built out. A Development Project Proposal has a calculated variable called units_proposed that is the total number of units that will be built. The calculated variable annual_construction_schedule on the Development Project Proposal Components dataset uses units_proposed to select which of the velocity functions in this table should apply based on the building_type_id and units_proposed of the Development Project Proposal Component.
### 20.3.6 The `demolition_cost_per_sqft` Table

This table provides information to the developer model about the costs of demolition by building type. These numbers are used to calculate the cost of demolition of existing development so that a more accurate cost of redevelopment can be calculated.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>building type</td>
</tr>
<tr>
<td>demolition_cost_per_sqft</td>
<td>integer</td>
<td>cost in dollars per sqft of demolition</td>
</tr>
</tbody>
</table>

### 20.3.7 The `building_sqft_per_job` Table

This table contains information on the amount of space each job will take in a particular building type, by zone.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone_id</td>
<td>integer</td>
<td>the zone the record applies to</td>
</tr>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>the building type the record applies to</td>
</tr>
<tr>
<td>building_sqft_per_job</td>
<td>integer</td>
<td>the sqft per job each job will take in a particular building type in a particular zone</td>
</tr>
</tbody>
</table>

### 20.4 Database Tables About Development Constraints

#### 20.4.1 The `development_constraints` Table

Note that this table is substantially different than its counterpart for a gridcell-based model application. This is because the real estate development model is fundamentally different as well. The model that uses this table in a parcel-based application is the development_project_proposal_sampling_model (see page 103), which evaluates alternative development templates that can be placed on a parcel according to the constraints specified in this table, and then computes a return on investment on the remaining viable proposals, before choosing which proposals to build. This
table defines rules that restrict the possible development types a developer can create on a particular parcel. Each row defines one rule. Development is not allowed on any parcel that matches any of these rules. See also page 105.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint_id</td>
<td>integer</td>
<td>Unique rule identification number</td>
</tr>
<tr>
<td>constraint_type</td>
<td>string (14)</td>
<td>units_per_acre or far (floor-area-ratio)</td>
</tr>
<tr>
<td>generic_land_use_type_id</td>
<td>integer</td>
<td>Id of a record in the generic_land_use_types table</td>
</tr>
<tr>
<td>maximum</td>
<td>integer</td>
<td>Maximum value for the allowed development, in terms of the constraint type</td>
</tr>
<tr>
<td>minimum</td>
<td>integer</td>
<td>Minimum value for the allowed development, in terms of the constraint type</td>
</tr>
<tr>
<td>plan_type_id</td>
<td>integer</td>
<td>Id of a record in the plan_types table.</td>
</tr>
</tbody>
</table>

- constraint_id must be a unique positive integer.

### 20.5 Database Tables About Target Vacancy Rates

#### 20.5.1 The target_vacancies table

The target_vacancies table is used by the development proposal choice model. It gives the model information about acceptable vacancy rates. The table has one row for each year the simulation runs. Each row gives target values for the residential and nonresidential vacancies for that year, which are defined below. Only data after the base year is used.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>integer</td>
<td>Year of the simulation for which the vacancy targets apply</td>
</tr>
<tr>
<td>target_vacancy</td>
<td>float</td>
<td>Ratio of unused space to total space, based on residential_unit or sqft</td>
</tr>
<tr>
<td>building_type_id</td>
<td>Integer</td>
<td>Id of a record in the building_types table</td>
</tr>
</tbody>
</table>

- There must be exactly one row for each year to be simulated.
- target_vacancy must be between 0 and 1, inclusive.

### 20.6 Database Tables About Refinement of Simulation Results

#### 20.6.1 The refinements table

This table is used by the refinement model to adjust simulation results to incorporate added information or constraints specified by the user. The entries in this table define refinements to make to an existing simulation run. No fields can be null, if the attribute is not needed put a single quote (‘) in the field.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refinement_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>agent_expression</td>
<td>string</td>
<td>string expression defining what agents to add or subtract, e.g. households, jobs</td>
</tr>
<tr>
<td>location_capacity_attribute</td>
<td>string</td>
<td>defines a capacity attribute such as non_residential_sqft</td>
</tr>
<tr>
<td>location_expression</td>
<td>string</td>
<td>expression defining where to add or subtract agents, e.g. ‘zone = 123’</td>
</tr>
<tr>
<td>amount</td>
<td>integer</td>
<td>number of agents to add or subtract</td>
</tr>
<tr>
<td>year</td>
<td>integer</td>
<td>year to which this refinement applies</td>
</tr>
<tr>
<td>action</td>
<td>string</td>
<td>add, subtract, or target are the valid entries</td>
</tr>
<tr>
<td>transaction_id</td>
<td>integer</td>
<td>if two or more records have matching transaction ids the refinement model will attempt to balance between the refinements</td>
</tr>
</tbody>
</table>
Data for Zone-based Applications

The zone based modeling is the newest model system and should be considered experimental at this point. The zone based model system was itself modeled after that gridcell model system. Consequently many of the tables are common with it. Here are tables unique to the zone based model system.

21.1 Database Tables About Buildings

21.1.1 The pseudo_buildings table

As part of the creation and testing of a zonal-level version of UrbanSim, a pseudo-buildings table has been created to contain the summary contents of the real estate development in a zone. Pseudo buildings are meant to represent the amount of commercial, governmental, industrial, and residential space in a zone. There are 4 pseudo building records per zone_id, 1 each for each of the land uses. The attributes are updated during the simulation run by the model system.
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pseudo_building_id</td>
<td>integer</td>
<td>unique identifier</td>
</tr>
<tr>
<td>annual_growth</td>
<td>integer</td>
<td><em>(optional)</em> this is the amount that this type of building is allowed to grow per simulation year in terms of floor space or residential units</td>
</tr>
<tr>
<td>residential_units</td>
<td>integer</td>
<td>the number of residential units for residential pseudo buildings</td>
</tr>
<tr>
<td>zone_id</td>
<td>integer</td>
<td>the zone in which this pseudo building is in</td>
</tr>
<tr>
<td>avg_value</td>
<td>integer</td>
<td>the average value per unit or job space depending on the building_type_id</td>
</tr>
<tr>
<td>building_type_id</td>
<td>integer</td>
<td>the building type that matches up with the building_types table</td>
</tr>
<tr>
<td>job_spaces_capacity</td>
<td>integer</td>
<td>the total number of job spaces allowed in this pseudo building</td>
</tr>
<tr>
<td>residential_units_capacity</td>
<td>integer</td>
<td>the total number of residential units allowed in this pseudo building</td>
</tr>
<tr>
<td>commercial_job_spaces</td>
<td>integer</td>
<td>the total number of commercial job spaces currently in this pseudo building</td>
</tr>
<tr>
<td>industrial_job_spaces</td>
<td>integer</td>
<td>the total number of industrial job spaces currently in this pseudo building</td>
</tr>
<tr>
<td>governmental_job_spaces</td>
<td>integer</td>
<td>the total number of governmental job spaces currently in this pseudo building</td>
</tr>
</tbody>
</table>
Part V

Command-line Interface to Opus and UrbanSim
Running a Simulation or Estimation

22.1 Running a Simulation

An alternative way of running a simulation from the GUI (as described in Chapter 9) is to launch it from a command line. Opus contains a set of scripts that simplify the process of creating a base year cache, starting, and restarting a set of simulation runs. These scripts are a set of command line applications, or tools, in the 'opus_core/tools' directory. This section describes how to run a simulation using these tools.

All scripts described below print a help message when called with the -h or --help option.

22.1.1 Run Management

The Services Database

Before we begin, it is important to note that the Opus framework uses a database to store information about the simulations that have been run. This database is by default called services.

The default database engine, hostname and other necessary information for the services database is set in 'OPUS_HOME/settings/database_server_configurations.xml'. By default, it will use sqlite (a database management system that just uses local files). You can, however, update the appropriate settings in the XML to configure the services database to use a mysql, postgres, or mssql server. The services database is created automatically on the specified database server by Opus if it does not yet exist when it is needed.

Create Baseyear Cache

Opus simulations usually run on data stored in a base year cache. If the raw data are stored in a database, one can use an Opus tool to create such a base year cache. The script needs a configuration module passed as an argument. If the module is for example located at 'psrc/configs/subset_configuration.py' under one of the paths found in the PYTHONPATH environment variable, the command

```
python create_baseyear_cache.py psrc.configs.subset_configuration
```

caches the database defined in the subset_configuration module into a baseyear cache. Run this command from the 'opus_core/tools' directory. An option --cache-directory can be used to pass the directory to be cached into. Alternatively, this can be specified in the configuration as an entry 'cache_directory'. See also
Section 22.3 for configuration and cache control options.

Start a simulation using a Python Dictionary Configuration

The configuration for a simulation is specified by a configuration object. Opus supports a configuration defined either as a Python dictionary, or as an xml project file (see Section 22.1.1). This section describes the former. In both cases, the configuration is used to specify different parts of the simulation, such as the years in which to run UrbanSim, what UrbanSim models to run each year, what types of development projects exist, how to configure each type of development project, etc. See, for instance, the run configuration in seattle_parcel/configs/baseline.py (Python dictionary version) or seattle_parcel/configs/seattle_parcel.xml (xml version).

To use the dictionary version of a configuration, it is required that the referenced configuration Python module defines a class (e.g. SubsetConfiguration) in a file whose name is subset_configuration.py. The class name should be the CamelCase version of the lowercase_with_underscores file name.

To start a simulation using a dictionary-based configuration, use the script start_run with the desired configuration. First, change to the directory opus_core/tools and (using for example the configuration from the previous section) execute:

```
python start_run.py -c psrc.configs.subset_configuration
```

Use the --help option to see the possible command line parameters for start_run. Also see Section 22.3.1 for configuring a simulation run.

Start a simulation using an XML Configuration

To start a simulation using an xml configuration from the command line, use the script start_run with the desired configuration. Change to the directory that holds the Opus source code and execute:

```
python opus_core/tools/start_run.py -x seattle_parcel/configs/seattle_parcel.xml -s Seattle_baseline
```

(typed all on one line).

Notice that the -x option takes a path to the file with the xml configuration. Since xml configurations can hold multiple scenarios, the scenario name must also be specified using the -s option.

What Happens When Running a Simulation?

Here are the steps that occur when you start a run via the start_run script:

- It copies files from the baseyear cache into a cache for the current run.
- It adds a row to the run_activity table in the services database, using a new run_id value unique to this run. In order to help match runs with their cache directory, the name of the cache directory begins with the run_id value, e.g. 'run_342.2006_04_25_09_40'.
- For each simulated year:
-- Forks a new process to run the set of UrbanSim models for this year. The set of models to be run is specified by the configuration. Using a separate process helps reduce memory usage, and helps reduce the impact of problems such as memory leaks. This process writes a log file named, e.g., 'year_2003_log.txt'.

- The run activity table includes status information about the run. If the simulation succeeds, it will add another row to the run activity indicating it is done. If the simulation fails, it will add a row indicating that. The run activity also contains a copy of the configuration, which is used when restarting a run.

- Whenever a row is added to the run_activity table, a row is either added or updated in the available_runs table in the services database. This table has a single row per run and records the row’s current state and information.

### Restart a simulation

If you halt a run or it fails, you can restart it at the beginning of any year. To restart the run with run_id 42 at the beginning of year 2005, do:

```bash
code python restart_run.py 42 2005 -p seattleParcel
```

from the opus_core/tools directory. If the services database is using sqlite, -p <project_name> argument needs to be provided. Note that the above command will delete any simulation cache directories for years 2005 onward, since this information is no longer valid once the simulation is restarted at the beginning of 2005. One can suppress this behavior using the option --skip-cache-cleanup.

### 22.2 Running an Estimation

Model estimation can be done in Opus GUI (see Chapter 8) and command line (see Section 23.6).

### 22.3 Configurations

A configuration is a specification of what parts to use in the simulation or estimation and how to configure each part. Parts include the list and order of models to run, where to get the input values, where to store the output data, what years to simulate, what tables to store into the UrbanSim baseyear and simulation caches, etc. Each of these parts in turn may be configurable. Configurations are used in many places in Opus. Typically, they are specified via a Python dictionary that then is used to create an instance of the Configuration class.

#### 22.3.1 Run Manager Configuration

As described in Sections 22.1.1 and 22.2, running UrbanSim simulation or estimation is controlled by a user-defined configuration. The following code, 'baseline.py', contains a fully specified configuration that influences the run management. Mandatory entries and default values for optional entries are marked in the comments. The actual values for the listed entries are only examples.
from opus_core.configuration import Configuration
from opus_core.database_management.configurations.scenario_database_configuration \
    import ScenarioDatabaseConfiguration
from opus_core.database_management.configurations.estimation_database_configuration \
    import EstimationDatabaseConfiguration
from opus_core.configurations.baseyear_cache_configuration \
    import BaseyearCacheConfiguration
from urbansim.configurations.creating_baseyear_cache_configuration \
    import CreatingBaseyearCacheConfiguration

class Baseline(Configuration):
    def __init__(self):
        config = {
            'project_name': 'urbansim_gridcell',
            'description': 'baseline',
            'model_system': 'urbansim.model_coordinators.model_system', # mandatory
            'base_year': 2000,  # default: read from table 'base_year' in cache
            'years': (2001, 2030), # mandatory
            'cache_directory': 'd:/urbansim_cache/', # mandatory
            'scenario_database_configuration': ScenarioDatabaseConfiguration(
                database_name = 'my_baseyear_database'), # mandatory for simulation
            'estimation_database_configuration': EstimationDatabaseConfiguration(
                database_name = 'my_estimation_database'), # mandatory for estim.
            'creating_baseyear_cache_configuration': CreatingBaseyearCacheConfiguration(
                default: 'opus_tmp'+random string
                cache_directory_root = 'd:/urbansim_cache',
                cache_from_database = False, # default: True
                # mandatory if 'cache_from_database' is False
                baseyear_cache = BaseyearCacheConfiguration(
                    # mandatory for this block
                    existing_cache_to_copy = 'd:/urbansim_cache/run_397.2006_05_23_18_21',
                    # default: all years in 'existing_cache_to_copy'
                    years_to_cache = range(1996,2001)
                ),
                tables_to_cache = [ # default: []
                    'gridcells', 'households', 'jobs', 'zones'
                ]
                tables_to_cache_nchunks = { # default: each table defaults to 1
                    'gridcells':2,
                },
                tables_to_copy_to_previous_years = { # default: no copied tables
                    'development_type_groups':1996, # table name and year to put it in
                    'development_types':1996,
                    'development_type_group_definitions':1996,
                    'urbansim_constants': 1996,
                },
                unroll_gridcells = True  # default: True
            ),
        Configuration.__init__(self, config)
The 'model_system' entry is the full Opus path to the model system that will be used by the run manager to run/estimate a set of models.

Entry 'years' determines for what years the simulation should run as a tuple with first and last year to run.

Entry 'cache_directory' is used by the scripts create_baseyear_cache.py and start_estimation.py only. It is not used for simulations.

Entry 'creating_baseyear_cache_configuration' contains a configuration for creating the simulation cache. Entry 'cache_directory_root' is the root directory where data should be cached during processing. The actual cache directory is created as a subdirectory of this location.

The entry 'tables_to_cache' is used by the script create_baseyear_cache.py. Only tables listed here are cached from database into the baseyear cache.

If a database table is so large that Python runs out of memory when copying it to cache, you can reduce memory usage (but increase the time it takes to cache the data) by increasing the number of “chunks” in which the dataset’s attributes are read from the table. By default, all attributes of a table are read in a single chunk. Setting the 'tables_to_cache_nchunks' configuration will tell the caching code to use that many chunks. For instance, if a dataset has 11 attributes, setting 'tables_to_chunk_nchunks' to 3 will use three chunks loading 4, 4, and 3 attributes, in each chunk.

In the 'baseyear_cache' block, the directory with the already cached data should be put into the entry existing_cache_to_copy. The run manager then copies data from that directory into the simulation cache for this run. If you want to copy only selected years, they can be specified in the entry years_to_cache as a list of those years; by default all years are copied. Note that this behaviour can be alternatively controlled directly from the command line (see start_run.py --h) which has priority over entries in this configuration.

The 'tables_to_copy_to_previous_years' entry is used when a lag variable needs to compute data for before the base year. If this is the case, add those tables to this list, and indicate the year to which to copy the tables. In general, it is safe to copy the tables to the earliest year created by the unroll gridcell process. You can determine what this year is by examining the year directories created in your baseyear cache.

The entry unroll_gridcells is specific to the urbansim gridcell project. It controls if gridcells are unrolled into years before the base year.

There are several run manager configurations in Opus. See for example the directory `psrc/configs` for configuration of different PSRC runs.

### 22.3.2 Model System Configuration

If one would pass the above configuration to the run manager, it would perform steps as described in Section 22.1.1, but no model would be run. The configuration should in addition contain entries that control what models, in what order and with what input and output should be run. It determines the behaviour of the class ModelSystem. UrbanSim basic configuration of the model system can be found in the file `urbansim/configs/general_configuration.py` as an example.

The set of models to run is specified by the entry “models”. It is a list of user-defined model names. The order in this list also specifies the order in which they are processed. For example, the UrbanSim gridcell project consists by default of following models:
Note that the list can contain a particular model multiple times if that model should run multiple times within one year, such as the “events_coordinator” or “residential_land_share_model” in the list above.

We can also define a situation when the same model should be run on different subsets of a dataset, so called model group. Then we can give the names of the group members to be run, or just configure the model group to be run on all subsets. This is the case of “employment_location_choice_model” in the list above (see Section 22.3.3 for further details).

By default, the ModelSystem runs the method run() of the listed models. Each entry in this model list can be alternatively a dictionary containing one entry: The name of the entry is the model name, the value is a list of model methods to be processed. Thus, one can combine estimation and simulation of different models.

In addition (or alternatively), the configuration can contain an entry “models_in_year”. It is a dictionary where keys are years. Each value is expected to be such list of models as above. In each year, it is checked if “models_in_year” (if it is present) contains that year. If it is the case, its list of models is run, instead of the global set of models. This allows users to set different set of models for different years, for example an additional model can be run only in the first year, or last year.

For each entry in the model list there must be a corresponding entry in the “controller” configuration which specifies how models are initialized, what methods to run and what arguments should be passed in. This will be described in Section 22.3.3.

Furthermore, the configuration can contain the following entries (the given values are defaults set by our system):

```python
'datasets_to_cache_after_each_model': [],
'flush_variables': False,
'seed': None,
'debuglevel': 0
```

The entry 'datasets_to_cache_after_each_model' specifies names of datasets that are flushed from memory to simulation cache at the end of each model run. This reduces the memory usage, but can increase the run time. We recommend to put datasets in this list that contain huge amount of data, e.g. ['gridcell', 'household', 'job'].

22.3. Configurations
'flush_variables' can further decrease the memory usage. If it is True, after each variable computation all dependent variables are flushed to simulation cache, regardless to what dataset the variables belong to. Nevertheless, it increases the run-time considerably.

Entry 'seed' specifies the seed of the random number generator that is set at the beginning of each simulated year. It is passed to the numpy function seed(). If it is None, the function reads data from /dev/urandom or seeds from the clock. See seed() in numpy.random module for more details.

'debuglevel' controls the amount of output information.

Models usually need various datasets to run with. They are specified in the configuration entry 'datasets_to_preload'. For example,

```yaml
'datasets_to_preload': {
    'gridcell':{"id_name": "grid_id"},
    'household':{}
}
```

It is a dictionary that has dataset names as keys. Each value is again a dictionary with argument-value pairs that are passed to the corresponding dataset constructor. ModelSystem creates those datasets at the beginning of each simulated year and they are accessible to the models definition in the controller through their names (see Section 22.3.3 for details). One should put here all datasets that will be passed as arguments to the model constructors or to the model methods to be processed.

From the preloaded datasets, ModelSystem creates a dataset pool (accessible to the models as dataset_pool). Creating this pool is controlled by the entry

```yaml
'dataset_pool_configuration': DatasetPoolConfiguration(
    package_order=['urbansim', 'opus_core'],
)
```

### 22.3.3 Model Controller Configuration

The run configuration can contain an entry 'models_configuration' which can include any information specific to models or common to a set of models. The value of this entry is a dictionary. Model specific information would be included in an entry of the same name as the model name used in the entry 'models' (see Section 22.3.2). The ModelSystem class makes this information available to the controller by creating two local variables: 'models_configuration' (containing the value of 'models_configuration' and available to all models) and 'model_configuration' (available to each model at the time of its processing and containing information for this model). See the variable 'models_configuration' in the file 'urbansim/configs/general_configuration.py' for an example how UrbanSim configures models.

Each model that is included in the configuration entry 'models' must have a controller entry in the 'models_configuration' entry. More specifically, the model specific section of 'models_configuration' is expected to contain an entry 'controller' for each model. For example, a controller specification for the model specified by the name 'land_price_model' would be contained in config['models_configuration']['land_price_model']['controller'].

If a model is specified as a model group it is possible to define a member specific controller, called `member_name + '_' + model_name`, e.g. 'home_based_employment_location_choice_model'. When choosing the right controller, the ModelSystem checks for the member specific name. If it is not found, it uses the group name, in this example 'employment_location_choice_model'.
The value of this controller entry is a dictionary with a few well-defined entries:

"import" A dictionary where keys are module names and values are names of classes to be imported.

"init" A dictionary with a mandatory entry "name". Its value is the name of the class (or class.method) that creates the model. It can be the name of the model class itself. Or, if the model is created via a method e.g. get_model() of a class MyModelCreator, it would be given as "MyModelCreator().get_model".

Optional entry "arguments" specifies arguments to be passed into the constructor. It is given as a dictionary of argument names and values. All values are given as character strings and are later converted by ModelSystem to python objects. If an argument value is suppose to be a character string object, it must be given in double quotes, e.g. "'my_string'".

If the model in the 'models' entry of the configuration is specified as model group, the controller must contain an entry

"group_by_attribute" Its value is a tuple of a grouping dataset name and grouping attribute (see Sec. 24.7.2). They define the specific kinds of subsets of agents on which this model can be run. This dataset must be contained in the datasets_to_preload entry of the configuration. For example, in the controller of the "employment_location_choice_model" this entry is ('job_building_type', 'name'), since the attribute 'name' of the dataset 'job_building_type' contains the various building types of jobs for which we want to run the model, i.e. 'commercial', 'governmental', 'industrial' and 'home_based'. If the 'group_members' entry (of the 'models' entry of the configuration) for this model is equal to '_all_'; the model runs for all values found in this dataset. The 'group_members' can also be a list specifying explicitly for which types the model should be run.

The ModelSystem class evaluates the given imports and creates an instance of the model by processing the 'init' entry. The remaining entries below are related to specific methods of the created model instance. As mentioned in Section 22.3.2, models that are listed in the 'models' entry of the run configuration can be also specified using a list of methods to be processed. If the list is not given, a method run() is assumed to be the only method to be processed. The ModelSystem iterates over the set of methods. It first processes a "preparation" method (if required) and then the method itself. For this purpose, the controller should contain the following entries:

'prepare_for...' where ... is the the method to be processed, e.g. 'prepare_for_run' is the method to call to prepare to run. This configuration entry is a dictionary with an optional entry 'name' giving the name of the preparation method. If 'name' is missing, the method name is assumed to be the same as this entry name. Optional entry 'arguments' specifies arguments of this method (see 'arguments' in 'init' above). Optional entry 'output' defines the name(s) of the output of this method. It can be then used as an input to other methods or models. The entry 'prepare_for...' is optional and if it's missing, no preparation procedure is invoked. There can be as many 'prepare_for...' entries as there are methods specified.

procedure The procedure name must match to the method names given in 'models' (there must be one entry per method), or be called 'run' if no methods are specified in 'models'. It is a dictionary with optional arguments 'arguments' and 'output' (see above).

The entry 'arguments' in the above items can contain any character strings that are convertable (using python’s eval()) to python objects, including python expressions. They must be objects that are known to the ModelSystem, for example datasets that are defined in 'datasets_to_preload' (described in Section 22.3.2), since those are created prior to the simulation. They can be called either by the dataset name, or using datasets['name']. Also, the model_configuration and models_configuration objects described in Section 22.3.3 can be used in 'arguments'. Other objects that ModelSystem provides are cache_storage (Storage object for the simulation cache storage in the simulated year), base_cache_storage (Storage object for the baseyear cache storage in the base year), model_resources (all preloaded datasets as an object of Resources), resources (Configuration passed into the simulation), datasets (all preloaded datasets as a dictionary), year (simulated year), base_year (the base year), dataset_pool (object of class DatasetPool
pointing to the current dataset pool), debuglevel (integer controlling the amount of output messages). If you are using any class names as arguments, you need to make sure, that those classes are known to the ModelSystem, e.g. by putting the appropriate import statement into the 'import' section of the controller.

Here is an example of a controller settings for the land price model in UrbanSim:

```python
run_configuration['models_configuration']["land_price_model"]['controller'] = {
    "import": "urbansim.models.corrected_land_price_model": "CorrectedLandPriceModel",
    "init": {"name": "CorrectedLandPriceModel"},
    "prepare_for_run": {"arguments": {"specification_storage": "base_cache_storage",
                                       "specification_table": "land_price_model_specification",
                                       "coefficients_storage": "base_cache_storage",
                                       "coefficients_table": "land_price_model_coefficients"},
                     "output": "(specification, coefficients)"
    },
    "run": {"arguments": {"n_simulated_years": "year - base_year",
                           "specification": "specification",
                           "coefficients": "coefficients",
                           "dataset": "gridcell",
                           "data_objects": "datasets",
                           "chunk_specification": "{'nchunks':2}"},
            "debuglevel": "debuglevel"
    }
}
```

Note on an implementation of model group: A constructor of a model group, must take as its first argument an object of class ModelGroupMember (Sec. 24.7.2). The controller should though ignore this argument, since the ModelSystem automatically takes care of creating this object and passing it to the model constructor.

### 22.4 Output

#### 22.4.1 Exporting the Output Data

The simulation reads and writes all of its data from the simulation cache. It does not directly read or write to any database. If you wish to move data from the simulation cache to a SQL database, use the `do_export_cache_to_sql_database.py` tool located in 'opus_core/tools'.

Deleting the File-Based Cache

The `delete_run` script in 'opus_core/tools' directory provides an easy way to delete cached run data while maintaining the consistency of the services database.

To delete all data for run with run_id 42, and remove that run's information from the services database use:

```bash
python delete_run.py --run-id 42
```
To delete a set of years without removing the information from the services database, use the `--years-to-delete` option. This option takes an arbitrary Python expression that creates a list of integers. For instance, to remove the cached data for years 2001 through 2029 use:

```
python delete_run.py --run-id 42 --years-to-delete range(2001,2030)
```
CHAPTER TWENTYTHREE

Interactive Exploration of Opus and UrbanSim

This chapter gives a tutorial on using opus_core and urbansim in an interactive mode. Running Python interactively provides a convenient way for both beginners and experienced users to experiment with features and try out code. This tutorial is directed mainly to modelers, developers and other Opus users who will experiment with single components of the package and develop new features, rather than use the system of models as whole.

If necessary, install the supporting software, install Opus, and test the installation, as described in Appendix A.

You can start a Python session by opening a command window and typing python at the command prompt. Alternatively, the Wing IDE also provides a convenient Python shell window.

Any code after the >>> is Python code. To follow along with this tutorial, enter this directly into a Python interpreter. If you are unfamiliar with Python, read Troubleshooting Python, Section 23.5, at the end of this chapter before proceeding through the code.

23.1 Working with Data Sets

A dataset in Opus is considered as a \( n \times m \) table where \( n \) is the number of entries and \( m \) is the number of characteristics, also called attributes. One of the characteristics must have unique values that are numeric larger than 0.

Suppose you have a set of household agents with two characteristics, income and number of persons per household, which are uniquely identified by household IDs. The file ‘data/tutorial/households.tab’ in the urbansim package contains an example dataset for 10 households:
In Opus datasets are independent from the physical storage of the data. A data storage is represented by a python object. We create a storage object for the ASCII file ‘households.tab’:

```python
>>> import os
>>> import urbansim
>>> us_path = urbansim.__path__[0]
>>> from opus_core.storage_factory import StorageFactory
>>> storage = StorageFactory().get_storage('tab_storage',
    storage_location = os.path.join(us_path, 'data/tutorial'))
```

The `storage` here specifies that the data are stored as an ASCII file in a table format. Opus can support many types of storage formats, including formats you define. See Section 24.2 for more details.

Now we create a household dataset with the `opus_core` class `Dataset` (see Section 24.1), using the created storage object:

```python
>>> from opus_core.datasets.dataset import Dataset
>>> households = Dataset(in_storage = storage,
    in_table_name = 'households',
    id_name='household_id',
    dataset_name='household')
```

Dataset supports lazy loading. Thus, there are no entries loaded for `households` at this moment:

```python
>>> households.get_attribute_names()
[]
```

But the dataset ‘knows’ about attributes living on the given storage:

```python
>>> households.get_primary_attribute_names()
['household_id', 'income', 'persons']
```

The data are loaded as they are needed. For example, loading the unique identifier of the dataset gives:
Other attributes can be loaded via the `get_attribute()` method which returns a numpy array:

```python
>>> households.get_attribute("income")
array([ 1000.,  2000.,  5000.,  3000.,   500., 10000.,  8000.,
       1000.,  3000., 15000.])
```

Each attribute of a Dataset is stored as a numpy array.

In the above example, each of the attributes is loaded separately. Alternatively, we can load multiple attributes at once, which can be useful when loading data from a slow storage, such as a SQL database:

```python
>>> households.load_dataset()
>>> households.get_attribute_names()
['household_id', 'persons', 'income']
```

An optional argument `attributes` can be passed to the `load_dataset()` method that specifies names of attributes to be loaded, e.g. `attributes=['income', 'persons']`.

We can also plot a histogram of the income attribute (this method requires the `matplotlib` library):

```python
>>> households.plot_histogram("income", bins = 10)
```

or (if the `rpy` library is installed)
We can investigate a correlation between attributes by plotting a scatter plot (rpy library required):

```python
>>> households.r_scatter("persons", "income")
```

Correlation coefficient: 0.919147133827

The correlation coefficient between two attributes and the correlation matrix of several attributes, respectively, can be obtained by:
A summary of data in a dataset can by given by:

```python
>>> households.summary()
```

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>mean</th>
<th>sd</th>
<th>sum</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>persons</td>
<td>2.7</td>
<td>1.34</td>
<td>27</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>income</td>
<td>4850.0</td>
<td>4749.56</td>
<td>48500</td>
<td>500</td>
<td>15000</td>
</tr>
</tbody>
</table>

Size: 10 records

identifiers:

```
household_id in range 1 - 10
```

To add an attribute to the set of households, for example each household’s location, we do

```python
>>> households.add_primary_attribute(data=[4,6,9,2,4,8,2,1,3,2], name="location")
```

```python
>>> households.get_attribute_names()
['household_id', 'persons', 'location', 'income']
```

If the attribute "location" already exists in the dataset, the values are overwritten.

To change specific values in a dataset, one can use

```python
>>> households.modify_attribute(name="location", data=[0,0], index=[0,1])
```

```python
>>> households.get_attribute("location")
array([0, 0, 9, 2, 4, 8, 2, 1, 3, 2])
```

Here the argument `index` determines the index of the data that are modified.

To determine the location of household with `household_id = 5`, do

```python
>>> households.get_data_element_by_id(5).location
4
```

In order to store data in one of the supported formats, you can use the `storage` object created at the beginning of this section, or create a new one using a different type of storage:
Each dataset should have a unique dataset name that is used as an identification in variable computation (see Sec-

tion 24.3.1).

```python
>>> households.write_dataset(out_storage=storage,
   out_table_name="households_output")
```

The urbansim package contains many pre-defined dataset classes, such as HouseholdDataset, GridcellDataset, JobDataset, ZoneDataset, FazDataset, NeighborhoodDataset, RaceDataset, RateDataset. Some datasets are described in Section 25.2, Table 25.1. They are all children of Dataset with pre-defined values for some arguments, such as id_name, in_table_name or dataset_name.

23.2 Working with Models

Generally, each model class has a method run() that runs a simulation and (if applicable) a method estimate() that runs an estimation of the model. Models are initiated by a constructor that sets class attributes common to both methods.

23.2.1 Choice Model

The class ChoiceModel implemented in opus_core is initiated with a choice set and a set of components specifying some of the model behavior. The model can be estimated and run for a set of agents (see Section 24.4.3 for details).

Initialization

Suppose we would like to simulate a process of households deciding among 3 choices using the discrete choice model theory. The number of persons in each household should influence their decisions. We will model this behavior by multinomial logit using a system of utility functions:

\[
\begin{align*}
    v_1 &= \beta_{01} \\
    v_2 &= \beta_{12}x \\
    v_3 &= \beta_{03} + \beta_{13}x
\end{align*}
\]

Here, \( \beta_{01} \) and \( \beta_{03} \) are alternative specific constants and \( x \) is a household variable "persons".

The model is initialized by
Thus, the model is composed by external implementations of model steps, such as computing utilities, computing probabilities and computing choices, specified as package.module_name. Those modules must contain a class of the same name and a method run() that performs the actual computation (see Section 24.4.3 and Sections 24.5.1-24.5.3 for more details).

**Estimation**

In order to estimate coefficients $\beta$ from the above equations, we define a specification:

```python
>>> from numpy import array
>>> from opus_core.equation_specification import EquationSpecification
>>> specification = EquationSpecification(
    coefficients = array(["beta01", "beta12", "beta03", "beta13" ]),
    variables = array(["constant", "household.persons", "constant", "household.persons" ]),
    equations = array([[1, 2, 3, 3 ]])
)
```

Each of the arguments is an array where the $i$th element describes the $i$th coefficient in an equation determined by the $i$th element in the argument equations. For example, beta12 is a coefficient connected to the household variable “persons” in equation 2, and beta03 is a constant used in equation 3. The prefix “household” in the variable name specifies the name of the dataset households. In other words, we are using here a dataset-qualified name of an attribute explained in Section 24.1.3. An optional argument submodels can extend the specification to a model with multiple submodels. The EquationSpecification class is described in Section 24.6.1 in more detail.

Our estimation data must include an attribute specifying choices that households made:

```python
>>> households.add_primary_attribute(data=[1,2,2,2,1,3,3,1,2,1], name="choice_id")
```

The estimation is done by passing the specification, the agent set for estimation and a name of the module that implements the actual estimation to the method estimate():

```python
>>> coefficients, other_results = choicemodel.estimate(
    specification, households,
    procedure="opus_core.bhhh_mnl_estimation")
```
Estimating Choice Model (from opus_core.choice_model): started on
Wed Nov 5 12:04:17 2008

submodel:  -2
Convergence achieved.
Akaike's Information Criterion (AIC):  26.142396805
Number of Iterations:  144

***********************************************
Log-likelihood is:    -9.07119840248
Null Log-likelihood is: -10.9861228867
Likelihood ratio index:  0.174303938155
Adj. likelihood ratio index: -0.189791752496
Number of observations:  10
Suggested |t-value| >   1.51742712939
Convergence statistic is:  0.000990037670084

-----------------------------------------------
Coeff_names     estimate    std err    t-values
beta01  0.432678      3.14438   0.137604
beta03  -4.51824      21.7087    -0.208131
beta12  0.180115      1.72541    0.10439
beta13  1.34052       4.18176    0.320564

***********************************************
Elapsed time:  0.064521 seconds
Estimating Choice Model (from opus_core.choice_model): completed.........0.1 sec

The estimation module given in the argument procedure is a child of EstimationProcedure (see Section 24.5.6), must contain a method run() and should return a dictionary with keys "estimators" and "standard_errors" respectively, that contain arrays of the estimated values and their standard errors, respectively. The estimate() method returns a tuple where the first element is an instance of class Coefficients and the second element is a dictionary with all results returned by the estimation procedure.

A coefficient object can be stored in a storage. For example, to store the computed coefficients as an ASCII file ‘mycoef.tab’ in the directory defined in the storage object on page 157, do

```python
>>> coefficients.write(out_storage=storage, out_table_name="mycoef")
```

Again, other types of storage can be used here.

Simulation

The coefficients that result from the estimation run can be directly plugged into a simulation run:
The resulting choices is an array specifying the choice for each household. We can now assign those values to the dataset:

```python
>>> households.modify_attribute(name="choice_id", data=choices)
```

Note that multiple runs will produce different results, which is due to random numbers used within the model. In order to receive reproducible results, one can fix the seed of the random number generator. The call above was preceded by

```python
>>> from numpy.random import seed
>>> seed(1)
```

For demonstration purposes, we use the same dataset of households for estimation and simulation. This would not be usually the case in real simulation runs.

We can also create a coefficient class and assign their values directly (see Section 24.6.2 for more details):

```python
>>> from opus_core.coefficients import Coefficients
>>> coefficients = Coefficients(
    names=array(["beta01", "beta12", "beta03", "beta13"]),
    values=array([0.5, 0.2, -5.0, 1.3]))
```

If a variable $x$ in the utility equations is choice dependent, one can create a dataset of choices and assign this attribute to the dataset. Besides a list and an array, the argument choice_set in the ChoiceModel constructor also accepts a dataset.

### 23.2.2 Location Choice Model

The LocationChoiceModel class implemented in urbansim is a special case of the ChoiceModel class (see Section 25.4.5 for details). The choice set is a set of locations that agents choose from. A set of locations can be sampled to each agent.

Suppose that in addition to the set of 10 households, we have a set of 9 locations with the attributes cost of living and capacity, respectively, stored in a file ‘locations.tab’:
One can represent this dataset using again the Dataset class:

```python
>>> locations = Dataset(in_storage = storage,
                        in_table_name = 'locations',
                        id_name='location',
                        dataset_name='gridcell')
```

We use here the storage object created on page 157, since the table is stored in the same directory.

Suppose, we wish to simulate a process of agents choosing locations. As a first example, suppose our only predictor is the location attribute cost with a coefficient value of $-0.01$ (modeling a negative effect of cost on the choice preferences). (For simplicity, we skip the estimation process and consider the coefficient value as given.) As in the previous section, we create a coefficient object, a specification object and a choice model object, respectively:

```python
>>> coefficients = Coefficients(names=("costcoef", ), values=(-0.01,))
>>> specification = EquationSpecification(variables=("gridcell.cost", ),
                                         coefficients=("costcoef", ))
>>> from urbansim.models.household_location_choice_model import 
   HouseholdLocationChoiceModel
>>> hlcm = HouseholdLocationChoiceModel(
    location_set = locations,
    sampler=None,
    compute_capacity_flag=False)
```

The household location choice model (HLCM) is a child of LocationChoiceModel with some useful default settings. The argument sampler specifies a module to be used for sampling locations to each agent. If it is None, all locations are considered as a possible alternative for each agent. The argument compute_capacity_flag specifies if the procedure should take capacity of locations into account.

We can run the HLCM by using the method run() which takes as obligatory arguments specification, coefficients and the set of agents for which the model runs.
Running Household Location Choice Model
(from urbansim.models.household_location_choice_model):
started on Wed Nov 5 12:53:11 2008

Total number of individuals: 10
HLCM chunk 1 out of 1.: started on Wed Nov 5 12:53:11 2008
Number of agents in this chunk: 10
HLCM chunk 1 out of 1.: completed.................................0.0 sec
Running Household Location Choice Model
(from urbansim.models.household_location_choice_model): completed...0.0 sec

The results of the HLCM run determine locations that agents have chosen. The model modifies values of the attribute ‘location’ of the agent set or adds this attribute to the dataset, if it doesn’t exist:

```python
>>> households.get_attribute("location")
array([5, 5, 5, 5, 5, 7, 7, 2, 5, 5])
```

One way of visualizing results of the HLCM run is to plot a histogram of the choices, including the capacity of each location (requires matplotlib library):

```python
>>> hlcm.plot_choice_histograms(capacity=locations.get_attribute("capacity"))
```

In our case, more agents decided for the fifth and seventh location than there are units available (which corresponds to the fact that those are two of the three cheapest locations).

In the above example, the discrete choice model consists of steps such as computing utilities via the `opus_core.linear_utilities` class, computing probabilities via the `opus_core.mnl_probabilities` class.
and computing choices via the opus_core.random_choices class. These components (default values) can be easily exchanged by other implementations. For example, the class urbansim.lottery_choices for computing choices takes into account capacity and forces agents to re-decide, if there is an overflow in the locations capacity:

```python
>>> number_of_agents = "gridcell.number_of_agents(household)"
>>> vacant_capacity = "capacity - gridcell.number_of_agents(household)"
>>> hlcm2 = HouseholdLocationChoiceModel(
    location_set = locations,
    sampler=None,
    choices="urbansim.lottery_choices",
    compute_capacity_flag=True,
    capacity_string=vacant_capacity,
    number_of_agents_string=number_of_agents,
    number_of_units_string="capacity",
    run_config={"lottery_max_iterations":10})
```

The argument choices is the full name of the module in which the choice class is implemented. This module must contain a class of the same name, i.e. lottery_choices in this case, be a child of opus_core class Choices (see Section 24.5.3) and have a method run(). Further arguments define an expression for computing vacant capacity, expression or variable that computes number of agents in each location (more about variable names in Section 24.3.1), name of the variable/attribute that determines the total number of units for each location. The argument run_config is a dictionary that contains parameters for the simulation. In this example it contains a value of how many times the households can re-decide if there is an overflow.

We run the above model:

```python
>>> seed(1)
>>> result = hlcm2.run(specification, coefficients, households)
Running Household Location Choice Model
(from urbansim.models.household_location_choice_model):
    started on Wed Nov 5 18:18:58 2008
Total number of individuals: 10
HLCM chunk 1 out of 1.: started on Wed Nov 5 18:18:58 2008
    Number of agents in this chunk: 10
    Available capacity: 15.0 units.
    Number of unplaced agents: 0 (in 4 iterations)
HLCM chunk 1 out of 1.: completed.................................0.0 sec
    gridcell.number_of_agents(household).................................0.0 sec
Running Household Location Choice Model
(from urbansim.models.household_location_choice_model): completed...0.0 sec
>>> hlcm2.plot_choice_histograms(capacity=locations.get_attribute("capacity"))
```
>>> households.get_attribute("location")
array([9, 3, 9, 5, 4, 3, 7, 2, 8, 1])

Now there is no overflow. Moreover, fourth and sixth locations still have free units available, since they are the most expensive places. The agents were ‘forced’ to re-decide three times (i.e. 4 iterations in total). If the maximum number of iterations given in \texttt{run\_config} would be reached without placing all agents, the whole model would run automatically again. It would not be very useful in this simple example, but the model is set-up for more complex situations, including sampling of locations. Therefore in a new run, agents would sample different locations which could solve the collision from the previous run.

Finally, agents that were not placed have values smaller equal zero in the “location” attribute.

\texttt{urbansim} implements several location choice models. Figure 25.1 in Section 25.4 shows a hierarchical structure of the models.

### 23.2.3 Regression Model

Opus offers an infrastructure for estimation and simulation of a regression model (see Section 24.4.4 for details). Suppose our set of grid cells from the previous section has an attribute “distance\_to\_cbd” that contains information about the distance to the central business district (cbd):

<table>
<thead>
<tr>
<th>location</th>
<th>cost</th>
<th>distance_to_cbd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>600</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>400</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>800</td>
<td>3</td>
</tr>
</tbody>
</table>
One can add the new attribute to the existing set of locations by:

```python
>>> locations.add_primary_attribute(name="distance_to_cbd",
   data=[5,10,5,1,20,0,7,7,3])
```

The cost of living in this dataset is highly correlated with the distance to cbd and we can thus predict the cost using the regression model.

**Initialization**

The model is initialized by:

```python
>>> from opus_core.regression_model import RegressionModel
>>> rm = RegressionModel(regression_procedure="opus_core.linear_regression")
```

As in the case of choice model, the model is composed by model components, here by plugging the appropriate regression procedure. The regression procedure is a child of Regression (see Section 24.5.5), must have a method `run()` that gets as arguments a multidimensional data array and a one-dimensional array of coefficients, and returns a one-dimensional array of the regression outcome.

The `linear_regression` module implemented in `opus_core` is the default procedure for the `RegressionModel` constructor and thus, it can be omitted.

**Estimation**

The specification for the above example consists of one variable and one constant for the intercept:

```python
>>> specification = EquationSpecification(
   variables=array(["constant", "gridcell.distance_to_cbd"]),
   coefficients=array(["constant", "dcbd_coef"])
```

The estimation for predicting the cost of living is run by:
Estimating Regression Model (from opus_core.regression_model):  
started on Mon Mar 19 21:14:33 2007

Estimate regression for submodel -2  
Number of observations: 9  
R-Squared: 0.536420010196  
Adjusted R-Squared: 0.470194297367  
Suggested |t-value| > 1.48230380737

-----------------------------------------------
<table>
<thead>
<tr>
<th>Coeff_names</th>
<th>estimate</th>
<th>SE</th>
<th>t-values</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>1114.07</td>
<td>213.758</td>
<td>5.21183</td>
</tr>
<tr>
<td>dcbd_coef</td>
<td>-71.1493</td>
<td>24.9995</td>
<td>-2.84603</td>
</tr>
</tbody>
</table>

-----------------------------------------------

Estimating Regression Model (from opus_core.regression_model): completed...0.4 sec

The estimation procedure that is passed as an argument is expected to be a child of `EstimationProcedure` (see Section 24.5.6) and have a method `run()` that takes as arguments a multidimensional data array and an instance of a class specified by the argument `regression_procedure` in the model constructor. Thus, the estimation procedure can use the same code that is used for simulation.

The resulting object of class `Coefficients` called `coef` can be stored or directly used for predicting cost of other locations.

```python
>>> coef.summary()
Coefficient object:
size: 2
names: ['constant', 'dcbd_coef']
values:
[1114.07348633, -71.14933777]
standard errors:
[213.75848389, 24.99952126]
t_statistic:
[5.211833, -2.846028]
submodels: [-2 -2]
```

Simulation

Suppose we have four locations with distance to cbd 2, 4, 6 and 8 respectively. We create a location dataset using a storage type `dict`. This type of storage is useful when we want to pass data directly without storing them on a physical storage media.
>>> dstorage = StorageFactory().get_storage('dict_storage')
>>> dstorage.write_table(table_name='gridcells',
    table_data= {'id':array([1,2,3,4]),
                 'distance_to_cbd':array([2,4,6,8])})
>>> ds = Dataset(in_storage=dstorage, in_table_name='gridcells',
               id_name='id', dataset_name='gridcell')

We have created a table in the RAM space called 'gridcells' which is passed into the Dataset constructor. Now we run the regression model with coefficients estimated in the previous section:

>>> cost = rm.run(specification, coefficients=coef, dataset=ds)
Running Regression Model (from opus_core.regression_model):
    started on Mon Mar 19 21:35:23 2007
    Total number of individuals: 4
    RM chunk 1 out of 1.: started on Mon Mar 19 21:35:23 2007
    Number of agents in this chunk: 4
    RM chunk 1 out of 1.: completed......................................0.0 sec
Running Regression Model (from opus_core.regression_model): completed....0.0 sec
>>> cost
array([ 971.77478027, 829.47613525, 687.17749023, 544.87878418])

As expected, the resulting cost decreases with increasing distance to cbd.

23.3 Opus Variables

23.3.1 Variable Concept

As mentioned in Section 23.1, a dataset has a set of attributes, such as income or persons, that are stored in a file or database. We call such characteristics primary attributes. In addition, one is usually interested in attributes that are computed, for example using some transformation of primary attributes. We call those attributes variables, or computed attributes. They are simply handled as additional “columns” of a dataset to which they belong to, here denoted as “parent dataset”.

In Opus, a variable is a class derived from the opus_core class Variable. Opus expressions (Chapter 13), as used in the expression library in the GUI and elsewhere, in fact just are compiled into an automatically generated subclass of class Variable. Section 24.3 gives additional details about this class, including a discussion of how expressions are compiled.

In the GUI description, we used the expression library as a place to store and reuse variable definitions. We can also write expressions as Python strings and use them from the command line. The syntax of these expression language is that of Python, but the semantics are somewhat different — for example, a name like gridcell.distance_to_cbd is a reference to the value of that variable. (If you just evaluated this in a Python shell you’d get an error, saying that the gridcell package didn’t have a distance_to_cbd attribute.) Further, expressions are evaluated lazily. Here is a simple example:
The expressions are fed to the `compute_variables` method of Dataset (Section 23.3.1), just as for simple variable references. (Given a new expression, Opus will compose a new variable definition, including a `compute` and a `dependencies` method, which computes the value of that expression. It then saves the new variable definition in case the same expression is encountered again — see Section 24.3.4. However, usually the user need not be concerned about this.) The `compute_variables` method evaluates each variable (or expression) in turn, and returns the value of the last one.

For variables defined by hand as Python classes (rather than using Tekoa expressions), the name of the class for a given variable is identical to the name of the module in which it is implemented. The module is stored in a directory whose name corresponds to the name of the parent dataset. Note that the variable name must be all lower case.

The variable class must have a method `compute()` that returns a numpy array of variable values. The size of that array must correspond to the number of entries in the parent dataset. The `compute()` method takes an argument called `dataset_pool` containing references to the appropriate set of datasets to use for computing this variable. From the `compute()` method, the parent dataset can be accessed by `self.get_dataset()`.

If the variable depends on other attributes, they must be listed in the method `dependencies()`, which returns a list of all dependent variables and attributes in their fully-qualified names (see Section 24.1.3 for details on attribute specification).

As an example, consider a variable “is_in_wetland” for the gridcell dataset `locations` from Sections 23.2.2 and 23.2.3. The variable returns `True` for entries whose percentage of wetland is more than a certain threshold, and `False` otherwise. The module 'is_in_wetland.py', containing a class `is_in_wetland`, is stored in the directory ‘gridcell’ because

```python
>>> locations.get_dataset_name()
'gridcell'
```

The class is defined as follows:

```python
from opus_core.variables.variable import Variable
class is_in_wetland(Variable):
    def dependencies(self):
        return ["gridcell.percent_wetland"]
    def compute(self, dataset_pool):
        return self.get_dataset().get_attribute("percent_wetland") > \
            dataset_pool.get_dataset('urbansim_constant')['percent_coverage_threshold']
```

The dependent attribute is a primary attribute and therefore specified as a dataset-qualified name. For our example, we populate the primary attribute:

```python
>>> locations.add_primary_attribute(name="percent_wetland",
        data=[85,20,0,90,35,51,0,10,5])
```
As another example, consider the population variable to compute the population in each gridcell. We refer to a variable using a Python path, which provides a means for Python to find modules and classes in a directory structure. So, a reference to this particular module using a fully qualified path would be urbansim.gridcell.population. You can find all of the existing variables by browsing on disk in the source code directory. The directory structure mirrors the parts of the variable name, so that urbansim.gridcell.population is really pointing to ‘opus/src/urbansim/gridcell/population.py’, which is also included in Figure 23.1 on page 193.

Dataset Pool

When computing variable values, Opus may need to access datasets used by dependent variables, such as the urbansim_constant dataset required by is_in_wetland, above. These datasets are provided by the dataset_pool object passed into the variable’s compute method. This object is an instance of the DatasetPool class (see Section 24.7.1). It is responsible for keeping a set of datasets for use when computing variables. If the requested dataset is in the pool already, it is returned directly, otherwise it tries to find it in the given storage.

In order to find the appropriate dataset class for the requested dataset, the dataset pool object searches the ‘datasets’ directories of the Opus packages in package_order. The first one to contain the appropriately named module and class is used. For instance, using the storage object created on page 157, for the definition:

```python
>>> from opus_core.dataset_pool import DatasetPool
>>> dataset_pool = DatasetPool(package_order=['urbansim', 'opus_core'], storage=storage)
```

if dataset_pool is asked for the “household” dataset, and does not already have one in its pool, it will look in the storage. In order to create the corresponding dataset classes it searches for a file ‘household_dataset.py’ (containing the class HouseholdDataset) first in ‘urbansim/datasets’ and then in ‘opus_core/datasets’. The first one found will be used.

```python
>>> dataset_pool.datasets_in_pool()
{}
>>> hs = dataset_pool.get_dataset("household")
>>> dataset_pool.datasets_in_pool()
{'household': <urbansim.datasets.household_dataset.HouseholdDataset object at 0x197c2f70>}
>>> hs.size()
10
```

If the dataset pool is asked for urbansim_constant and ‘urbansim’ is included in package_order, it will use some urbansim specific constants defined in ‘urbansim/constants.py’ (in addition to user defined constants found on storage).

For the purpose of this example, we set the required constant by:

```python
>>> constant = dataset_pool.get_dataset("urbansim_constant")
>>> constant["percent_coverage_threshold"] = 50
```
Invoking Computation

Computation of a variable is invoked using the Dataset method compute_variables() passing the variable name in its fully-qualified form and a dataset pool if other datasets are required for this variable:

```python
>>> locations.compute_variables("urbansim.gridcell.is_in_wetland", dataset_pool=dataset_pool)
urbansim.gridcell.is_in_wetland..........................................0.0 sec
array([ True, False, False,  True, False,  True, False, False, False], dtype=bool)
```

The method returns the resulting values of the variable. The compute_variables method also accepts a list of variables to be computed. In this case it returns values of the last variable in the list.

In addition, within the dataset, the variable can be accessed using its un-qualified name. Thus, the un-qualified names of variables including the primary attributes must be unique.

```python
>>> locations.get_attribute("is_in_wetland")
array([ True, False, False,  True, False,  True, False, False, False], dtype=bool)
```

### 23.3.2 Interaction Variables

In order to work with variables that determine an interaction between two datasets, there is a subclass of Dataset, called InteractionDataset. Attributes of this class are stored as two-dimensional arrays (see Section 24.1.8 for details).

To create an interaction set for households and gridcells from the previous sections, do

```python
>>> from opus_core.datasets.interaction_dataset import InteractionDataset
>>> interactions = InteractionDataset(dataset1 = households, dataset2 = locations)
```

The dataset name is composed from dataset names of the two interacting datasets:

```python
>>> interactions.get_dataset_name()
'household_x_gridcell'
```

Thus, an interaction variable for such dataset will be implemented in the directory ‘household_x_gridcell’.

The InteractionDataset class contains several methods that are useful for variable computation which must return a two-dimensional array. For example, a variable defined as a multiplication of cost and income can be implemented as:
from opus_core.variables.variable import Variable

class cost_times_income(Variable):
    def dependencies(self):
        return ["gridcell.cost", "household.income"]

    def compute(self, dataset_pool):
        return self.get_dataset().multiply("income", "cost")

This cost_times_income variable is mostly for illustration; it can also be defined more conveniently as an expression (see Chapter 13).

If we wish that only a subset of each dataset interact (for example for memory reasons), we can pass the corresponding indices to the constructor:

>>> from numpy import arange
>>> interactions = InteractionDataset(dataset1 = households, dataset2 = locations,
index1 = arange(5), index2 = arange(3))

Here only the first 5 households and the first three gridcells interact. The compute() method of the interaction variable should return an array of size index1 × index2.

>>> interactions.compute_variables(["urbansim.household_x_gridcell.cost_times_income"])
array([[ 500000., 200000., 600000.],
[ 1000000., 400000., 1200000.],
[ 2500000., 1000000., 3000000.],
[ 1500000., 600000., 1800000.],
[ 250000., 100000., 300000.]]

urbansim uses interaction variables mainly in ChoiceModel classes, where agents interact with dataset of choices. The household location choice model object hlcm from Section 23.2.2 can be for example estimated using two variables, one of which is an interaction variable:
>>> specification = EquationSpecification(
    variables=Array(["gridcell.cost",
                     "urbansim.household_x_gridcell.cost_times_income"]),
    coefficients=Array(["costcoef", "cti_coef"]))

>>> # place households
>>> households.add_primary_attribute(data=[2,8,3,1,5,4,9,7,3,6], name="location")
>>> coef, other_results = hlcm.estimate(specification, households)

Estimating Household Location Choice Model
(from urbansim.models.household_location_choice_model):
    started on Thu Nov 6 12:12:47 2008
urbansim.household_x_gridcell.cost_times_income......................0.0 sec
submodel: -2
Convergence achieved.
Akaike's Information Criterion (AIC): 39.8199022026
Number of Iterations: 18
*****************************************************************************
Log-likelihood is:  -17.9099511013
Null Log-likelihood is:  -21.9722457734
Likelihood ratio index:  0.184882998032
Adj. likelihood ratio index:  0.0938590753688
Number of observations:  10
Suggested |t-value| >  1.51742712939
Convergence statistic is:  0.000302603242051
*****************************************************************************
Coeff_names estimate    std err    t-values
  costcoef  -0.00312452  0.00339464  -0.920429
  cti_coef   4.45803e-07  5.34429e-07   0.834168
*****************************************************************************
Elapsed time:  0.02 seconds
Estimating Household Location Choice Model
(from urbansim.models.household_location_choice_model): completed...0.0 sec

23.3.3 Versioning

Opus uses a mechanism of versioning of variables and attributes. Each variable/attribute is initialized with a version number 0. Any subsequent change of that variable/attribute increments the version number. This allow us to only recompute variables, if it is really needed. This means that if a computation is invoked (by e.g. compute_variables()), it is checked, if any of the version numbers of all dependent variables has changed since the last computation. The computation is performed only, if there was any change in the dependencies tree. The mechanism is described in Section 24.3.3 in more detail.

For example,
The first call of `compute_variables()` didn’t perform the actual computation, because nothing has changed since our previous computation on page 175. After we modify one element of the “income” attribute (note that “income” is a dependent variable of “cost_times_income” defined in the `dependencies()` method), the variable is recomputed and the version number is increased.

### 23.3.4 Using Arguments in Variable Names

Opus offers the possibility of including a number or a character string into variable name which is then passed to the variable constructor as an argument. The module/class name of such variable corresponds to a pattern in which a number is replaced by ‘DDD’ and a string is replaced by ‘SSS’. For example, a similar variable to `is_in_wetland` in Section 23.3.1 can be implemented in a way that the threshold is directly included in the variable name, such as `is_in_wetland_if_threshold_is_80`. Here, we give an example of variable of the type `is_near_location_if_threshold_is_number`. As `location` we can use e.g. ‘highway’, ‘arterial’, or ‘cbd’. The class name is `is_near_SSS_if_threshold_is_DDD` and must have a constructor implemented which takes a string and a number as arguments:

```python
from opus_core.variables.variable import Variable

class is_near_SSS_if_threshold_is_DDD(Variable):
    def __init__(self, location, number):
        self.location = location
        self.number = number
        Variable.__init__(self)

    def dependencies(self):
        return ['gridcell.distance_to_' + self.location]

    def compute(self, dataset_pool):
        distance_to_location = self.get_dataset().get_attribute(
            'distance_to_' + self.location)
        return distance_to_location < self.number
```

We can then invoke the computation for the ‘cbd’ location and different thresholds by changing the variable name:
>>> res = locations.compute_variables(
    map(lambda threshold:
        'urbansim.gridcell.is_near_cbd_if_threshold_is_%s' % threshold, [2, 4, 7]))
urbansim.gridcell.is_near_SSS_if_threshold_is_DDD...................0.0 sec
urbansim.gridcell.is_near_SSS_if_threshold_is_DDD...................0.0 sec
urbansim.gridcell.is_near_SSS_if_threshold_is_DDD...................0.0 sec

>>> locations.get_attribute("is_near_cbd_if_threshold_is_2")
array([False, False, False, True, False, True, False, False, False], dtype=bool)
>>> locations.get_attribute("is_near_cbd_if_threshold_is_4")
array([False, False, False, True, False, True, False, False, True], dtype=bool)
>>> locations.get_attribute("is_near_cbd_if_threshold_is_7")
array([ True, False, True, True, False, True, False, False, True], dtype=bool)

If the dataset `locations` would have an attribute “distance_to_highway”, we could use the same variable implementation for variables `is_near_highway_if_threshold_is_...`

The arguments of the constructor are passed in the same number and order as they appear in the variable name. For example, if the name would contain only one pattern, say DDD, the constructor would expect only one argument, namely an integer. The method `dependencies()` is called from `Variable.__init__()`, therefore any class attributes used in `dependencies()` (such as `self.location` in this case) must be set before the call of `Variable.__init__()`.  

```
Note: Arguments for variable names will probably be replaced with a more standard syntax in the future, for example `is_near(feature='highway', threshold=2)`. However, we don’t have a specific date yet for this change. We will try to preserve backward compatibility.
```

### 23.3.5 Expression Names and Aliasing

The expression library includes a name field for each expression. This is used to bind the name to the corresponding expression. When using such expressions intermixed with Python code, another mechanism is needed to give a name to an expression – this is done with something much like an assignment statement, for example:

```
lnpop = ln(urbansim.gridcell.population)
```

This is treated as an expression, which can occur in the list of expressions given to `compute_variables` or in an ‘aliases.py’ file (see below). The value of the new alias is returned as the value of the expression if it’s the last item on the list of variables and expressions.

It is convenient, and often more efficient (Section 13.11), to gather all the expressions and aliases for a particular package and dataset into one place. When using XML-based configurations, the expression library component of the XML configurations supports this nicely. In older parts of the code using dictionary-based configurations, the optional ‘aliases.py’ file supports this functionality instead. This file should define a single variable `aliases` to be a list of expressions, each of which should define an alias. This file is then placed in the same directory as variables for that package and dataset. For example, to define aliases relevant to `urbansim.gridcell`, put an ‘aliases.py’ file into the ‘urbansim.gridcell’ directory. These aliases can then be referred to using the fully-qualified name of the alias. When finding a variable referenced by a fully-qualified name, the system first searches the aliases file (if present), and then the variable definitions in the appropriate directory.

As an example, the directory `opus_core.test_agent` contains one variable definition, for the variable
income_times_2. (This directory and variable are used for unit tests for opus_core.) The file ‘aliases.py’ in that same directory contains the following:

```python
aliases = [
    'income_times_5 = 5*opus_core.test_agent.income',
    'income_times_10 = 5*opus_core.test_agent.income_times_2'
]
```

The first alias refers to a primary attribute (test_agent.income), and the second to the variable. These aliases can then be referred to using a fully-qualified name, in exactly the same way as a variable, for example

```python
opus_core.test_agent.income_times_5.
```

See the unit tests in ‘opus_core.variables.expression_tests.aliases_file’ for examples of using these aliases.

### 23.3.6 Using Interaction Sets in Expressions

If you access an attribute of a component of an interaction set in the context of that interaction set, the result is converted into a 2-d array and returned. These 2-d arrays can then be multiplied, divided, compared, and so forth, using the numpy functions and operators. For example, suppose we have an interaction set ‘household_x_gridcell’. The component ‘household’ set has an attribute `income` with values [100, 200, 300]. (These numbers are just to explain the concepts — obviously they aren’t realistic incomes.) The ‘gridcell’ component has an attribute `cost` with values [1000, 1200]. Then evaluating

```python
household_x_gridcell.compute_variables('urbansim.household.income')
```

will return a 2-d array

```plaintext
[[100, 100],
 [200, 200],
 [300, 300]]
```

and

```python
household_x_gridcell.compute_variables('urbansim.gridcell.cost')
```

returns

```plaintext
[[1000, 1200],
 [1000, 1200],
 [1000, 1200]]
```

Then

```python
household_x_gridcell.compute_variables(urbansim.gridcell.cost*urbansim.household.income')
```
evaluates to

$$\begin{bmatrix}
[100000, 120000], \\
[200000, 240000], \\
[300000, 360000]
\end{bmatrix}$$

Both the arguments to the operation and the result can be used in more complex expressions. For example, if we wanted to give everyone a $5000 income boost, and also scale the result, this could be done using

$$(\text{household.income} + 5000) \times \text{gridcell.cost} \times 1.2.$$ 

As another example, the model specification from page 176 can be modified by using an expression for the interaction and taking its log:

```python
>>> specification = EquationSpecification(
    variables=("gridcell.cost", 
              "ln(urbansim.gridcell.cost*urbansim.household.income)"), 
    coefficients=("costcoef", "cti_coef"))
```

### 23.3.7 Using Aggregation and Disaggregation in Expressions

The methods `aggregate` and `disaggregate` are used to aggregate and disaggregate variable values over two or more datasets. The `aggregate` method associates information from one dataset to another along a many-to-one relationship, while the `disaggregate` method does the same along a one-to-many relationship. Some examples are:

- `zone.aggregate(gridcell.population)`
- `zone.aggregate(2.5*gridcell.population)`
- `zone.aggregate(urbansim.gridcell.population)`
- `neighborhood.aggregate(gridcell.population, intermediates=[zone, faz])`
- `neighborhood.aggregate(gridcell.population, intermediates=[zone, faz], function=mean)`
- `zone.aggregate(gridcell.population, function=mean)`
- `region.aggregate_all(zone.my_variable)`
- `region.aggregate_all(zone.my_variable, function=mean)`
- `faz.disaggregate(neighborhood.population)`
- `gridcell.disaggregate(neighborhood.population, intermediates=[zone, faz])`

The syntax and semantics for these is as follows.
Aggregation
Suppose we have three different geographical units: gridcells, zones and neighborhoods. We have information available on the gridcell level and would like to aggregate this information for zones and neighborhoods. We know the
assignments of gridcells to zones and of zones to neighborhoods.
First, we place the data for three neighborhoods and five zones into a dict storage:

>>> dstorage = StorageFactory().get_storage('dict_storage')
>>> dstorage.write_table(table_name='neighborhoods',
table_data={"nbh_id":array([1,2,3])}
)
>>> dstorage.write_table(table_name='zones',
table_data={"zone_id":array([1,2,3,4,5]),
"nbh_id": array([3,3,1,2,1])}
)

Then, we create the corresponding datasets:

>>> neighborhoods = Dataset(in_storage=dstorage, in_table_name='neighborhoods',
dataset_name="neighborhood", id_name="nbh_id")
>>> zones = Dataset(in_storage=dstorage, in_table_name='zones',
dataset_name="zone", id_name="zone_id")

Note that zones contain assignments to neighborhoods in the attribute ‘nbh_id’. For the gridcell set, consider the
dataset locations defined on page 165. We add assignments of those nine locations to the zones:

>>> locations.add_primary_attribute(name="zone_id", data=[3,5,2,2,1,1,3,5,3])

Note that any assignment must be done by using an attribute of the same name as the unique identifier of the dataset
that the assignment is made to.
As the next step, we prepare a dataset pool for the variable computation, since we are dealing with variables that
involve more than one dataset. To make things easy, we explicitly insert our three datasets into the pool:

>>> dataset_pool = DatasetPool(package_order=['urbansim', 'opus_core'],
datasets_dict={'gridcell': locations,
'zone': zones,
'neighborhood':neighborhoods})

An aggregation over one geographical level for the locations attribute ‘capacity’ can be done by:

>>> aggr_var = "aggregated_capacity = zone.aggregate(gridcell.capacity)"
>>> zones.compute_variables(aggr_var, dataset_pool=dataset_pool)
aggregated_capacity = zone.aggregate(gridcell.capacity)..................0.0 sec
array([ 4., 5., 4., 0., 2.])

By default, the aggregation function applied to the aggregated data is the ‘sum’ function. This can be changed by
passing the desired function as second argument in the variable name:
23.3. Opus Variables

181


The `aggregate` method accepts the following aggregation functions: sum, mean, variance, standard deviation, minimum, maximum, center_of_mass. These are functions of the scipy package `ndimage`.

An aggregation over two or more levels of geography is done by passing a third argument into the `aggregate` method. It is a list of dataset names over which it is aggregated, excluding datasets for the lowest and highest level. For example, aggregating the gridcell attribute ‘capacity’ for the neighborhood set can be done by:

```python
>>> aggr_var2 = 
    "neighborhood.aggregate(gridcell.capacity, function=sum, intermediates=[zone])"
>>> neighborhoods.compute_variables(aggr_var2, dataset_pool=dataset_pool)
neighborhood.aggregate(gridcell.capacity, function=sum, intermediates=[zone]):
    completed...0.3 sec
array([ 6., 0., 9.])
```

Disaggregation

Disaggregation is done analogously. The `disaggregate` method takes information from a coarse set of entities and allocates it to a finer set of entities, in the manner of a one-to-many relationship. By default, the function for allocating data is to simply replicate the data on the parent entity for each inheriting entity. The method takes one required argument, an attribute/variable name, and one optional argument, a list of dataset names. Here we add an attribute “is_cbd” to the neighborhood set and distribute it across gridcells:

```python
>>> neighborhoods.add_primary_attribute(name="is_cbd", data=[0,0,1])
>>> disaggr_var = 
    "is_cbd = gridcell.disaggregate(neighborhood.is_cbd, intermediates=[zone])"
>>> locations.compute_variables(disaggr_var, dataset_pool=dataset_pool)
is_cbd = gridcell.disaggregate(neighborhood.is_cbd, intermediates=[zone]):
    completed...0.0 sec
array([[0, 0, 1, 1, 1, 1, 0, 0, 0]])
```

Note that since we used the dataset-qualified name for “is_cbd” in the `disaggregate` method, the attribute must be a primary attribute of `neighborhoods`. The `aggregate` and `disaggregate` methods both must have the dataset name of the dataset for which they are computed before the method name, e.g. `gridcell.disaggregate`.

To aggregate over all members of one dataset, one can use the built-in method `aggregate_all`. It must be used with a dataset that has one element which is the case of the `opus_core` dataset `AlldataDataset` implemented in
the directory ‘datasets’. For example, the total capacity for all gridcells can be determined by:

```python
>>> from opus_core.datasets.alldata_dataset import AlldataDataset
>>> alldata = AlldataDataset()
>>> alldata.compute_variables(
    "total_capacity = alldata.aggregate_all(gridcell.capacity, function=sum)",
    dataset_pool=dataset_pool)
```

```
total_capacity = alldata.aggregate_all(gridcell.capacity, function=sum)....0.0 sec
array([ 15.])
```

In addition to `sum`, the `aggregate_all` class accepts all functions that are accepted by the `aggregate` class; the default is `sum`.

If the attribute being aggregated or disaggregated is a simple variable, it should be either dataset-qualified or fully-qualified, i.e. always including the dataset name and optionally including the package name. The attribute being aggregated can also be an expression. (In this case, behind the scenes the system generates a new variable for that expression, and then uses the new variable in the aggregation or disaggregation operations. However, this isn’t visible to the user.) The result of an aggregation or disaggregation can also be used in more complex expressions, e.g. `ln(2*aggregate(gridcell.population))`.

### 23.3.8 Number of Agents

A common task in modeling is to determine a number of agents of one dataset that are assigned to another dataset. For this purpose, Opus contains a built-in method `number_of_agents`, which takes as an argument the name of the agent dataset. For example, our household dataset is assigned to the following locations:

```python
>>> households.modify_attribute(name="location",
   data=[2, 8, 3, 1, 5, 4, 9, 7, 3, 6])
```

Then, the number of households in each location can be determined by:

```python
>>> dataset_pool.add_datasets_if_not_included({'household': households})
>>> locations.compute_variables("gridcell.number_of_agents(household)",
   dataset_pool=dataset_pool)
gridcell.number_of_agents(household).....................0.0 sec
array([ 1., 1., 2., 1., 1., 1., 1., 1., 1., 1.])
```

Note that we had to add the household dataset to the dataset pool in order to have it available in the computation process.

Similarly, the number of zones in neighborhoods is computed by

```python
>>> neighborhoods.compute_variables("neighborhood.number_of_agents(zone)",
   dataset_pool=dataset_pool)
neighborhood.number_of_agents(zone).....................0.0 sec
array([ 2., 1., 2.])
```

As in the case of `aggregate` and `disaggregate`, the `number_of_agents` method must be preceded by the ‘owner’ dataset name, e.g. `neighborhood.number_of_agents` for computing on the neighborhood dataset.
23.4 Creating a New Model

In most cases, a model would perform some operations on datasets. Opus’ only requirements for model classes are:

1. Being a child class of the `opus_core.Model`, and
2. have a method `run()`.

Optionally, it can have a class attribute `model_name`.

Thus, the following code is a model:

```python
>>> from opus_core.model import Model
>>> from opus_core.logger import logger
>>> class MyModel(Model):
...     model_name = "my model"
...     def run(self):
...         logger.log_status("I'm running!")
...         return

Then

```python
>>> MyModel().run()
Running my model (from __main__): started on Tue Mar 28 17:41:04 2006
I'm running!
Running my model (from __main__): completed..............................0.0 sec
```

Packages `opus_core` and `urbansim` implement several models that can be used as parent classes when developing a new model. The whole model hierarchy is shown in Figure 25.1 in Section 25.4. We give here an example of implementing a new chunk model, making use of the `opus_core.Class ChunkModel` (see Section 24.4.2) which automatically processes the model in several chunks.

Suppose we wish to generate a certain number \( n \) of normally distributed random numbers to each agent of a dataset. The mean and variance of the distributions are agent’s specific and are given by two existing attributes of the dataset. The model returns an array of the means of the generated numbers for each dataset member. The number of the generated values \( n \) is user defined and thus it will be passed as an argument. Since we expect that both \( n \) and the dataset size can be large, we choose the `ChunkModel` as the parent class which provides flexibility in saving memory. For this model, we only need to define the method `run_chunk()` containing the actual computation, since the `run()` method is defined in the parent class (see Section 24.4.2 for its arguments).

The model can be coded as follows:
>>> from opus_core.chunk_model import ChunkModel
>>> from numpy import apply_along_axis
>>> from numpy.random import normal

>>> class MyChunkModel(ChunkModel):
    model_name = "my chunk model"
    def run_chunk(self, index, dataset, mean_attribute, variance_attribute, n=1):
        mean_values = dataset.get_attribute_by_index(mean_attribute, index)
        variance_values = dataset.get_attribute_by_index(variance_attribute, index)
        def draw_rn(mean_var, n):
            return normal(mean_var[0], mean_var[1], size=n)
        normal_values = apply_along_axis(draw_rn, 0, (mean_values, variance_values), n)
        return normal_values.mean(axis=0)

The first two arguments of run_chunk() are obligatory (determined by the parent class), the remaining ones are application specific. index is an index of members of dataset that are processed in that chunk. The parent class takes care of “chopping” the dataset into appropriate chunks. mean_attribute and variance_attribute are names of the dataset attributes that determine the means and variances, respectively. The model extracts the means and variances for dataset members of this chunk, generates a matrix of normally distributed random numbers of size \(n \times \text{number of agents in the chunk}\), and returns the means for each agent.

In order to use this model, we need to create a dataset with the two required attributes for means and variances. In our case, we have a dataset of 100,000 entries. The mean and variance for the first half of the entries is 0 and 1, respectively. The mean and variance for the second half of the entries is 10 and 5, respectively.

```python
>>> from numpy import arange, array
>>> from opus_core.storage_factory import StorageFactory
>>> storage = StorageFactory().get_storage('dict_storage')
>>> storage.write_table(table_name='dataset',
    table_data={'id':arange(100000)+1,
               'means':array(50000*[0]+50000*[10]),
               'variances':array(50000*[1]+50000*[5])}
)
>>> from opus_core.datasets.dataset import Dataset
>>> mydataset = Dataset(in_storage=storage, in_table_name='dataset',
    id_name='id', dataset_name='mydataset')
```

Invoking a run of this model in five chunks is done by

```python
>>> from opus_core.chunk_model import ChunkModel
>>> from numpy import apply_along_axis
>>> from numpy.random import normal

>>> class MyChunkModel(ChunkModel):
    model_name = "my chunk model"
    def run_chunk(self, index, dataset, mean_attribute, variance_attribute, n=1):
        mean_values = dataset.get_attribute_by_index(mean_attribute, index)
        variance_values = dataset.get_attribute_by_index(variance_attribute, index)
        def draw_rn(mean_var, n):
            return normal(mean_var[0], mean_var[1], size=n)
        normal_values = apply_along_axis(draw_rn, 0, (mean_values, variance_values), n)
        return normal_values.mean(axis=0)

The first two arguments of run_chunk() are obligatory (determined by the parent class), the remaining ones are application specific. index is an index of members of dataset that are processed in that chunk. The parent class takes care of “chopping” the dataset into appropriate chunks. mean_attribute and variance_attribute are names of the dataset attributes that determine the means and variances, respectively. The model extracts the means and variances for dataset members of this chunk, generates a matrix of normally distributed random numbers of size \(n \times \text{number of agents in the chunk}\), and returns the means for each agent.

In order to use this model, we need to create a dataset with the two required attributes for means and variances. In our case, we have a dataset of 100,000 entries. The mean and variance for the first half of the entries is 0 and 1, respectively. The mean and variance for the second half of the entries is 10 and 5, respectively.

```python
>>> from numpy import arange, array
>>> from opus_core.storage_factory import StorageFactory
>>> storage = StorageFactory().get_storage('dict_storage')
>>> storage.write_table(table_name='dataset',
    table_data={'id':arange(100000)+1,
               'means':array(50000*[0]+50000*[10]),
               'variances':array(50000*[1]+50000*[5])}
)
>>> from opus_core.datasets.dataset import Dataset
>>> mydataset = Dataset(in_storage=storage, in_table_name='dataset',
    id_name='id', dataset_name='mydataset')
```

Invoking a run of this model in five chunks is done by
>>> from numpy.random import seed
>>> seed(1)
>>> results = MyChunkModel().run(chunk_specification={'nchunks':5},
dataset=mydataset,
    mean_attribute="means",
    variance_attribute="variances",
    n=10)

Running my chunk model (from __main__): started on Wed Mar 21 12:00:53 2007
Total number of individuals: 100000
ChunkM chunk 1 out of 5.: started on Wed Mar 21 12:00:53 2007
    Number of agents in this chunk: 20000
ChunkM chunk 1 out of 5.: completed.................................0.7 sec
ChunkM chunk 2 out of 5.: started on Wed Mar 21 12:00:54 2007
    Number of agents in this chunk: 20000
ChunkM chunk 2 out of 5.: completed.................................0.7 sec
ChunkM chunk 3 out of 5.: started on Wed Mar 21 12:00:54 2007
    Number of agents in this chunk: 20000
ChunkM chunk 3 out of 5.: completed.................................0.7 sec
ChunkM chunk 4 out of 5.: started on Wed Mar 21 12:00:55 2007
    Number of agents in this chunk: 20000
ChunkM chunk 4 out of 5.: completed.................................0.7 sec
ChunkM chunk 5 out of 5.: started on Wed Mar 21 12:00:56 2007
    Number of agents in this chunk: 20000
ChunkM chunk 5 out of 5.: completed.................................0.7 sec
Running my chunk model (from __main__): completed....................3.7 sec

The run() method expects the first two arguments, the remaining ones are optional from the parent point of view.
The first argument specifies the number of chunks (see Section 24.4.2). By playing with different values for nchunks
and n one can see how quickly one can run out of memory.

Check the results, e.g. by checking the means of the two halves:

```python
>>> results[0:50000].mean()
0.0010989375305175781
>>> results[50000:].mean()
10.000937499999999
```

## 23.5 Troubleshooting Python

If you are unfamiliar with Python, here are some guidelines.

- In Python, (, {, and [ each have different meanings. Be careful to use the correct type of “parentheses”.
- Be careful about whether the name has a single underscore, e.g. __one__, versus two leading underscores, e.g. __two__.
  These can look similar in our documentation, so look carefully.
- In Python, words that have two double-underscores before and after the word, e.g. __init__ or __path__,
  generally denote “special” symbols.
- A command can be split over multiple lines. Normally, the Python continuation symbol ‘\’ at the end of each
  non-finished line is required. There is one exception: expressions in parentheses, straight brackets, or curly
  braces do not need ‘\’.
• Indentation matters in Python. Code blocks (except with expressions in parentheses, straight brackets, or curly braces) are defined by their indentation. We recommend only using spaces and not tabs, since combining them can be quite confusing if different systems display the tabs using different numbers of spaces.

23.6 Interactive Estimation and Diagnostics

Please note: the seattle_parcel sample data used in the following description is only to demonstrate functionality. One should not read too much into the results, since seattle_parcel is a small subset taken from the PSRC project datasets to make its size more manageable. In particular, estimating models with certain specifications may give peculiar results, such as SE and t-values equal to -1.#IND or -1.#INF due to an insufficient number of observations, collinearity, or outliers in the data. Estimation results that include erroneous results such as these cannot saved or exported to a sql database, or viewed in the OPUS GUI.

23.6.1 Estimation of a Regression Model

In this section, we take up the topic of estimating and diagnosing models interactively, from the command line. Eventually the functions described here will be available in the Graphical User Interface, but as of now they are only available from the interactive command mode.

Assume we want to estimate the real estate price model in the seattle_parcel project, which includes submodels to separately specify the model for each general land use type. If you are using a computer with the Windows operating system, open a command shell, and type the following:

```
python -i c:\opus\src\urbansim\tools\start_estimation.py
-x c:\opus\project_configs\seattle_parcel_default.xml -m real_estate_price_model
```

(Put this command in one single line or with proper continuation mark for respective operation system.)

Note that this assumes that your Opus packages are installed in the directory c:\opus. Also note that the argument of the -x option must be given as an absolute path.

Depending on exactly what the specification is, the result of the call would look something like the listing below:
Since the model was estimated in interactive mode, using the -i option in the python command to start the estimation, the program remains active after estimation is completed, and additional commands may be directly entered at the python prompt: >>>. Assume that we want to further explore the data in submodel 30 (mixed-use properties).

One of the first things one might wish to do is to examine the correlation among the variables in a model. We can do this by using one of the built-in estimator methods, plot_correlation, with the following command:

```python
>>> estimator.plot_correlation(30)
```

The method computes a correlation matrix for the data used in submodel 30 and generates a plot of this correlation, as shown in Figure 23.2:

Note that when a plot is generated from the command line in Python, control of Python is focused on the graphics window, and there is no Python prompt available. When finished viewing the figure, exit the graphics window by clicking on the x to close it, and the Python prompt will return for more interactive commands.

We can retrieve the data for submodel 30 as a dataset that we can further analyze using Opus methods for the Dataset class. Begin with the following command to retrieve the data and assign it to an object called ds30 (for submodel 30):
The syntax above indicates that we are executing a method called get_data_as_dataset of the class estimator, which is the class that is running the estimation of the model. The value 30 in parentheses is an argument being passed to this method, to identify that we want to retrieve only the subset of the data that corresponds with submodel 30. If we wanted all the data, we would leave out the argument, but keep the empty parentheses. Note that nothing special happens when this command is executed. If it succeeded, it will create the new dataset object called ds30, and return to the python prompt. At this point, we can use a variety of built in methods for the dataset class to further explore the data. The first of these methods is summary(), which computes a statistical summary of the data in this object, like so:

```
>>> ds30.summary()

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>mean</th>
<th>sd</th>
<th>sum</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>lnemp20tw</td>
<td>8.94</td>
<td>1.61</td>
<td>4075.61</td>
<td>5.02388</td>
<td>12.2689</td>
</tr>
<tr>
<td>hbwavgtmda</td>
<td>13.15</td>
<td>1.8</td>
<td>5995.99</td>
<td>10.1786</td>
<td>18.0341</td>
</tr>
<tr>
<td>lnunits</td>
<td>1.08</td>
<td>1.47</td>
<td>492.089</td>
<td>0</td>
<td>5.63121</td>
</tr>
<tr>
<td>ln_bldgage</td>
<td>3.08</td>
<td>1.41</td>
<td>1404.12</td>
<td>-0.287682</td>
<td>4.60517</td>
</tr>
</tbody>
</table>

Size: 456 records

identifiers:
  id in range 1-456
```

Another useful Dataset method is the plot_histogram, which computes a histogram for an attribute of a dataset and plots it, as shown in Figure 23.3:

```
>>> ds30.plot_histogram('ln_bldgage')
```

After interactive exploration of the data used in the model, we might choose to drop or add variables from the specification. For demonstration purposes, drop lnemp30da from the specification of submodel 30, in the GUI, and save the project. Alternatively this could be done by editing the seattle_parcel.xml (be careful to use an editor that will not damage the format of the file, for example in Windows you can use a Notepad version that has added XML support). Once the specification has been edited and saved, re-run the estimation for only submodel 30 like this:
>>> estimator.reestimate(30)
Estimating Real Estate Price Model (from urbansim.models.real_estate_price_model):
started on Thu May 22 09:36:12 2008
Estimate regression for submodel 30
Number of observations: 456
WARNING: Estimation may led to singularities. Results may be not correct.
R-Squared: 0.23563241597137002
Adjusted R-Squared: 0.22368917247092268
Suggested |t-value| > 2.4743671533372704

---

<table>
<thead>
<tr>
<th>Coeff_names</th>
<th>estimate</th>
<th>SE</th>
<th>t-values</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>7.69055</td>
<td>0.83843</td>
<td>9.17256</td>
</tr>
<tr>
<td>hbwavgtmda</td>
<td>-0.0395769</td>
<td>0.0164105</td>
<td>-2.41168</td>
</tr>
<tr>
<td>ln_bldgage</td>
<td>-0.0340862</td>
<td>0.0220738</td>
<td>-1.54419</td>
</tr>
<tr>
<td>ln_invfar</td>
<td>0.26758</td>
<td>0.0346613</td>
<td>7.71983</td>
</tr>
<tr>
<td>lnemp20tw</td>
<td>-0.030372</td>
<td>0.0279526</td>
<td>-1.08655</td>
</tr>
<tr>
<td>lngcdacbd</td>
<td>-0.545171</td>
<td>0.200477</td>
<td>-2.71937</td>
</tr>
<tr>
<td>lnnsqft</td>
<td>-0.118709</td>
<td>0.0256177</td>
<td>-4.63389</td>
</tr>
<tr>
<td>lnunits</td>
<td>0.204933</td>
<td>0.0268876</td>
<td>7.62182</td>
</tr>
</tbody>
</table>

---

Estimating Real Estate Price Model (from urbansim.models.real_estate_price_model):
completed...0.1 sec

Notice that now we see a warning, indicating a problem in the specification. Let’s drop the least significant variable, lnemp20tw, and try estimating again:

>>> estimator.reestimate(30)
Estimating Real Estate Price Model (from urbansim.models.real_estate_price_model):
started on Thu May 22 09:43:26 2008
Estimate regression for submodel 30
Number of observations: 456
R-Squared: 0.23361810689099238
Adjusted R-Squared: 0.22337692346414595
Suggested |t-value| > 2.4743671533372704

---

<table>
<thead>
<tr>
<th>Coeff_names</th>
<th>estimate</th>
<th>SE</th>
<th>t-values</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>6.99788</td>
<td>0.544696</td>
<td>12.8473</td>
</tr>
<tr>
<td>hbwavgtmda</td>
<td>-0.0378939</td>
<td>0.0163405</td>
<td>-2.31901</td>
</tr>
<tr>
<td>ln_bldgage</td>
<td>-0.037881</td>
<td>0.0218001</td>
<td>-1.73765</td>
</tr>
<tr>
<td>ln_invfar</td>
<td>0.271372</td>
<td>0.0344921</td>
<td>7.86765</td>
</tr>
<tr>
<td>lngcdacbd</td>
<td>-0.390517</td>
<td>0.141209</td>
<td>-2.76553</td>
</tr>
<tr>
<td>lnnsqft</td>
<td>-0.121596</td>
<td>0.0254847</td>
<td>-4.77133</td>
</tr>
<tr>
<td>lnunits</td>
<td>0.203752</td>
<td>0.0268711</td>
<td>7.58258</td>
</tr>
</tbody>
</table>

---

Now the results do not indicate a warning, though we could experiment further to refine the specification. This interactive approach is very efficient for rapidly experimenting with model specifications and re-estimating a single model. The same approach is available to estimate a subset of the models, using the following syntax:

>>> estimator.reestimate(submodels=[3,7,9])
23.6.2 Estimation of a Choice Model

From the command shell (but not in python), we can also start the estimation of a choice model, like the housing type choice model created earlier in this chapter:

```
python -i c:\opus\src\urbansim\tools\start_estimation.py
-x c:\opus\project_configs\seattle_parcel.xml -m housing_type_choice_model
```

(Put this command in one single line or with proper continuation mark for respective operating system.)

The resulting output is shown below:

```
(clipped listing)
...
  single_family=(household.disaggregate(building.building_type_id)==19)+1....0.3 sec
  submodel: -2
  Convergence achieved.
  Akaike's Information Criterion (AIC): 333880.22475104179
  Number of Iterations: 18
  ***********************************************
  Log-likelihood is: -166938.1123755209
  Null Log-likelihood is: -177492.11908410664
  Likelihood ratio index: 0.059461832801627756
  Adj. likelihood ratio index: 0.05945056469569318
  Number of observations: 256067
  Suggested |t-value| > 3.5289083875852207
  Convergence statistic is: 0.00048567907076085262
  -----------------------------------------------
  Coeff_names          estimate       std err       t-values
  constant             0.600053       0.00561164   106.93
  income               1.13913e-05    5.77362e-08   197.299
  ***********************************************
  Elapsed time: 11.355681 seconds
  Estimating Choice Model (from opus_core.choice_model): completed...12.5 sec
```

Note that the same method used for the regression model, `get_data_as_dataset`, can be used to retrieve the data and analyze it interactively.

There are additional tools in the estimator that provide help for exploring and analyzing the data and model:

- `plot_utility` helps reveal the relative influence of each variable on the utility of the choice model. After replacing the estimation procedure of the choice model “bhhh_mnl_estimation” with “bhhh_mnl_estimation_with_diagnose” in the configuration, enter the interactive estimation mode as shown above, then in the Python shell

  ```python
  >>> estimator.plot_utility()
  ```

- `prediction_success_table` provides a way to do in-sample validation, by applying the estimated coefficients to the estimate data to get predicted choices and comparing these choices to the observed ones.
For location choice model, for example, Household Location Choice Model, where there are too many alternatives for this to be useful, a summary topology can be provided:

```python
>>> estimator.create_prediction_success_table()

>>> estimator.create_prediction_success_table(summarize_by= "building_type_id=building.building_type_id")

>>> estimator.create_prediction_success_table(summarize_by= "large_area_id = household.disaggregate(faz.large_area_id, intermediates = [zone, parcel, building])")

# log output to a tab delimited file

>>> estimator.create_prediction_success_table(summarize_by= "area_type_id=building.disaggregate(zone.area_type_id, intermediates=[parcel])",log_to_file='a.out')
```
from opus_core.variables.variable import Variable
from urbansim.functions import attribute_label
from variable_functions import my_attribute_label

class population(Variable):
    """Compute the total number of people residing in a gridcell, by summing the attribute 'persons' over all households in the gridcell""
    _return_type="int32"
    hh_persons = "persons"

def dependencies(self):
    return [attribute_label("household", self.hh_persons),
            attribute_label("household", "grid_id"),
            my_attribute_label("grid_id")]

def compute(self, dataset_pool):
    households = dataset_pool.get_dataset('household')
    return self.get_dataset().sum_dataset_over_ids(households, self.hh_persons)

from opus_core.tests import opus_unittest
from opus_core.tests.utils.variable_tester import VariableTester
from numpy import array

class Tests(opus_unittest.OpusTestCase):
    def test_my_inputs(self):
        gridcell_grid_id = array([1, 2, 3])
        # specify an array of 4 hh's, 1st hh's grid_id = 2 (it's in gridcell 2), etc.
        household_grid_id = array([2, 1, 3, 2])
        # specify how many people live in each household
        hh_persons = array([10, 5, 20, 30])

        tester = VariableTester(
            __file__,
            package_order=['urbansim'],
            test_data={
                "gridcell":{
                    "grid_id":gridcell_grid_id
                },
                "household":{
                    "household_id":array([1,2,3,4]),
                    "persons":hh_persons,
                    "grid_id":household_grid_id
                }
            }]
        )

        should_be = array([5, 40, 20])
        tester.test_is_close_for_family_variable(self, should_be)

if __name__=='__main__':
    opus_unittest.main()

Figure 23.1: Definition of the 'population' variable as a Python class
Figure 23.2: Correlation Matrix Plot for Submodel 30 in Real Estate Price Model

Figure 23.3: Histogram of Lnsqft from Data in Submodel 30 of the Real Estate Price Model
Part VI

Opus and UrbanSim API
The opus_core Opus Package

Opus is organized as a set of “Opus packages”. Each Opus package encapsulates a set of functionality in a structure defined by a set of required directories and files. This chapter describes main objects provided by the Opus package called opus_core. This package is intended to provide a fairly general functionality, including data representation and manipulation, various models for use in different domains, support for specification of models, or definition of variables. The package by itself does not provide a self-contained system of configured models that would run after pressing a button. It is rather a collection of tools for building other opus packages.

24.1 Datasets

The basic class for dealing with data is called Dataset. A dataset is a collection of attributes for a particular type of entity, such as a set of grid cells, or a set of households. Each member in this set has the same set of characteristics, such as income of households. In Opus, these characteristics are called attributes.

Conceptually, a dataset is similar to a table. Each attribute is a column. Each row describes the attribute values for one member of the dataset.

Some attributes are read from a data store, we call them primary attributes. Others, computed attributes, are computed by Opus variable definitions. Attributes can be also modified or created by models.

In this section, we describe the main functionality of Dataset. For a list of additional methods, see Appendix D.1.

24.1.1 Initialization

The Dataset class is initialized by the following arguments:

- resources - an object of class Resources (dictionary). It can contain any of the remaining arguments, but if an argument of the constructor is not None, it has a priority over the entry in resources.
- in_storage - an object of class Storage for reading the data from (see Section 24.2).
- id_name - a list of character strings giving the names of the unique identifiers of the dataset. If it is an empty list, a “hidden” identifier will be created, i.e. an additional attribute of the dataset that enumerates the entries. Note that the unique identifier must have integer values larger than 0.
- dataset_name - a character string giving the name of the dataset.
**out_storage** - an object of class *Storage* for writing the data into (see Section 24.2). This argument can be omitted in the constructor and instead directly passed to the `write_dataset()` method, which is the only method of the class that uses this argument.

**in_table_name** - name of the table, directory or file that contains the data for this dataset (see Section 24.2 for the different meanings of this argument in the different storage classes). This name also is the name used for this dataset in the cache (see below).

**out_table_name** - name of the table, directory or file that the dataset should be written into (see Section 24.2 for the different meanings of this argument in the different storage classes). This argument can be omitted in the constructor and instead directly passed to the `write_dataset()` method.

All arguments are merged with *resources* and kept in the class attribute *resources*.

The constructor determines all attributes available on the input storage (primary attributes), without loading them into memory. If *id_name* is an empty list, it loads at least one attribute from the storage in order to obtain the dataset size. The constructor also sets up a reference to the singleton *AttributeCache* which is used for caching data (see below).

### 24.1.2 Loading Attributes

Primary attributes are read lazily, i.e. they are loaded from the input storage as they are needed, for example when they are required by a model or by a variable definition. The loading can be also explicitly invoked by the method `load_dataset()` which by default loads all attributes found on the data storage. An optional argument *attributes* can be passed which is a list of attributes to be loaded.

Alternatively, the method `get_attribute(name)` invokes the specified attribute to be loaded into memory if it has not been done before and returns an array of values for this attribute. Note that when using a slow storage, such as MySQL, loading several attributes at once using `load_dataset()` (as opposed to one by one using `get_attribute()`) might be more time efficient.

Loaded or computed dataset attributes can be flushed into a simulation cache to save memory (using method `flush_attribute(name)` for a specific attribute or `flush_dataset()` for the whole dataset). Opus uses an FLT storage for the simulation cache (see Section 24.2.4) which deals with reading and writing data in a very efficient way. Thus, when using a slow storage, it might be of an advantage to load the whole dataset at the beginning of the dataset usage and flush all attributes. Then any subsequent load will be performed on the fast cache.

### 24.1.3 Attribute Names

Attributes for a dataset can be specified in following ways:

**Un-qualified name** (e.g. `population`) It is a character string without any dots or parentheses. It may contain numbers. Un-qualified names within a dataset must be unique. This way of specifying attributes may be used only for accessing already existing attributes within a dataset, e.g. in the method `get_attribute()`.

**Dataset-qualified name** (e.g. `gridcell.population`) Specifies a primary attribute of a specific dataset. It consists of a dataset name (e.g. `gridcell`) and an un-qualified attribute name (e.g. `population`). The dataset name allows you to disambiguate attributes of the same name in different datasets, when using outside of a dataset, e.g. in a model specification or in variable dependencies.

**Fully-qualified name** (e.g. `urbansim.gridcell.population`) Specifies an Opus variable. It is the full name of the module or class in which the variable is defined. See Section 24.3.1 below, for more information.
Expressions (e.g. \( \ln(\text{urbansim.gridcell.population+1}) \)) Expressions are composed from variable names, constants, functions, and operators — see Chapter 13 and Section 24.3.4.

Expressions and fully-qualified names can be given an alias using the syntax alias = expr. In that case, the alias name is used as un-qualified name for this attribute and thus, values of this attribute can be accessed by get_attribute(alias).

24.1.4 Computing Variables

A variable (or equivalently, a computed attribute) is specified either by an expression of by a fully-qualified name (see Section 24.3). Datasets automatically compute each variable when, and only when, needed. This mechanism uses the dependency information from each variable which gives an information about what variables this variable depends on. Additionally, it is checked if variable’s dependent variables have changed (versioning mechanism). The computation is invoked by the dataset method compute_variables() which computes each of the given variables if either (a) this variable has not been computed before, or (b) the inputs to this variable (the values of variables upon which this variable depends) have changed since the last computation. Thus, invoking compute_variables() on a single variable may either result in no more computation, or have a ripple effect of computing many variables upon which this one variable depends. Lazily computing variables both helps minimize the computational load as well as eliminating the need to worry about when variables are computed: it will happen when, and only when, it is needed.

The method compute_variables() takes the following arguments:

- names - a list of variable names to be computed,
- dataset_pool - an object of class DatasetPool which maintains a ‘pool’ of additional datasets that the variables (and their dependent variables) need for the computation (see Section 24.7.1), and
- resources - an object of class Resources which can contain additional arguments to be passed to each of the variable computation (see Section 24.3.2).

The method returns an array of values that result from computing the last variable in the list of variable names.

24.1.5 Visualizing Datasets

Opus offers a few methods for plotting values of a dataset. They usually require specific libraries to be installed, such as matplotlib (see your Installation directions at your ‘opus_docs/docs/install.html’).

One-dimensional Plots

Attributes in a dataset are stored as one-dimensional arrays, thus they can be plotted for example as a histogram or as a scatter plot. The Dataset offers the following methods:

- plot_histogram(name, ...) creates a histogram of attribute name using matplotlib.
- r_histogram(name, ...) creates such histogram including a density line using the rpy library.
- plot_scatter(name_x, name_y, ...) creates a scatter plot for two attributes using matplotlib.
- r_scatter(name_x, name_y, ...) creates a scatter plot using the rpy library.
Two-dimensional Plots

One often is interested in a spatial graphical presentation of a dataset attribute. By default, datasets have no spatial coordinate axes. If you want a dataset to have a coordinate system, set the dataset’s class attribute _coordinate_system to be a tuple of the attribute names for the x and y coordinates, e.g. to ('relative_x','relative_y'). This information then will be used by routines that need it.

plot_map(name, ...) creates an image using the matplotlib library.
r_image(name, ...) does the same using the rpy library.

24.1.6 Attribute Box

For each attribute that is loaded into memory or computed, the dataset creates an instance of AttributeBox. This object holds all information about the attribute, such as the data, attribute name, version, type, if it is cached or in memory, and a corresponding instance of class Variable if this attribute is a variable. This information can be accessed by the AttributeBox methods get_data(), get_variable_name(), get_version(), get_type(), is_cached(), is_in_memory(), get_variable_instance(). The attribute box for a particular attribute can be accessed by the dataset method _get_attribute_box(attribute_name), where the attribute_name is given in one of the forms from Section 24.1.3.

24.1.7 Subsets of Dataset

Opus implements a child class of Dataset, called DatasetSubset, which allows to define a subset of a dataset. Conceptually, it is a viewing window for the parent Dataset object, not a copy. Thus, any change in the parent object is seen by the child.

It is initialized by passing the parent object and an index to the constructor. The index determines indices of elements within the parent object that should be seen. Any call of get_attribute() on the subset returns values corresponding to the index. A subset can be also created using the Dataset method create_subset_window_by_ids(ids) called on the parent object.

Note that only methods for viewing attributes (such as get_attribute() or summary()) make sense for using with a subset. Other methods that would for example modify attributes (such as compute_variables()) could destroy the parent dataset.

24.1.8 Interaction Sets

Opus allows the programmer to create a dataset representing an interaction between two datasets, by using the class InteractionDataset, which is a child of Dataset. It serves mainly to enable the use of the Opus variable concept for variables that are defined as an interaction between datasets, and thus are of a 2-d shape. The class does not support loading from and writing into a storage.

Initialization

The class is initialized by the following arguments:

resources - this argument has the same meaning as in Dataset.
dataset1 - an instance of Dataset.
dataset2 - an instance of Dataset.
index1 - indices of dataset1 over which the interaction is done. If it is not given, all elements are taken.
index2 - indices of dataset2 over which the interaction is done. If it is not given, all elements are taken.
dataset_name - a name of the interaction set. Default value is the dataset name of dataset1 connected by ‘_x_’ to the dataset name of dataset2.

Using InteractionDataset

The method get_attribute() returns a 2-d array, where a value $x_{ij}$ belongs to interaction of the $i$-th element within elements of dataset1 given by index1 and the $j$-th element within elements of dataset2 given by index2. Thus, the array size corresponds to size of index1 x size of index2.

The two interacting datasets can be accessed by the method get_dataset() which takes either 1 or 2 as an argument value for dataset1 or dataset2.

The method get_attribute_of_dataset(name, dataset_number) returns values of the given attribute that belongs to the dataset given by dataset_number where only values associated to the corresponding index are included. Thus, a call get_attribute_of_dataset("attr", 1) gives an array of size index1, whereas a call get_dataset(1).get_attribute("attr") gives an array containing values for all elements of dataset1.

The method compute_variables() determines from each of the fully-qualified names of variables or expressions, to which dataset the variable belongs to. If it belongs to the dataset1 or dataset2, it calls compute_variables() on those datasets (the values are computed for all elements of the datasets, regardless of the given index). If it is an interaction variable, it is computed only for elements given by index1 and index2.

The InteractionDataset class contains several methods that are useful for variable computation on a 2-d array, such as multiply(), divide(), is_same_as(), is_less_or_equal(), is_greater_or_equal(), all of which take as arguments two interacting attribute names.

An interaction set is automatically created and used in the ChoiceModel class (see Section 24.4.3).

24.2 Data Storage

One of the design elements of Opus is to allow users to choose alternative data storage methods depending on their needs, while keeping a consistent internal representation of data used by the system. Currently, Opus classes exist for reading and writing data in Random Access Memory (RAM), to database tables, tab- and comma-delimited ASCII files, and binary files. Other storage methods can be added as the need arises.

The generic class that supports data storage is called Storage. Every class that we describe below derives from this class and implements the storage interface. All such classes therefore provide the following methods:

load_table(): Loads the data stored in the data repository and returns it as a dictionary, where each key represents an attribute (column) of the table. The only required argument is table_name. Optional arguments include column_names (a list of columns that you want loaded; the default is to load all) and lowercase (for forcing all names column names to lowercase).

write_table(): Writes data out to the respective storage. The two non-optional parameters are table_name and table_data, the former naming the table and the latter defining the data to be written. More specifically, the
table_data argument should be a dictionary, where the keys represent a column and the values represent the data in those columns. This is the same format as that returned by the load_table() method. An optional parameter mode allows you to determine whether an existing table of the same name will either be overwritten or appended to with the new data. By default the table is overwritten.

get_storage_location(): Returns the location at which data is being stored for this storage object. In the case of storage locations that exist on disk, a file path is returned. In the case of a database, an OpusDatabase object will be returned.

get_column_names(): Takes a table_name as an argument and returns a list of attribute names that were found on the storage.

table_exists(): Takes a table_name as an argument and returns whether a table by that name exists at the respective storage location.

get_table_names(): Returns a list of all the names of tables that exist at the respective storage location.

Developer note: In order to be able to implement the storage interface, the new storage subclass must implement methods load_table, get_storage_location, write_table, and get_column_names.

The predefined storage classes in opus_core are dict_storage, csv_storage, tab_storage, sql_storage, flt_storage, dbf_storage, and esri_storage implemented in modules of the same name in opus_core.store. Their instances can be created using the method get_storage(type, storage_location, ...) of class StorageFactory; type is the storage type (e.g. “sql_storage”) and storage_location is either a location on disk or an OpusDatabase, depending on the type of storage being created. Other optional, storage-type specific arguments can also be passed in to the get_storage method. We now give a brief description of each available storage type, and how each may deviate from the above descriptions.

24.2.1 dict_storage

The simplest storage class. It is an in-memory implementation of the storage interface. The constructor of this class has no parameters (no storage_location parameter needed). Obviously, the first method to be used before being able to do anything useful with an object of this type is the method write_table().

24.2.2 csv_storage and tab_storage

Both csv_storage and tab_storage provide file-based storage. They are based upon Python’s csv module, and thus will appropriately format data as necessary.

The constructor of this class expects an entry storage_location in its arguments. It should be a base directory on a hard drive where the data will be stored to and loaded from.

For these storage classes, all implemented methods of the storage interface which accept a table_name parameter interpret it as a file name without extension (relative to the base directory) in which the data is stored (or where it should be stored). The extension is added automatically. The data is stored with one attribute per column. The first row contains the attribute names and optional type information for each column. The type information, if provided, is appended to the column name and separated by a colon, as in id:int32 which specifies to use the numpy int32 type for storing the id values. The type may be any of the numpy types in Table 24.1.

If type information is not included, Opus will use float64 for numeric data and string for string data.

For instance, the csv_storage for a dataset with three attributes a, b, and c containing two rows of data could look
<table>
<thead>
<tr>
<th>Column Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool8</td>
</tr>
<tr>
<td>int8</td>
</tr>
<tr>
<td>uint8</td>
</tr>
<tr>
<td>int16</td>
</tr>
<tr>
<td>uint16</td>
</tr>
<tr>
<td>int32</td>
</tr>
<tr>
<td>uint32</td>
</tr>
<tr>
<td>int64</td>
</tr>
<tr>
<td>uint64</td>
</tr>
<tr>
<td>float32</td>
</tr>
<tr>
<td>float64</td>
</tr>
<tr>
<td>complex64</td>
</tr>
<tr>
<td>complex128</td>
</tr>
<tr>
<td>string</td>
</tr>
</tbody>
</table>

Table 24.1: Allowable column types for csv_storage and tab_storage. *x* in the last line is an integer determining length of the string.

like:

```
a:int8,b:float32,c:string40
1,3.14,hello
2,2.18,there
```

24.2.3 sql_storage

The constructor of this class expects an entry `storage_location` in its arguments. This parameter should be an `OpusDatabase` and it governs a connection to a database. Here’s an example of obtaining an `OpusDatabase` and then creating a storage object that can read and write from that database:

```python
>>> import os
>>> from opus_core.database_management.database_server_configuration import DatabaseServerConfiguration
>>> from opus_core.database_management.database_server import DatabaseServer
>>> from opus_core.storage_factory import StorageFactory
>>> config = DatabaseServerConfiguration(hostname = 'my.host.net',
                                          username = 'me',
                                          password = 'my_password',
                                          protocol = 'mysql')
>>> db_server = DatabaseServer(config)
>>> db = db_server.get_database(database_name = '\mydatabase')
>>> storage = StorageFactory().get_storage('sql', storage_location = db)
```

Note that you should probably never have to create a `DatabaseServerConfiguration` object directly – instead, you should instantiate the child classes of EstimationDatabaseConfiguration, IndicatorsDatabaseConfiguration, ServicesDatabaseConfiguration, or ScenarioDatabaseConfiguration. Configurations of these types will default to the server information found in “[OPUS_HOME]/settings/database_server_configuration.xml”.
Now you have obtained a sql_storage object. For this storage class, all implemented methods of the storage interface which accept a table_name parameter interpret it as name of a table in the database which is to be read from or written to, depending on the method.

24.2.4 flt_storage

As in the case of tab storage, the constructor expects an entry storage_location in its argument. It is the base directory where the data are stored to and loaded from.

For this storage class, all implemented methods of the storage interface which accept a table_name parameter interpret it as a subdirectory in the base directory in which each attribute is stored as a single file in a binary format. The file names correspond to attribute names with an extension that determines the type of the stored data. The file extension consists of three parts, one character for the byte order, one character for the type of the data, and one or more characters for the size of the column in bytes. The extension is similar to the dtype for numpy arrays, only the character for the byte order is changed from '<', '>', or '|'(for little endian, big endian, or irrelevant) to 'l', 'b', or 'i'. The method write_table() stores attribute according to the scheme above.

24.2.5 dbf_storage

The dbf_storage provides DBase-formated file-based storage for datasets. It uses the dbfpy Python package available at http://sourceforge.net/projects/dbfpy.

The constructor of this class expects an entry storage_location in its arguments. It should be a base directory on a hard drive where the data will be stored to and loaded from.

For this storage class, all implemented methods of the storage interface which accept a table_name parameter interpret it as a file name (relative to the base directory) without an extension in which the data are stored in an "dbf" file. The extension "*.dbf" is added automatically. Each table is stored in one file. The format of these files is described at http://www.clicketyclick.dk/databases/xbase/format/data_types.html#DATA_TYPES.

24.2.6 esri_storage

The esri_storage class provides ESRI-based file storage for datasets. It uses the ESRI Geoprocessing framework, interfacing via Python, to provide access to ESRI storage objects. Consequently, ArcGIS v9.x is required to utilize this storage class. Any 9.x version of ArcGIS should work fine, although it is recommended that you read through our instructions for installing Opus along with ArcGIS here http://trondheim.cs.washington.edu/cgi-bin/trac.cgi/wiki/PythonAndArcGIS.

The constructor of this class takes a storage_location in its arguments. This can be one of several different types of paths. It can be a standard directory path (e.g. '/c/temp'), a path to a personal or file geodatabase (e.g. '/c/temp/mydb.mdb' or '/c/temp/mydb.gdb'), or it can be a path to an ArcSDE database connection (e.g. 'Database Connections/your_db_conn.sde'). In the case that you use an ArcSDE path, that connection must already exist in ArcCatalog with the proper connection information.

The esri_storage class will read the attributes of any file type that ESRI normally provides access to within ArcGIS. This includes the attribute tables of shapefiles, standalone dbf files, feature classes in geodatabases, and tables within geodatabases.

The esri_storage class will write dbf tables or tables within a geodatabase. If the constructor were given a regular file path, dbf tables will be written. In this case, in order to conform with dbf standards, long column names will be
truncated and renamed to be unique. In the case of geodatabase tables, tables will be written with the full column and table names intact.

For additional information about specific methods in the `esri_storage` class, see the python module itself for docstrings that document specifics about the methods and arguments.

### 24.3 Opus Variables

An Opus variable is an algorithm that provides a value for each element of a dataset. In most cases, the algorithm transforms or summarizes existing data.

#### 24.3.1 Variable Names

An Opus variable defined as a Python class is identified by its fully-qualified name, such as `urbansim.gridcell.population`. The fully-qualified variable name encodes three pieces of information: the package containing it (e.g., `urbansim`), the name of the dataset to which this attribute belongs (e.g., `gridcell`), and the un-qualified name of the variable which is unique within that dataset (e.g., `population`). Using fully-qualified path names explicitly indicates where each variable’s definition comes from.

Opus contains a class `VariableName` that is initialized by passing a name in one of the forms above. Instances of this class can be also used when accessing attributes or invoking variable computation.

The similarity to a Python `import` statement is not a coincidence: Opus uses an `import` statement to load the variable’s definition.

The un-qualified part of an Opus variable name may specify a “template” that matches a family of related variables. The variable `mypackage.mydataset.number_of_SSS_projects_within_DDD_meters`, for instance, matches `mypackage.mydataset.number_of_residential_projects_within_500_meters` as well as `mypackage.mydataset.number_of_industrial_projects_within_1000_meters`. When Opus looks for the definition of this variable, it matches any SSS to strings of alphabetic characters and underscores, and any DDD to integers. These values are then passed to the variable code, allowing it to modify its behavior according to the specified values. When there is an ambiguity, Opus will issue a fatal error.

Opus variable names must be lower-case, except for any SSS and DDD pattern makers. This reduces the change of problems with incorrect case, or with migrating Opus code between operating systems that are case-sensitive and those that are not.

#### 24.3.2 Implementation

The behavior of each Opus variable is defined in a Python class that is a child of the generic class `Variable`. The name of the Python module and class containing the variable implementation must be the same as the un-qualified part of the Opus variable name, e.g. class `population` is implemented in file `population.py` for variable `urbansim.gridcell.population`. The module should be stored in a directory whose name is the dataset name for which the variable is computed, in the corresponding package. In the above example, ‘population.py’ is stored in the `gridcell` directory in the top level of the `urbansim` package directory. This scheme allows Opus to find that variable. It also means there may be only one Opus variable per Python variable module.
Creating a Variable Object

An instance of an existing variable class is created using the method `get_variable()` of the class `VariableFactory`. It takes as arguments the variable name and the dataset which the variable belongs to. It performs template matching of the name and invokes the constructor of the corresponding variable. If there were any templates replaced by `SSS` or `DDD`, they are passed to the constructor in order as they appear in the name. It then calls the Variable method `set_dataset()` which sets the given dataset as a class attribute `dataset` of the variable. Thus, using the Variable method `get_dataset()` each variable object has an access to its “owner” dataset.

The `VariableFactory` method `get_variable()` returns an instance of the created variable class.

Variable objects are created automatically within the dataset method `compute_variables()` and stored in attribute boxes (Section 24.1.6).

Required Method `compute()`

Each variable class has a required method `compute()` which defines how to compute this variable.

It takes as an argument an object of class `DatasetPool` which maintains a “pool” of additional datasets that this variable needs for its computation (see Section 24.7.1). Note that this method is called from the dataset method `compute_variables()` (see Section 24.1.4), and its `dataset_pool` argument is passed from there.

The implementation of the `compute()` method can assume that all variables that are listed in the `dependencies()` method (see below) are already computed and accessible by the dataset method `get_attribute()`.

The `compute()` method should return a numpy array of the size of number of elements in the “owner” dataset.

Optional Method `dependencies()`

This optional method takes no arguments and returns a list of variable names that this variable needs in order to compute its values. Each variable name must be either an expression, a fully-qualified Opus variable name, or a dataset-qualified primary attribute name. It is important to list all dependent variables here and not invoke the computation of the dependent variables from the `compute()` method, since the dependencies tree mechanism would not work correctly (see Section 24.3.3). If the names of the dependent variables are not known at the time this method is called and are determined dynamically, the dependencies list can be extended from the `compute()` method using the Variable method `add_and_solve_dependencies()` which takes as arguments a list of additional dependencies (each specified either as a character string or an objects of class `AttributeBox`) and the dataset pool object. `add_and_solve_dependencies()` computes the additional variables and extends the dependencies list only when the `compute()` method runs for the first time.

Other Methods and Properties

The following behavior is only available if the computation is invoked from the dataset method `compute_variables()` (see Section 24.1.4).

If the variable defines the property `_return_type` to be one of the following numpy types, Opus will automatically cast the results into that type:

- `bool8`
24.3.3 Variable Dependencies Tree

Opus implements a mechanism of computing variables structured in a dependencies tree. A variable is only computed if its children have changed since the last computation. Thus, a dataset object keeps a version information about each attribute or variable and each variable class keeps information about which versions of the dependent variables this variable was computed with. This mechanism is automatically invoked by calling the method `compute_variables()` of the class `Dataset`.

The argument `resources` passed into the `compute_variables()` method can contain an entry `check_variables`. This can be either a list of variable names or a '*' (meaning 'all variables'). For each of those variables a 'check' is performed. This means that it will issue a warning if the values being cast are too large to fit into the destination type. In addition, each Opus variable may have the optional `pre_check()` and `post_check()` methods to implement the “programming by contract” model of software development (see Section 26.3). These methods are also invoked on the 'checked' variables.
As an example, consider Figure 24.1. There are 8 variables/attributes, belonging to 3 datasets, ds1, ds2, and ds3. The arrows show the dependency hierarchy between variables. The hierarchy is defined in the method dependencies() of each variable by listing all direct 'children' variables as expressions, fully-qualified names or dataset-qualified names (in this example the names in the parentheses in the figure). Note that two attributes here are primary, namely “ds3.var_d” and “ds2.var_e”, and thus are defined in their dataset-qualified name (there is no variable implementation for those attributes). For example, the dependencies() method of the root variable “pkg.ds1.my_variable” returns a list of three elements: ['pkg.ds1.var_a', 'pkg.ds1.var_b', 'pkg.ds2.var_b'], and dependencies() of the variable “pkg.ds1.var_a” returns a list with one expression: ['var_c = log(ds3.var_d)/log(pkg.ds3.var_e)']. As described in Section 24.3.4, expressions do not have user-defined implementations (they are created on the fly). Therefore there is no need to define dependencies for the variable “ds3.var_c”, Opus determines them automatically.

Now suppose we have a system of several models, three of which are invoking the computation of “my_variable” of dataset ds1. Suppose also that initially we have created the three datasets and loaded the two primary attributes. Opus assigns to each newly created attribute a version number 0 in its attribute box. This situation is shown in the left upper corner of Figure 24.2. Box number 1 shows that there are only two attributes in the system, both with version number 0.

Box 2 shows a situation when the first model invokes the computation of “my_variable” \(^1\). The system works through the defined dependencies and computes all variables needed to compute “my_variable”. In this process, again, each newly computed variable gets the version number 0 (illustrated by the small boxes above each variable), stored in the attribute box of each variable. Additionally, each Variable instance keeps the version number for each dependent variable, on which this variable was computed. In the figure, this is illustrated by the small boxes bellow each variable. Note that all elements in Figure 24.2 that are created or change their values are shaded.

The box number 3 shows a situation in which another model changes values of variable “var_e” of the dataset ds2 which causes an increment of the version number. Therefore, when the next model invokes compute_variables("my_variable"), Opus determines that there is a mismatch between versions of “ds.var_e” on which variables “ds1.var_b” and “ds2.var_b” were computed and its current version (the mismatch is visualized by red crosses). Then all variables above “ds.var_e” are recomputed, their version numbers incremented, and the version numbers in the dependencies lists are updated (box 4).

A similar situation arises when another model changes the variable “ds3.var_e” (box 5). The next call of compute_variables("my_variable") causes a recomputing of 4 variables (box 6) In the right lower corner of Figure 24.2, the state of the three datasets is shown at the end of the run of our example models.

Note that the dependencies tree is constructed using the dependencies() method of Variable. Therefore a missing item in this method translates to a missing branch of the tree and thus failing to determine a need for recomputing.

### 24.3.4 Expression Language Implementation

Opus expressions are written in a domain-specific programming language called Tekoa (Chapter 13). Each newly-encountered expression is compiled into an automatically generated subclass of Variable, with similar functionality to that described above. This section describes how this compilation is done. For the most part, the implementation shouldn’t be relevant to the Opus user. But for the curious, for programmers who want to modify or extend the system, or (heaven forfend) in case something goes wrong, here is a description of the implementation. The code to implement expressions is in opus_core.variables.

---

\(^1\) Because of the dependencies on variables of other datasets, the correct call is
ds1.compute_variables("pkg.ds1.my_variable"), dataset_pool=pool) where pool is constructed as described in Section 24.7.1 and contains datasets “ds2” and “ds3”. We simplify the notation here for a better readability.
Figure 24.2: Scenario of computing and recomputing variables.
Regarding Tekoa extensions, one straightforward extension is to add additional functions to the language. The currently available unary functions are listed in Section 13.4. These all operate on numpy arrays. Additional functions in the domain-specific language can be supported by adding additional definitions to `opus_core.variables.functions`. (If you believe this extension would be of general interest, please coordinate with the Opus/UrbaSim implementors so that it can find its way into the code base.)

When a new expression is encountered, the system automatically compiles a new subclass of `Variable` that implements the computation defined by that expression. If the expression is a simple attribute or fully-qualified variable, evaluating the expression reduces to getting the value of the attribute or computing the value of the existing variable. Otherwise, the expression system generates and compiles a new variable to implement an expression. It keeps a cache of expressions that have already been processed, so that autogenerated variables can be reused when possible. These autogenerated variables have names like `autogenvar034`. They are compiled and live just in the current process — they aren’t stored on disk, so that the user never needs to see them, and so that different processes running on the same machine don’t interfere with each other.

Since expressions use standard Python syntax, they can be parsed using the standard Python parser module, rather than needing to write one. The parse tree for the expression is analyzed and the dependencies extracted to generate the dependencies method — the user doesn’t need to declare the dependencies for an expression. The `compute()` method for the autogenerated variable includes the user’s expression directly as part of the method. To enable this to work correctly, the method includes statements to set up the local environment in the method so that all of the names are properly bound. Here is an example. Suppose the input expression is `ln_bounded(urbansim.gridcell.population)`. Then the automatically generated class will be:

```python
class autogenvar034(Variable):
    def dependencies(self):
        return ['urbansim.gridcell.population']
    def name(self):
        return 'ln_bounded(urbansim.gridcell.population)'
    def compute(self, dataset_pool):
        urbansim = DummyName()
        urbansim.gridcell = DummyDataset(self, 'gridcell', dataset_pool)
        urbansim.gridcell.population = self.get_dataset().get_attribute('population')
        return ln_bounded(urbansim.gridcell.population)
```

The name of the class is generated (there is a class variable `autogen_number` in the class `AutogenVariableFactory` that starts at 0 and gets incremented each time it’s used in a new name).

The dependencies method is constructed by parsing the expression and finding all of the other variables that it references, and putting those into the returned list.

The compute method ends with a return statement that just returns `expr`. To make this work, we need to provide local bindings for e.g. `urbansim.gridcell.population`. We bind a local variable (named `urbansim` in the example) to an instance of `DummyName`, whose sole purpose in life is to have an attribute `gridcell` (and maybe other attributes if there are multiple dependencies). Then `urbansim.gridcell` is bound to an instance of `DummyDataset`, which is used in place of a real dataset in the autogenerated code. We then add a population attribute to `urbansim.gridcell`, bound to the value of the appropriate dataset attribute. (We use the dummy dataset rather than adding attributes to the real dataset, which might interfere with other attributes or not be garbage-collected as soon as they might otherwise be.) For the `get_attribute` call to get the value of the population attribute, we use the short version of the name – its value should already have been computed by virtue of being listed in the `dependencies()` method.

If the expression includes an alias, for example `pop = ln_bounded(urbansim.gridcell.population)`, then the code is all the same as above, except that the final return statement is replaced with...
Figure 24.3: Models in opus_core. Arrows indicate the inheritance hierarchy: \( a \rightarrow b \) means \( a \) inherits from \( b \).

```python
pop = ln_bounded(urbansim.gridcell.population)
return pop
```

The aggregate, disaggregate, and number_of_agents methods are defined on DummyDataset, so that they can be used in expressions.

### 24.4 Models

The opus_core package offers a few simple models that can serve as parent classes for user specific models or can be used directly. Their hierarchy is shown in Figure 24.3. Model and ChunkModel are abstract classes, each of the remaining models implements a specific functionality. The classes are described in detail in the next sections.

#### 24.4.1 Model Class

Model is the base class for implementing Opus models. It provides an automatic logger which prints out information about the model start, end and processing time. Model supports two methods: an obligatory method `run()` and an optional method `estimate()`. Each child of Model can define a class attribute `model_name` that is used as an identification of the specific model in the logger output. Any arguments can be passed into the `run()` and `estimate()` methods and those can return any values.

#### 24.4.2 ChunkModel Class

This class enables running models (i.e. processing the `run()` method) in several chunks, for example if a run of the whole model would require too much memory. The class does not have any effect on an `estimate()` method.
The Run Method

Input:

chunk_specification - a dictionary specifying how to determine the number of chunks to run the model in. It must contain either the key 'records_per_chunk' or the key 'nchunks'. The method passes it to the constructor of ChunkSpecification (see Section 24.7.3). None translates to 1 chunk.

dataset - an object of class Dataset along whose elements the model will be chunked.

dataset_index - an index of elements within dataset that are to be chunked. Default is None which means that all elements of dataset are considered.

result_array_type - a type of the resulting array. It can be any numerical type of numpy array. The default value is float32.

... - optional additional arguments that are passed to the method run_chunk(). They are expected to be keyword arguments.

Algorithm:

For each chunk (determined by the chunk_specification) the model calls the method run_chunk() and passes as arguments the corresponding portion of dataset_index, the whole dataset and all additional arguments. By default the order of elements in dataset_index is preserved. This can be changed by redefining the method get_agents_order() (see below).

Output:

The class returns an array of the same size as dataset_index and of type result_array_type (passed to the method as input). Values of this array are return values of the (possibly) multiple calls of the method run_chunk(). The i-th value in this array is a result for the dataset_index[i]-th element of dataset.

Implementing a Child Class

All models that are derived from the ChunkModel class must have the method run_chunk() implemented. It has two non-keyword arguments: an integer array and an object of class Dataset. The first argument is an index of elements within the second argument. The method can have additional keyword arguments. The method is expected to return an array of the same size as the size of the first argument (i.e. the chunk size). The i-th value of the result is expected to be associated with the dataset element indexed by index[i].

Optionally, the method get_agents_order() can be implemented in the child class. It determines the order in which dataset elements are passed into chunks. This method takes a dataset as an argument which is an object of class DatasetSubset containing only those elements of the original dataset that are defined by dataset_index. It returns an index of elements within the given subset determining the order in which the elements should be processed. For example, if this method returns a randomized index, the elements passed into run_chunk() will be processed in randomized order.

An example of a user-defined ChunkModel is in Section 23.4.

24.4.3 ChoiceModel Class

This class implements a functionality of a discrete choice model, an approach widely used in land use and transportation modeling. In principal, a set of agents make choices from a finite set of possibilities (alternatives). The choice
selection is based on probabilities derived from utilities. These are computed on the basis of given variables and coefficients. The model allows the coefficients to be also estimated within this framework. See also Section 14.6.

As there are many different aspects to consider in specifying a discrete choice model, the software has been designed in a modular way to accommodate substantial flexibility in configuring these models. Each component included in the model is passed as character string that determines the module=class name (as a fully qualified name) in which the component is implemented.

Initialization

The class is initialized by passing the following arguments:

**choice_set** - A list, array or dataset that represents the finite set of possible choices. It should have numeric values larger than zero.

**utilities** - A fully qualified name of the module for computing utilities (see Section 24.5.1). Default value is 'opus_core.linear_utilities'.

**probabilities** - A fully qualified name of the module for computing probabilities (see Section 24.5.2). Default value is 'opus_core.mnl_probabilities'.

**choices** - A fully qualified name of the module for determining final choices (see Section 24.5.3). Default value is 'opus_core.random_choices'.

**submodel_string** - If model contains submodels, this character string specifies what agent’ attribute determines those submodels. Default is None, i.e. no submodels.

**choice_attribute_name** - Name of the attribute that identifies the choices. This argument is only relevant if **choice_set** is not an instance of Dataset. Otherwise the choices are identified by the unique identifier of the **choice_set**. Default value is 'choice_id'.

**interaction_pkg** - This argument is only relevant if there is an implementation of an interaction dataset that corresponds to interaction between agents and choices. It is the name of the Opus package where the implementation lives. Default value is 'opus_core'.

**run_config** - A collection of additional arguments that control a simulation run. It should be of class Resources. Default is None.

**estimate_config** - A collection of additional arguments that control an estimation run. It should be of class Resources. Default is None.

**dataset_pool** - A pool of datasets needed for computation of variables. Default is None.

The initialization method creates a class attribute upc_sequence, using the passed arguments utilities, probabilities and choices. It is an object of class upc_sequence (see Section 24.5.4). A class attribute choice_set is an object of Dataset. If the argument choice_set is a list or array, a Dataset is created using the values of the argument as the unique identifier. The name of this unique identifier is the value of choice_attribute_name.

The Run Method

The run() method runs the simulation on basis of a given specification and coefficients.

*Input:*
**specification** - an instance of class `EquationSpecification` specifying variables to be used in the simulation (see Section 24.6.1).

**coefficients** - an instance of class `Coefficients` that contains values of coefficients to be used in the simulation (see Section 24.6.2).

**agent_set** - an instance of class `Dataset` representing the whole set of agents to be used for the variable computation.

**agents_index** - an index within the `agent_set` determining which agents enter the choice process. If it is not given (default), all agents are considered.

**chunk_specification** - a dictionary specifying how to determine the number of chunks to run the model in. It is passed to the `run` method of `ChunkModel` (see Section 24.4.2). Default is `None`, which is equivalent to 1 chunk.

**data_objects** - a dictionary containing additional datasets needed for computing variables. This argument is obsolete – the datasets should be included in the `dataset_pool` argument of the constructor.

**run_config** - additional `Resources` for controlling the simulation run.

**Algorithm:**
The algorithm is implemented in the method `run_chunk()` called from the parent class `ChunkModel` for each chunk. It overwrites the `ChunkModel` method `get_agents_order()` in a way that it returns a permutation of the agents indices. Thus, the agents make their choices in a random order.

The method creates an interaction set between the agent set and the choice set. It computes all variables given in the specification. Variables that are specific to one of the datasets are computed on all elements of that dataset, interaction variables are computed only on elements entering the choice process. The method then creates the corresponding data matrix using `agents_index` for selecting the appropriate data values. It runs one simulation per submodel (for submodels specified in the specification) by calling the `run()` method of the `upc_sequence` attribute (see 24.5.4) and passing the data matrix and the coefficients for the corresponding submodel.

If `run_config` contains the entry 'demand_string' having a character string value, an aggregated demand for each choice is computed and stored as an additional attribute (of that name) of the choice dataset.

**Output:**
The method returns an array of size `agents_index`, representing the choices that agents (elements of `agent_set` determined by `agents_index`) made. Agents whose choice is less equal zero were not included in the choice process, for example because they do not belong to any submodels given in the specification.

The Estimate Method

The `estimate()` method runs an estimation of coefficients on basis of a given specification.

**Input:**

**specification** - an instance of class `EquationSpecification` specifying variables to be used in the estimation (see Section 24.6.1).

**agent_set** - a `Dataset` representing the whole set of agents to be used for the variable computation.

**agents_index** - an index within the `agent_set` determining which agents will be used for the estimation. If it is not given (default), all agents are considered.

**procedure** - a character string giving the fully qualified name of the estimation procedure (see Section 24.5.6). This argument can be also passed via `estimate_config` as an entry 'estimation'. The default value is `None`.

24.4. Models
**data_objects** - a dictionary containing additional datasets needed for computing variables. This argument is obsolete – the datasets should be included in the `dataset_pool` argument of the constructor.

**estimate_config** - additional Resources for controlling the estimation run.

**Algorithm:**

In addition to `agents_index`, the number of agents entering the estimation can be controlled by an entry 'estimation_size_agents' in `estimate_config` which should have a value between 0 and 1. It gives the portion of `agents_index` (or `agent_set` if `agents_index` is not given) that will be used in the estimation. The indices are then randomly sampled. In addition, an entry 'submodel_size_max' of `estimate_config` controls the maximum number of records within each submodel. If a submodel is larger than this number, records are randomly sampled to meet this threshold. The `agent_set` should contain an attribute of the same name as the unique identifier of the class attribute `choice_set`. Its values determine the current choices of the agents.

As in the `run()` method, an interaction set is created and the variables given in the specification are computed. Then for each submodel the corresponding data matrix is built and the `run()` method of the class given by the argument `procedure` (or alternatively by an entry “estimation” in `estimate_config`) is called, passing the data array, the class attribute `upc_sequence` and `estimate_config` (after adding entries needed for the estimation) as arguments (see Section 24.5.6 for more details). From the returned dictionary, items “estimators”, “standard_errors”, “other_measures” and “other_info” are extracted. After results from all submodels are collected, a `Coefficient` object is created using those extracted values (see Section 24.6.1).

**Output:**

The method returns a tuple of the created `Coefficient` object and a dictionary with one entry for each submodel. Each entry is a dictionary returned by estimation procedure for that submodel.

### 24.4.4 RegressionModel Class

The `RegressionModel` class implements a model based on a regression procedure. In summary, there is a set of observations, whose attributes or/and variables are used to predict a certain outcome, using some coefficients. Those can be also estimated within this framework. See also Section 14.5.

As in the case of `ChoiceModel`, the regression model can be composed by plug-in modules.

**Initialization**

The class is initialized by passing the following arguments:

**regression_procedure** - a fully qualified name of the module/class in which the regression is implemented (see Section 24.5.5). The default value is ‘opus_core.linear_regression’.

**submodel_string** - If model contains submodels, this character string specifies what attribute of the observation set determines those submodels. Default is None.

**run_config** - A collection of additional arguments that control a simulation run. It should be a dictionary or an instance of class `Resources`. Default is None.

**estimate_config** - A collection of additional arguments that control an estimation run. It should be a dictionary or an instance of class `Resources`. Default is None.

**dataset_pool** - A pool of datasets needed for computation of variables. Default is None.
From the argument `regression_procedure` the initialization method creates a class attribute `regression`, using `RegressionModelFactory` (see Section 24.5). The remaining arguments are set as class properties.

The Run Method

The `run()` method runs the simulation on basis of a given specification and coefficients.

**Input:**

- **specification** - an instance of class `EquationSpecification` specifying variables to be used in the simulation.
- **coefficients** - an instance of class `Coefficients` that contains values of coefficients to be used in the simulation.
- **dataset** - a `Dataset` representing the whole set of observations.
- **index** - an index within `dataset` determining for which observations the prediction is to be made. If it is not given, the whole `dataset` is considered.
- **chunk_specification** - a dictionary specifying how to determine the number of chunks to run the model in. It is passed to the `run` method of `ChunkModel` (see Section 24.4.2).
- **data_objects** - a dictionary containing additional datasets needed for computing variables. This argument is obsolete – the datasets should be included in the `dataset_pool` argument of the constructor.
- **run_config** - additional `Resources` (or dictionary) for controlling the simulation run.
- **initial_values** - an array of initial values of the results. It is of the same size as `dataset`. Elements that are handled by the model (determined by `index` and specification) will be overwritten by the results. By default, the array is set to zeros.
- **procedure** - a fully qualified name of the module/class in which the regression is implemented (see Section 24.5.5). If it is None (default), the value of `regression_procedure` from the constructor is taken. This argument overwrites the class attribute `regression`.

**Algorithm:**
The algorithm is implemented in the method `run_chunk()` called from the parent class `ChunkModel` for each chunk. It invokes a computation of all variables given in the specification. Then for each submodel it creates a data matrix for values corresponding to `index` and invokes the `run()` method of the object stored in the class attribute `regression` (see Section 24.5.5).

**Output:**
The method returns an array of the same size as `index`, determining the outcome of the regression for each observation included in `index`.

The Estimate Method

The `estimate()` method runs an estimation of coefficients on basis of a given specification.

**Input:**

- **specification** - an instance of class `EquationSpecification` specifying variables to be used in the estimation.
- **dataset** - a `Dataset` representing the whole set of observations.
outcome_attribute  - a character string determining the dependent variable (a fully qualified name).

index - an index within the dataset determining which observations will be used for the estimation. If it is not given, the whole dataset is considered.

procedure - a character string giving the fully qualified name of the estimation procedure (see Section 24.5.6). This argument can be also passed via estimate_config as an entry 'estimation'. The default value is None.

data_objects - a dictionary containing additional datasets needed for computing variables. This argument is obsolete – the datasets should be included in the dataset_pool argument of the constructor.

estimate_config - additional Resources (or dictionary) for controlling the estimation run.

Algorithm:
In addition to index, the number of dataset members entering the estimation can be controlled by an entry 'estimation_size_agents' in estimate_config which should have a value between 0 and 1. It gives the portion of index that will be used in the estimation. If it is less than 1, the indices are randomly sampled. In addition, an entry 'submodel_size_max' of estimate_config controls the maximum number of records within each submodel. If a submodel is larger that this number, records are randomly sampled to meet this threshold.

The method invokes computation of variables given in the specification as well as of the outcome attribute. For each submodel, it creates the corresponding data matrix and invokes the run() method of the module given by the argument procedure, passing data, the class attribute regression and estimate_config (after adding entries needed for the estimation) as arguments (see Section 24.5.6 for more details). From the returned dictionary, items "estimators", "standard_errors", "other_measures" and "other_info" are extracted. After results from all submodels are collected, a Coefficient object is created using those extracted values.

Output:
The method returns a tuple of the created Coefficient object and a dictionary with one entry for each submodel. Each entry is a dictionary returned by estimation procedure for that submodel.

Child Class RegressionModelWithInitialResiduals

This class impacts only the method run() (it doesn’t change the behaviour of the estimate() method). It computes initial errors of the observations to the predictions (residuals) if run for the first time. The error values are added to dataset as a primary attribute (with the same name as the outcome attribute adding the prefix '_init_error_'). Then in any run, the error is added to the outcome. Thus, in order to compute the residuals, the outcome attribute must be a known attribute of dataset prior to running the model. Note that running this model on the same set of observations which were used for the estimation, should result in the same outcome as the original values of the outcome attribute.

In addition to all parents arguments, this class requires the argument outcome_attribute to be passed into the constructor. The following entries of the run_config dictionary are accepted:

- 'exclude_missing_values_from_initial_error' - default is False.
- 'outcome_attribute_missing_value' - if the above entry is True, this value determines a missing value. Default is 0.
- 'exclude_outliers_from_initial_error' - default is False.
- 'outlier_is_less_than' - lower bound for excluding outliers from computing residuals.
- 'outlier_is_greater_than' - upper bound for excluding outliers from computing residuals.
24.4.5 SimpleModel Class

This model is an alternative way of computing a variable on a dataset, except that the resulting attribute becomes a primary attribute (see also Section 14.2).

The run() method takes the following arguments:

- **dataset**: an object of class Dataset for which the computation is done.
- **expression**: any variable or expression that can be computed on dataset.
- **outcome_attribute**: name of the outcome attribute. If it is None (default), alias of the expression is taken.
- **dataset_pool**: pool of datasets passed into the computation. Default is None.

The run() method computes the given expression on the given dataset. If the outcome_attribute already exists on the dataset, it is deleted. Results from the computation are assigned to outcome_attribute and the attribute is marked as primary. Values of this attribute are returned.

24.4.6 AllocationModel Class

The AllocationModel allocates given quantity for a dataset according to weights while meeting capacity restrictions (see Section 14.4).

The run() method takes the following arguments:

- **dataset**: an object of class Dataset for which the allocation is done.
- **outcome_attribute**: name of the resulting attribute.
- **weight_attribute**: an attribute/variable/expression of dataset which determines weights for the allocation.
- **control_totals**: an object of class Dataset holding data with the total amount of quantity to be allocated.
- **current_year**: integer value used for filtering out control totals for the current year.
- **control_total_attribute**: name of the attribute in control_totals that specifies the total amount to be allocated. If it is not given, the value of outcome_attribute is taken.
- **year_attribute**: name of the attribute in control_totals on which the filtering of the current year is done. Default is 'year'.
- **capacity_attribute**: name of the attribute/variable/expression in dataset specifying capacity for each member. Default is None which means there are no capacity restrictions.
- **add_quantity**: if True and the outcome_attribute exists in dataset, the resulting values are added to the current values of outcome_attribute. Default is False.
- **dataset_pool**: pool of datasets used in the computation. Default is None.

In addition to the year_attribute and control_total_attribute, the control_totals dataset can contain other attributes that must be known to the dataset (such as a geography), here called 'common' attributes. For each row of the control_totals that matches the current year, the total amount is distributed among members of dataset that have the same values of all common attributes as the row. The distribution is done using the given weights. If the capacity_attribute is given, the algorithm removes any allocations that exceed
the capacity and re-distributes it among remaining members. The resulting values are appended to \texttt{dataset} as \texttt{outcome\_attribute} which is marked as primary. If \texttt{add\_quantity} is True, the resulting values are added to the current values of \texttt{outcome\_attribute} if it exists. The \texttt{outcome\_attribute} is then flushed to cache and the values are returned.

24.4.7 JoinAttributeModificationModel Class

The model modifies an attribute of a dataset for members that are determined by given ids of a second dataset. The \texttt{run()} method takes the following arguments:

\begin{itemize}
  \item \texttt{dataset} - an object of class \texttt{Dataset} for which the modification is done.
  \item \texttt{secondary\_dataset} - an object of class \texttt{Dataset} used to specify members of \texttt{dataset} to be modified.
  \item \texttt{index} - index of members of the \texttt{secondary\_dataset} to be used. If it is \texttt{None}, all members are considered.
  \item \texttt{attribute\_to\_be\_modified} - name of an attribute of \texttt{dataset} to be modified. If it is \texttt{None} (default), an attribute of the same name as the id name of \texttt{secondary\_dataset} is taken.
  \item \texttt{value} - constant to be assigned to the selected members. Default value is 0.
  \item \texttt{filter} - name of an attribute/variable/expression used for filtering out members of the \texttt{secondary\_dataset}. Only members that correspond to value of the filter larger than 0 and are included in \texttt{index} are considered. Default is \texttt{None} which means no filtering.
  \item \texttt{dataset\_pool} - pool of datasets used in the computation. Default is \texttt{None}.
\end{itemize}

\texttt{dataset} must contain an attribute of the same name as the id attribute of the \texttt{secondary\_dataset}, here called the ‘join attribute’. The model finds members of \texttt{dataset} for which the values of the join attribute match the values in the \texttt{secondary\_dataset} (possibly restricted by the \texttt{index} and/or \texttt{filter}). For all those members the attribute \texttt{attribute\_to\_be\_modified} is changed to \texttt{value}.

24.5 Model Components

Opus models are designed in a highly modular way. It allows to easily change model behavior by exchanging components the model is composed from.

Components implemented as classes in \texttt{opus\_core} are shown in their hierarchical structure in Figure 24.4. The shaded boxes are classes that do not provide much functionality themselves, but rather serve as abstract classes. All model components should have a method \texttt{run()}. Model components can be composed from other model components. Thus, Opus models (child classes of \texttt{Model}) are themselves model components.

Instances of model components can be created using the class \texttt{ModelComponentCreator}, by passing the class name into the method \texttt{get\_model\_component()}. In such a case the class name must be the same as the module name. If it is not the case, one can use \texttt{ClassFactory} for this purpose. There are a few child classes of \texttt{ModelComponentCreator} implemented in \texttt{opus\_core} for creating specific model components.
24.5.1 Utilities Class

The Utilities class is used for computing utilities in discrete choice modeling. In opus_core, there is a Utilities class implemented, called linear_utilities, which computes linear utilities:

\[ U_{ni} = \sum_{j=1}^{J} \beta_{ij} x_{nij}. \]

Here \( i = 1, \ldots, I \) denotes the \( I \) different alternatives, \( n \) is an index for observations (agents), \( x \) denotes values of \( J \) variables (including constants) for each agent and alternative, and \( \beta \) is a coefficient matrix of size \( I \times J \). The run() method gets a 3-d array of data and a 2-d array of coefficients as arguments and returns a 2-d array of utilities.

A child class of linear_utilities, called linear_utilities_diagnose, serves analyzing changes in utilities due to changes in the data. For each variable set to its 5%- and 95%-quantile, respectively, while keeping the remaining variables at their median, it runs the linear_utilities module. Results are stored in a file. First line of the file corresponds to the 5%-quantile, second line to the 95%-quantile, and third line contains the difference between lines one and two. The name of the file can be passed to the run() method in its argument resources as a dictionary key 'utilities_diagnose_file'. Default name is 'util'.
24.5.2 Probabilities Class

The Probabilities class is an abstract class for computing probabilities in the discrete choice modeling framework. Given utilities $U$, the opus_core class mnl_probabilities computes multinomial logit probabilities:

$$P_{ni} = \frac{e^{U_{ni}}}{\sum_j e^{U_{nj}}}.$$ 

Thus, the method run() takes a 2-d array of utilities as an argument (number of agents $\times$ number of alternatives) and returns a 2-d array of probabilities (the same shape as utilities).

24.5.3 Choices Class

Choices is an abstract class for selecting choices according to given probabilities in the discrete choice modeling framework. Opus package opus_core supports two classes in this category. random_choices returns an index of randomly selected choices, one per observation. The number of alternatives is simply derived from the second dimension of the array of probabilities, which is passed as argument to the run() method. random_choices_from_index allows one to define an index array whose elements are returned as the selected choices. This index array should be contained in the argument resources (dictionary) as an entry index. This can be useful for example if we are not dealing with the whole set of alternatives, but rather with a subsampled set.

24.5.4 upc_sequence Class

In the discrete choice modeling framework, there is a certain order of steps that need to be evaluated, namely computing utilities, computing probabilities and selecting choices. This class allows to perform these steps using just one method call.

The class upc_sequence is composed by an object of Utilities, an object of Probabilities and an object of Choices. The objects are passed to the constructor. Alternatively, one can use the UPCFactory class, which creates an upc_sequence object from names (character strings) of the components.

The run() method calls the run() methods of the component classes in the order given above, passing results from one class to the next one as input values. Thus, the method takes a 3-d array of data and a 2-d array of coefficients as arguments and returns an array of choices. Any of the components can be eliminated by setting it to None. In such a case, the components receive results of the previously running component as input and the results of the very last running method is the return value of the run() method of upc_sequence.

24.5.5 Regression Class

The Regression class is an abstract class for user defined classes that can be used in the regression model. The class linear_regression implemented in opus_core computes outcome $Y$ as a linear combination of given data $X$ and coefficients $\beta$: $y_n = \sum_{j=1}^{J} \beta_j x_{nj}$. Here, $J$ denotes the number of variables entering the regression and $n$ is an index for observations. The run() method takes a 2-d array of data (of size number of observations $\times$ number of variables) and a 1-d array of coefficients as arguments and returns a 1-d array of outcome.

A child class of linear_regression, called linear_regression_with_normal_error, adds normally distributed random errors to the outcome: $y_n = \sum_{j=1}^{J} \beta_j x_{nj} + \delta_n$ where $\delta_n \sim N(\mu_n, \sigma_n^2)$. $\mu_n$ and $\sigma_n^2$, respectively, can be passed to the run() method in its argument resources (dictionary) as entries 'linear_regression_error_mean' and 'linear_regression_error_variance', respectively. Both can be specified either as a single value or as an array of size number of observations. By default, $\delta_n \sim N(0,1)$ for all $n$. 

220 Chapter 24. The opus_core Opus Package
24.5.6 EstimationProcedure Class

This class is an abstract class for modules that implement estimation of coefficients for one of the available models. **bhhh_mnl_estimation** implements the BHHH estimation algorithm for multinomial logit models and can be plugged into the ChoiceModel. As arguments, it gets a data array (of size number of observations \( \times \) number of alternatives \( \times \) number of variables) and an object **upc_sequence**. It uses the classes **Probabilities** and **Utilities** contained in **upc_sequence** for the maximum likelihood estimation. This assures that if **bhhh_mnl_estimation** is plugged into the **estimate()** method of ChoiceModel (Section 24.4.3), the model will be estimated by using the same code for computing utilities and probabilities as the **run()** method. The third argument of the **run()** method of this class is of type **Resources** and must contain an entry **selected_choice** which is a 0-1 matrix of size number of observations \( \times \) number of alternatives. For each agent, it contains a 1 on a position of the chosen alternative, otherwise 0s. Note that ChoiceModel prepares and passes this matrix automatically.

A child class of **bhhh_mnl_estimation**, called **bhhh_wesml_mnl_estimation**, implements the Weighted Endogenous Sampling Maximum Likelihood procedure, proposed by Manski, Lerman 1977. Here, the data are weighted by correction weights (observation share/sampled share) in order to take into account undersampled or oversampled observations. The correction weights should be implemented as a variable. Its fully-qualified name is passed to the **run()** method in the argument **resources** (dictionary) as an entry ‘wesml_sampling_correction_variable’.

Classes **bhhh_mnl_estimation_with_diagnose** and **bhhh_wesml_mnl_estimation_with_diagnose**, respectively, run the estimation of their parent classes, namely **bhhh_mnl_estimation** and **bhhh_wesml_mnl_estimation**, respectively, using the utilities component **linear_utilities_diagnose** (see Section 24.5.1).

**estimate_linear_regression** performs a parameters estimation via the least squares method. As arguments, it gets a data array (of size number of observations \( \times \) number of variables), an instance of class **Regression** (not used in this module) and an object **Resources**. The last argument must contain an entry **outcome** which is a 1-d array of an outcome for each observation. This class can be plugged into the RegressionModel which takes care of all arguments.

**estimate_linear_regression_r** estimates the parameters using the R function `lm`. Here, the rpy module is required. It should give the same results as **estimate_linear_regression**.

The estimation modules return a dictionary with several entries: Entries **estimators** and **standard_errors** contain arrays of estimated coefficients and their standard errors, respectively. An entry **other_measures** is a dictionary which should contain additional measures of the estimates, i.e. their values should be arrays of the same size as estimators. The two estimation modules in **opus_core** return here one entry, namely the **t_statistic**. The last entry in the dictionary returned by the modules, **other_info**, is a dictionary containing additional information about the estimation. Its values don’t follow any restriction on type and size. Thus, these can be also single values, such as likelihood ratio test statistics, degrees of freedom, \( R^2 \) etc.

Opus also implements a tool for variable selection in linear regression. Class **bma_for_linear_regression_r** uses the R package BMA. It prints out results computed by the R function `bic.glm` and plots an image of the results. The input arguments are identical to those in **estimate_linear_regression**. Additionally, if the dictionary **resources** contains an entry ‘bma_imageplot_filename’, the resulting imageplot is stored as a pdf file of that name. The **run()** method does not return any value. It should serve users as a tool to select variables which can be then plugged into **estimate_linear_regression**. The module rpy is required when using this component.

24.5.7 Sampler Class

The **Sampler** class is an abstract class for sampling alternatives for agents. It returns 2-d array with rows representing agents and columns representing sampled alternatives. It is not directly used in any **opus_core** model, but it is a building block that can be used in models of other packages. For example, we made a heavy use of this component in the urbansim package for creating sampled alternative set for agents in ChoiceModel.
Weighted Sampling

`weighted_sampling` is a child class of `Sampler` class. It randomly samples alternatives from choice population with probability proportion to their weights. Its `run()` method accepts the following arguments:

- **dataset1** - an instance of `Dataset`, to be used as agent set.
- **dataset2** - an instance of `Dataset`, to be used as choice set.
- **index1** - indices of `dataset1` for whom alternatives are sampled. If it is not given, all elements of `dataset1` are used.
- **index2** - indices of `dataset2` from which alternatives are sampled. If it is not given, all elements of `dataset2` are used.
- **sample_size** - number of alternatives sampled.
- **weight** - an array used as weight for elements of `dataset2` in unequal probability sampling; it has to be either of None, or of the same size as `index2` or `dataset2`. If it is not given, sampling is proceeded with equal probability.
- **include_chosen_choice** - whether agents’ chosen choice will be included in the return results. If it’s true, the chosen choices are in the first column of the return results.
- **resources** - an instance of `Resources` that can be used to pass any of the above arguments to the `run` method.

Stratified Sampling

`stratified_sampling` is a child class of `Sampler` class. It randomly samples alternatives from choice population according to their stratum setting. Its `run()` method accepts the following arguments:

- **dataset1** - an instance of `Dataset`, to be used as agent set.
- **dataset2** - an instance of `Dataset`, to be used as choice set.
- **index1** - indices of `dataset1` for whom alternatives are sampled. If it is not given, all elements of `dataset1` are used.
- **index2** - indices of `dataset2` from which alternatives are sampled. If it is not given, all elements of `dataset2` are used.
- **stratum** - an array indicates the stratum id for elements of `dataset2`; it has to be either of None, or of the same size as `index2` or `dataset2`. If it’s not given, all elements are treated as in 1 stratum.
- **weight** - like in `weighted_sampling`, weight is an array used as weight for elements of `dataset2` in unequal probability sampling; it has to be either of None, or of the same size as `index2` or `dataset2`. If it is not given, sampling is proceeded with equal probability.
- **sample_size** - number of alternatives sampled from one stratum; default value is 1.
- **sample_size_from_chosen_stratum** - number of alternatives sampled from agent’s chosen stratum. If it’s None, it’s equal to value specified by `sample_size` or `sample_rate`.
- **sample_rate** - calculate number of alternatives sampled from one stratum by multiplying this rate with number of observations in this stratum. If both `sample_rate` and `sample_size` are specified, use `sample_rate`.
- **include_chosen_choice** - whether agents’ chosen choice will be included in the return results. If it’s true, the chosen choices are in the first column of the return results.
- **resources** - an instance of `Resources` that can be used to pass any of the above arguments to the `run` method.

Both sampling classes return a tuple where the first element is the sampled index, and the second element is the index within the sampled index indicating the chosen choice.
24.6 Specification and Coefficients

24.6.1 Specification

Often, models are specified by a set of variables. These are connected to a set of coefficients calibrated to the observed data. Opus defines a class for such specification, called `EquationSpecification`.

Initialization

The constructor of `EquationSpecification` takes the following arguments (they all have default value None):

- **variables** - an array of variable names.
- **coefficients** - an array of coefficient names.
- **equations** - an array of equations.
- **submodels** - an array of submodels.
- **fixed_values** - an array of fixed values. Any non-zero value is considered as a constant value of the corresponding coefficient, i.e. it is not to be estimated.
- **other_fields** - a dictionary holding additional columns of the specification table.
- **specification_dict** - the specification is specified in a dictionary format (see below). This argument is considered only if the argument **variables** is None. In such a case, all arguments above are ignored.
- **in_storage** - an object of class `Storage` for loading specification from a storage.
- **out_storage** - an object of class `Storage` for specification output.

All arguments are set as class properties.

The arrays **variables** and **coefficients** must have the same size. If submodels, equations and fixed_values are not omitted, they too must have the same length as **variables**. It is interpreted as the \( i \)-th variable is connected to the \( i \)-th coefficient in the \( i \)-th equation (if there are any) in the \( i \)-th submodel (if there are any). If the \( i \)-th fixed value is non-zero, the \( i \)-th coefficient is not to be estimated. All entries of **other_fields** must also be of the same size as **variables**. Values of equations and submodels should be strictly positive integers. Coefficients should have different names across equations, i.e. if there would be \( i \) and \( j \) for which the \( \text{coefficients}[i]==\text{coefficients}[j] \) and \( \text{equations}[i]<>\text{equations}[j] \) and \( \text{submodels}[i]==\text{submodels}[j] \), it would lead to errors when connecting specification and coefficients.

An alternative way of defining a specification is a dictionary format (passed to the constructor via the argument **specification_dict**). Keys of the dictionary are submodels. If there is only one submodel, the value \(-2\) should be used as key. The value for each submodel entry is a list containing specification for the particular submodel. The elements of each list can be defined in one of the following forms:

- a character string specifying a variable in its fully qualified name or as an expression. In such a case, the coefficient name is determined by the alias of the variable.
- a tuple of length 2: variable name as above, and the corresponding coefficient name.
- a tuple of length 3: variable name, coefficient name, fixed value of the coefficient (if the coefficient should not be estimated).
• a dictionary with pairs variable name, coefficient name

`specification_dict` can contain an entry `'_definition_'` which should be a list of elements in one of the forms above. In such a case, the entries defined for submodels can contain only the variable aliases. The corresponding coefficient names and fixed values (if defined) are taken from the definition section. Examples of specifications in a dictionary format can be found in the unit tests of the `EquationSpecification` class.

Loading from and Writing into Storage

If the specification is stored in one of the supported storage formats, one can omit all arguments in the constructor and load the specification from the storage, using the method `load()`. It takes the following arguments:

- `resources` - an object of class `Resources`. If the remaining arguments are given, they will have priority over entries of the same name in `resources`.
- `in_storage` - an object of class `Storage` that overwrites the one given in the constructor.
- `in_table_name` - name of the table/file where the specification should be loaded from.
- `variables` - if this argument is given, it serves as a filter for the variables loaded from the storage.

For each of the class properties variables, coefficients, equations, submodels, and fixed_values, respectively, a table column is accepted on the storage (the first two are required, the others are optional). The column names are given in the `resources` entries `field_variable_name`, `field_coefficient_name`, `field_equation_id`, `field_submodel_id`, and `field_fixed_value`, respectively. Default values for the column names are `variable_name`, `coefficient_name`, `equation_id`, `sub_model_id`, and `fixed_value`, respectively. If the table contains columns of other names, they are loaded into the class attribute `other_fields`, each as a dictionary entry.

To store a specification into a storage, use the method `write(resources, out_storage, out_table_name)`. The behavior is analogous to the `load()` method. If equations or/and submodels are not used, the method stores values $-2$ in those columns.

### 24.6.2 Coefficients

Coefficients can be managed by the class `Coefficients`. Its behavior is similar to the one of `EquationSpecification`.

**Initialization**

The constructor takes the following arguments:

- `names` - an array of coefficient names.
- `values` - an array of coefficient values (of the same size as `names` if not empty).
- `standard_errors` - an array of standard errors of coefficients (of the same size as `names` if not empty).
- `submodels` - an array of submodels (of the same size as `names` if not empty).
- `in_storage` - an object of class `Storage` for loading coefficients from a storage.
**out_storage** - an object of class `Storage` for coefficients output.

**other_measures** - a dictionary for other coefficient measures, such as t-values. Keys are the names of the measures, values are arrays or of the same size as `names`.

**other_info** - dictionary storing other information about the coefficients, such as goodness of fit values.

The arguments are interpreted as the coefficient of $i$-th name has the $i$-th value, optionally the $i$-th standard error and is used in $i$-th submodel. This also applies to each entry in `other_measures`. Note that since equations are not used here, there has to be coefficients with different names for different equations, defined in the specification.

All arguments are set as class properties.

### Loading from and Writing into Storage

The method `load()` is similar to the one defined in `EquationSpecification`. It takes the arguments:

- **resources** - an object of class `Resources`. If the remaining arguments are given, they will have priority over entries of the same name in `resources`.

- **in_storage** - an object of class `Storage` that overwrites the one given in the constructor.

- **in_table_name** - name of the table/file where the coefficients should be loaded from.

For each of the class properties names, values, standard errors, and submodels, respectively, a table column is expected on the storage. The column names are given in the `resources` entries 'field_coefficient_name', 'field_estimate', 'field_standard_error', and 'field_submodel_id', respectively. Default values for the column names are 'variable_name', 'coefficient_name', 'equation_id', and 'sub_model_id'. If there are other fields in the table, their column names should be given in the entry 'other_fields' of `resources` which is a list of character strings. The default value is ['t_statistic', 'p_value'].

To store coefficients into a storage, use the method `write(resources, out_storage, out_table_name)`. The behavior is analogous to the `load()` method.

The class also allows to create a coefficient table in the L\LaTeX{} format. Method `make_tex_table()` accepts as arguments the file name (without '.tex'), optionally the directory path and headers for each column.

### 24.6.3 Specified Coefficients

In order to connect a specification with coefficients, Opus uses the class `SpecifiedCoefficients`. Its method `create()` takes an instance of class `Coefficients` and an instance of class `EquationSpecification` and creates arrays of coefficient values, standard errors and other measures. The shape of those arrays is such that they can be easily combined with data arrays when connecting coefficients to variable values. In particular, they have three dimensions, number of equations, number of variables and number of submodels.

For working with single submodels, there is a child class of `SpecifiedCoefficients`, called `SpecifiedCoefficientsFor1Submodel`, that is initialized by passing the parent object and the submodel number.

Those classes are created and used by two `opus_core` models, namely regression model for computing the regression and the choice model for computing the utilities.
24.7 Other Classes

24.7.1 Dataset Pool

A class called DatasetPool is designed to maintain a 'pool' of datasets. It is mainly used when computing variables. Therefore in most cases, it will be only needed if the Dataset method compute_variables() is used (for which an instance of DatasetPool is passed as an argument) in order to have external datasets available that are required by the various variable implementations (see an example in Section 23.3.1).

The class is initialized by passing arguments:

- **package_order** - a list of packages that are used for finding the corresponding dataset class. Default is an empty list.
- **package_order_exception** - a dictionary of exceptions from the package_order as pairs dataset name and list of packages for that dataset. Default is an empty dictionary.
- **storage** - an object of class Storage that contains data of datasets to be included in pool. Default is None.
- **datasets_dict** - a dictionary containing pairs of dataset name and dataset object. Default is an empty dictionary.

One can add a dataset into the pool using the method

```python
_add_dataset(dataset_name, dataset),
```

or by passing multiple datasets in one dictionary using the method

```python
add_datasets_if_not_included(datasets_dict).
```

A dataset can be accessed by the method

```python
get_dataset(dataset_name, dataset_arguments)
```

where the second argument is optional. If a dataset of the given name is included in the pool, it is returned. Otherwise, the class creates a Dataset object using the storage passed to the constructor. In order to create the appropriate Dataset class, it is searched for a module called `dataset_name_dataset.py` which should contain a class called `DatasetNameDataset`. The DatasetPool class searches for this module in the directory 'datasets' of packages given in the initialization argument 'package_order'.

24.7.2 ModelGroup and ModelGroupMember

These two classes are designed to be used in models that are considered as model groups, i.e. models that are to be run multiple times, each time on different subsets of one dataset.

A model group is defined by the following:

- There must be a dataset whose one attribute defines the names of the group members. These names must be unique within the dataset. We will call this dataset and its attribute grouping dataset and grouping attribute respectively. Values of the unique identifier of the grouping dataset will be called member codes.

- A dataset that is going to be subset for running the model member on must contain an attribute whose values represent the member codes. We will call this attribute agents grouping attribute.

Example: Suppose we would like to run a model on a 'job' dataset, subset according to its building type. Then our grouping dataset, say 'job_building_type', can contain the following data:
id | name
---|------
1  | 'commercial'
2  | 'industrial'
3  | 'residential'

The attribute ‘name’ is our grouping attribute, values of the ‘id’ attribute are member codes. The dataset ‘job’ contains an attribute, say ‘building_type’ that has value 1 for all commercial jobs, 2 for all industrial and 3 for all residential jobs. ‘building_type’ is then the agents grouping attribute.

The class ModelGroup is initialized by

- **dataset** - object of class Dataset that is the grouping dataset.
- **grouping_attribute** - name of the grouping attribute.

The class offers useful methods for accessing member names and codes of this group.

The class ModelGroupMember is initialized by

- **model_group** - object of class ModelGroup.
- **member_name** - name of this specific member. It must be contained in the grouping attribute of ‘model_group’.

A model that uses this class should use the method `set_agents_grouping_attribute(attribute)` to set the agents grouping attribute. Then the class can be used for subsetting any dataset according to the member codes, e.g. by the method `get_index_of_my_agents(dataset, index)` which selects those entries from the given index that correspond to this group member.

### 24.7.3 Configuration Classes

Opus implements a few classes that are used for configuration of objects.

- **GeneralResources** – a Python dictionary with some additional methods.
- **SessionConfiguration** – a singleton class (subclass of GeneralResources) that is to be configured with global settings for a user’s session. Requires parameter `in_storage` to not be `None` if creating a new instance, so SessionConfiguration knows from where to get the data for any dataset it creates.
- **Configuration** – a child of GeneralResources that implements a hierarchical representation of the user-specified parameters and settings.
- **Resources** – a child of Configuration. It has access to the SessionConfiguration.
- **ChunkSpecification** – class for configuring chunks used by the ChunkModel (see Section 24.4.2). It is initialized by a dictionary containing either the key ‘records_per_chunk’ or ‘nchunks’ with the appropriate value. Method `nchunks()` returns number of chunks. Method `chunk_size()` returns the number of records in one chunk.
25.1 Introduction

This chapter describes the different components — datasets and models — contained in the urbansim package.

25.2 Datasets

All datasets defined in urbansim are implemented as children of the Opus opus_core class Dataset described in Section 24.1. Each dataset sets default values for several class properties, such as a name of the unique identifier (id_name), dataset name (dataset_name), in_table_name and out_table_name. For example, the following code

```python
>>> from opus_core.datasets.dataset import Dataset
>>> households = Dataset(in_storage = storage,
                            id_name='household_id',
                            dataset_name='household',
                            in_table_name='households',
                            out_table_name='households')
```

is equivalent to

```python
>>> from urbansim.datasets.household_dataset import HouseholdDataset
>>> households = HouseholdDataset(in_storage = storage)
```

The urbansim datasets are defined in urbansim.datasets. We use the following naming convention: For module name we use the dataset name in lower case in singular form, where single words are connected by ‘_’, and ending with “_dataset”. For class name we capitalize the first letters in each word of the dataset name, use singular form and add ‘Dataset’ at the end. For example, a dataset for development projects is defined in class DevelopmentProjectDataset implemented in the module ‘development_project_dataset.py’. For interaction sets, we connect the two dataset names in the same way, but with an ‘x’ in the module name and an ‘X’ in the class name. For example, an interaction set of development projects and gridcells is defined in DevelopmentXProjectGridcellDataset implemented in ‘development_project_x_gridcell_dataset.py’.

228
25.2.1 Predefined Datasets

Table 25.1 lists dataset classes that are predefined in urbansim (in alphabetical order), including the default value for \texttt{dataset\_name}, \texttt{in\_table\_name}, \texttt{id\_name} and name of the module in which the class is implemented (excluding ".py"). In most cases they correspond to database tables described in Chapter 18. The corresponding table name is the value of \texttt{in\_table\_name}.

In addition, they are a few interaction sets defined in urbansim, listed in Table 25.2. The dataset name of such interaction set is created automatically from the names of the two interacting datasets. Note that for standard usage, there is no need to explicitly define an interaction set for every two specific datasets. If a specific definition is not found, Opus framework creates an abstract interaction set using the \texttt{InteractionDataset} in \texttt{opus\_core} (see Section 24.1.8).

25.3 Variables

Variables predefined in urbansim are structured according to the dataset to which they belong. They are implemented in directories of the same name as \texttt{dataset\_name} which is placed in the top directory of urbansim. For example, all gridcell variables are placed in 'urbansim/gridcell', all interaction variables of households and gridcells are placed in 'urbansim/household\_x\_gridcell'. All variables are defined as children of the \texttt{opus\_core} class \texttt{Variable} or as expressions in an ‘alias.py’ file, and thus, they follow the guidelines presented in Section 24.3.

25.4 Models

urbansim has a set of predefined models, each of them implemented as a child of one of the \texttt{opus\_core} models described in Section 24.4. They are hierarchically structured and a few of them are created using specific creators (see Figure 25.1). All urbansim models are placed in ‘urbansim/models’.

The models to execute when running UrbanSim are determined by the configuration (see Section 22.3.2). Its entry “models” is a list of character strings that identify the models to run in that order (see for example the list of models for running the gridcell version of UrbanSim on page 151). As discussed in Section 22.3.3, the controller of each model points to one of the model classes shown in Figure 25.1 and configures their inputs and outputs. In the following sections we describe each of the model classes in detail.

25.4.1 Land Price Model and Corrected Land Price Model

The \texttt{LandPriceModel} predicts prices of land, in particular the residential and non-residential land value for each member of the given location set. The prediction is done via a regression procedure and thus, the model is a child of the \texttt{RegressionModel} described in Section 24.4.4.

Initialization

The class is initialized by passing the following arguments:

\texttt{regression\_procedure} - a fully qualified name of the module/class in which the regression is implemented (see Section 24.5.5). Default value is “opus\_core.linear\_regression”.
<table>
<thead>
<tr>
<th>dataset_name</th>
<th>in_table_name (default)</th>
<th>id_name (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>building</td>
<td>buildings</td>
<td>building_id</td>
</tr>
<tr>
<td>building_type</td>
<td>building_types</td>
<td>building_type_id</td>
</tr>
<tr>
<td>city*</td>
<td>cities</td>
<td>city_id</td>
</tr>
<tr>
<td>county*</td>
<td>counties</td>
<td>county_id</td>
</tr>
<tr>
<td>development_constraint</td>
<td>development_constraints</td>
<td>constraint_id</td>
</tr>
<tr>
<td>development_event</td>
<td>development_events</td>
<td>grid_id, scheduled_year</td>
</tr>
<tr>
<td>development_event_history</td>
<td>development_event_history</td>
<td>grid_id, scheduled_year</td>
</tr>
<tr>
<td>development_group</td>
<td>development_type_groups</td>
<td>group_id</td>
</tr>
<tr>
<td>development_type</td>
<td>development_types, development_type_group_definitions</td>
<td>development_type_id</td>
</tr>
<tr>
<td>employment_control_total</td>
<td>annual_employment_control_totals</td>
<td>year, sector_id</td>
</tr>
<tr>
<td>employment_sector</td>
<td>employment_sectors</td>
<td>sector_id</td>
</tr>
<tr>
<td>employment_sector_group</td>
<td>employment_adhoc_sector_groups</td>
<td>group_id</td>
</tr>
<tr>
<td>faz*</td>
<td>fazes</td>
<td>faz_id</td>
</tr>
<tr>
<td>fazdistrict*</td>
<td>fazdistrict_id</td>
<td>grid_id</td>
</tr>
<tr>
<td>gridcell*</td>
<td>gridcells</td>
<td>household_id</td>
</tr>
<tr>
<td>household</td>
<td>households</td>
<td>–</td>
</tr>
<tr>
<td>household_characteristic</td>
<td>household_characteristics_for_ht</td>
<td>–</td>
</tr>
<tr>
<td>household_control_total</td>
<td>annual_household_control_totals</td>
<td>year</td>
</tr>
<tr>
<td>job</td>
<td>jobs</td>
<td>job_id</td>
</tr>
<tr>
<td>job_building_type</td>
<td>job_building_types</td>
<td>id</td>
</tr>
<tr>
<td>large_area*</td>
<td>large_areas</td>
<td>large_area_id</td>
</tr>
<tr>
<td>neighborhood*</td>
<td>neighborhoods</td>
<td>neighborhood_id</td>
</tr>
<tr>
<td>plan_type</td>
<td>plan_types</td>
<td>plan_type_id</td>
</tr>
<tr>
<td>plan_type_group</td>
<td>plan_type_groups</td>
<td>group_id</td>
</tr>
<tr>
<td>race</td>
<td>race_names</td>
<td>race_id</td>
</tr>
<tr>
<td>rate (households)</td>
<td>annual_relocation_rates_for_households</td>
<td>age_min, income_min</td>
</tr>
<tr>
<td>rate (jobs)</td>
<td>annual_relocation_rates_for_jobs</td>
<td>sector_id</td>
</tr>
<tr>
<td>region</td>
<td>regions</td>
<td>region_id</td>
</tr>
<tr>
<td>ring</td>
<td>rings</td>
<td>ring_id</td>
</tr>
<tr>
<td>target_vacancy</td>
<td>target_vacancies</td>
<td>year</td>
</tr>
<tr>
<td>travel_data</td>
<td>travel_data</td>
<td>from_zone_id, to_zone_id</td>
</tr>
<tr>
<td>urbansim_constant</td>
<td>urbansim_constants</td>
<td>–</td>
</tr>
<tr>
<td>vacant_land_and_building_type</td>
<td>building_types</td>
<td>building_type_id</td>
</tr>
<tr>
<td>zone*</td>
<td>zones</td>
<td>zone_id</td>
</tr>
</tbody>
</table>

Table 25.1: Datasets defined in urbansim. A dataset marked with * is a location set, i.e. it represents a set of locations of a specific geographical unit and can be visualized as a two-dimensional image.

<table>
<thead>
<tr>
<th>dataset_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>development_project_x_gridcell</td>
</tr>
<tr>
<td>household_x_gridcell</td>
</tr>
<tr>
<td>household_x_neighborhood</td>
</tr>
<tr>
<td>household_x_zone</td>
</tr>
<tr>
<td>job_x_gridcell</td>
</tr>
</tbody>
</table>

Table 25.2: Interaction datasets defined in urbansim.
Figure 25.1: Models in urbansim. Solid arrows indicate the inheritance hierarchy: $a \rightarrow b$ means $a$ inherits from $b$. Dashed arrows show relations between creators and models ($a$ creates $b$).

**filter** - name of a variable/attribute used to filter out elements for the regression (applied to both, the run() and estimate() method). Default value is “urbansim.gridcell.is_in_development_type_group_developable” which requires an entry 'developable’ in the table “development_type_groups”. It also requires that the location set is a gridcell set.

**submodel_string** - If model contains submodels, this character string specifies what attribute of the observation set determines those submodels. Default value is “development_type_id”.

**run_config** - A collection of additional arguments that control a simulation run. It should be a python dictionary or of class Resources. Default is None.

**estimate_config** - A collection of additional arguments that control an estimation run. It should be a python dictionary or of class Resources. Default is None.

**dataset_pool** - A pool of datasets needed for computation of variables. Default is None.

The constructor sets filter as a class attribute and calls the parent constructor (see Initialization in Section 24.4.4).

The Run Method

**Input:**
The run() method takes exactly the same arguments as its parent class: specification, coefficients, dataset, index, chunk_specification, data_objects, and run_config.
Algorithm:
If filter is given in the initialization, index is updated to only those elements for which the value of attribute/variable filter is larger than zero. It then invokes the run() method of RegressionModel passing all arguments where index is possibly modified. The returned value of this call is considered to be the prediction of the natural logarithm of total land value for each element of dataset included in index. Each of those values is then exponentiated and split into residential and non-residential land value, using the attribute “fraction_residential_land” (this must be a known attribute of dataset). Attributes “residential_land_value” and “nonresidential_land_value” (which also are expected to be known attributes of the dataset) are modified by replacing current values with the computed values.

If any of the computed values exceeds the maximal value of float32, a warning is issued and the value is clipped to the maximal possible value.

Output:
The method returns an index of values within dataset for which the land value was modified.

The Estimate Method

Input:
The estimate() method takes exactly the same arguments as its parent class: specification, dataset, outcome_attribute, index, procedure, data_objects, and estimate_config. The default value for outcome_attribute is “urbansim.gridcell.ln_total_land_value”.

Algorithm:
The method applies the filter (if given) in the same way as the run() method. It then calls the parent method estimate() passing all arguments and the possible modified index.

Output:
The method returns results of estimate() method of RegressionModel (see Section 24.4.4).

Model Configuration

This model is only used in the gridcell version of UrbanSim. For a production run the model is initialized with default values of the model constructor. The run() method is called by passing the following arguments:

- specification - An EquationSpecification object created with data from table “land_price_model_specification”.
- coefficients - A Coefficients object created with data from table “land_price_model_coefficients”.
- dataset - An instance of class GridcellDataset created with data from table “gridcells” (see Section 19.1 for the table structure).
- index - None. Thus, the model runs on all members of dataset which are possibly filtered using a filter passed to the constructor (by default “urbansim.gridcell.is_in_development_type_group_developable”, see the Initialization paragraph).

Corrected Land Price Model

This child model of LandPriceModel makes a correction of the land value to avoid its declination with development type change.
The `run()` method expects an additional argument, `n_simulated_years`, which gives the number of years that have been already simulated. If it is larger than 1, after running its parent’s code, a model called `CorrectLandValue` is invoked. This model determines those members of `dataset` whose development type changed and at the same time have lower total land value when comparing to previous simulated year. Values of the attributes “residential_land_value” and “nonresidential_land_value” of these members are replaced by values of the same attributes from previous year.

25.4.2 Residential Land Share Model

Residential land share model (implemented in class `ResidentialLandShareModel`) predicts what fraction of each location is residential land. It is a `RegressionModel` (Section 24.4.4) with an only method overwritten, the `run()` method.

The Run Method

*Input:*
It takes the same arguments as the `run()` of its parent: `specification`, `coefficients`, `dataset`, `index`, `chunk_specification`, `data_objects`, and `run_config`.

*Algorithm:*
The `run()` method of `RegressionModel` is invoked in order to obtain a regression outcome $y$. If the run was successful, the resulting quantity of this model, $y'$, is computed as

$$ y' = \frac{e^y}{1 + e^y} $$

$y'$ is an array of values for each member of `dataset` given by `index`. Those values are stored in the attribute given by the class property `attribute_to_modify`. This is by default “fraction_residential_land”. If such attribute does not exists in the `dataset`, it is created.

*Output:*
The method returns $y'$.

Model Configuration

The model is used in the gridcell version of UrbanSim only. It is called after the Events Coordinator (see Section 25.4.18). It processes only those locations that were updated by the Events Coordinator. The following arguments are passed to the run:

* `specification` - An `EquationSpecification` object created with data from table “residential_land_share_model_specification”.
* `coefficients` - A `Coefficients` object created with data from table “residential_land_share_model_coefficients”.
* `dataset` - An instance of class `GridcellDataset` that was passed to the Events Coordinator.
* `index` - Indices of the gridcells that were changed by the Events Coordinator, i.e. the first element of its output tuple.
25.4.3 Housing Price Model

The `HousingPriceModel` class predicts housing prices via a regression procedure and thus, the model is a child of the `RegressionModel` described in Section 24.4.4.

Initialization

The class is initialized by passing the following arguments:

- **regression_procedure** - a fully qualified name of the module/class in which the regression is implemented (see Section 24.5.5). Default value is “opus_core.linear_regression”.
- **filter_attribute** - name of a variable/attribute used to filter out elements for the regression (applied to both, the `run()` and `estimate()` method). Default value is “urbansim.gridcell.has_residential_units” which requires an entry ”residential_units” in the ”gridcells” table. It also requires that the regression is done on a gridcell dataset.
- **submodel_string** - If model contains submodels, this character string specifies what attribute of the observation set determines those submodels. Default value is “development_type_id”.
- **run_config** - A collection of additional arguments that control a simulation run. It should be a python dictionary or of class `Resources`. Default is None.
- **estimate_config** - A collection of additional arguments that control an estimation run. It should be a python dictionary or of class `Resources`. Default is None.
- **dataset_pool** - A pool of datasets needed for computation of variables. Default is None.

The constructor sets `filter_attribute` as a class attribute and calls the parent constructor (see Initialization in Section 24.4.4).

The Run Method

**Input:**
The `run()` method takes exactly the same arguments as its parent class: `specification`, `coefficients`, `dataset`, `index`, `chunk_specification`, `data_objects`, and `run_config`.

**Algorithm:**
If `filter_attribute` is given in the initialization, `index` is updated to only those elements for which the value of attribute/variable `filter_attribute` is larger than zero. It then invokes the `run()` method of `RegressionModel` passing all arguments where `index` is possibly modified. The returned value of this call is considered to be the prediction of housing price. `dataset` is expected to have an attribute ”housing_price” which is modified for members given by `index` by the computed values.

**Output:**
The method returns None.

The Estimate Method

**Input:**
The `estimate()` method takes exactly the same arguments as its parent class:
**specification, dataset, outcome_attribute, index, procedure, data_objects, and estimate_config.** The default value for *outcome_attribute* is “housing_price”.

**Algorithm:**
The method applies the *filter_attribute* (if given) in the same way as the *run()* method. It then calls the parent method *estimate()* passing all arguments and the possible modified *index*.

**Output:**
The method returns results of *estimate()* method of *RegressionModel* (see Section 24.4.4).

### 25.4.4 Real Estate Price Model

The *RealEstatePriceModel* class predicts real estate prices via a regression procedure (see Section 17.2.7). In order to avoid jumps in prices for the first simulated year, the initial residuals are added to the regression outcomes. Thus, the model is a child of the *RegressionModelWithInitialResiduals* described in Section 24.4.4 in a paragraph on page 216.

**Initialization**

The class is initialized by passing the following arguments:

- **regression_procedure** - a fully qualified name of the module/class in which the regression is implemented (see Section 24.5.5). Default value is “opus_core.linear_regression”.
- **filter_attribute** - name of a variable/attribute used to filter out elements for the regression (applied to both, the *run()* and *estimate()* method). Default value is None.
- **submodel_string** - If model contains submodels, this character string specifies what attribute of the observation set determines those submodels. Default value is “building_type_id”.
- **outcome_attribute** - A required attribute of the observation dataset. It is used to compute the initial residuals. Default is ’unit_price’. The attribute should be a primary attribute, unless it starts with the string ’ln_’. In such a case the observation set is required to have an attribute matching the *outcome_attribute* without the prefix ’ln_’.
- **run_config** - A collection of additional arguments that control a simulation run. It should be a python dictionary or of class *Resources*. Default is None.
- **estimate_config** - A collection of additional arguments that control an estimation run. It should be a python dictionary or of class *Resources*. Default is None.
- **dataset_pool** - A pool of datasets needed for computation of variables. Default is None.

The constructor sets *filter_attribute* as a class attribute and calls the parent constructor (see Initialization in Section 24.4.4).

**The Run Method**

**Input:**
The *run()* method takes exactly the same arguments as its parent class:
**specification, coefficients, dataset, index, chunk_specification, data_objects, and run_config.**
Algorithm:
If \texttt{filter\_attribute} is given in the initialization, index is updated to only those elements for which the value of attribute/variable \texttt{filter\_attribute} is larger than zero. It then invokes the \texttt{run()} method of \texttt{RegressionModel} passing all arguments where index is possibly modified. The returned value of this call is considered to be the prediction for the given \texttt{outcome\_attribute}. If \texttt{outcome\_attribute} starts with 'ln', the values are exponentiated and stored as a dataset attribute of the same name but without the prefix 'ln'. Otherwise the values of \texttt{outcome\_attribute} are overwritten by the new values. Only values are overwritten that correspond to dataset members given by index.

Output:
The predicted values stored in the dataset are returned.

The Estimate Method

Input:
The \texttt{estimate()} method takes exactly the same arguments as its parent class: \texttt{specification, dataset, outcome\_attribute, index, procedure, data\_objects}, and \texttt{estimate\_config}. The default value for \texttt{outcome\_attribute} is “unit\_price”.

Algorithm:
The method applies the \texttt{filter\_attribute} (if given) in the same way as the \texttt{run()} method. It then calls the parent method \texttt{estimate()} passing all arguments and the possible modified \texttt{index}.

Output:
The method returns results of \texttt{estimate()} method of \texttt{RegressionModel} (see Section 24.4.4).

Model Configuration

In the parcel version of UrbanSim, the model is initialized as follows:

\texttt{submodel\_string} - 'land\_use\_type\_id'

\texttt{outcome\_attribute} - 'ln\_unit\_price=ln(urbansim\_parcel.parcel.unit\_price)'

\texttt{filter\_attribute} - 'numpy.logical_or(urbansim\_parcel.parcel.building\_sqft, urbansim\_parcel.parcel.is\_land\_use\_type\_vacant)'

In the zone version of UrbanSim, the model is initialized as follows:

\texttt{submodel\_string} - 'building\_type\_id'

\texttt{outcome\_attribute} - 'ln\_unit\_price=ln(pseudo\_building.avg\_value)'

Remaining arguments in both cases get the default values of the model constructor.

The \texttt{run()} method is called by passing the following arguments:

\texttt{specification} - An \texttt{EquationSpecification} object created with data from table “real\_estate\_price\_model\_specification”.
coefficients - A `Coefficients` object created with data from table “real_estate_price_model_coefficients”.

dataset - For parcel projects, this is an instance of class `ParcelDataset` created with data from table “parcels”. For zone projects, this is an instance of class `PseudoBuildingDataset`.

index - None. Thus, the model runs on all members of `dataset` which are in case of the parcel project filtered using the filter passed to the constructor.

run_config - For zone projects it is None. For parcel projects it has entries:

```python
{
    'exclude_outliers_from_initial_error': True,
    'outlier_is_less_than': 3,
    'outlier_is_greater_than': 7
}
```

25.4.5 Location Choice Model

A location choice model is a choice model where the set of alternatives is a set of locations.

The class `LocationChoiceModel` is a child of `ChoiceModel` described in Section 24.4.3. In addition, it allows sampling of alternatives and filtering alternatives according to a specified filter.

Initialization

The class is initialized by passing the following arguments:

location_set - A dataset of locations to be chosen from.

sampler - A fully qualified name of the module for sampling alternatives. Default value is “opus_core.samplers.weighted_sampler”. If this argument is set to None, no sampling is performed.

utilities - A fully qualified name of the module for computing utilities (see Section 24.5.1). Default value is “opus_core.linear_utilities”.

probabilities - A fully qualified name of the module for computing probabilities (see Section 24.5.2). Default value is “opus_core.mnl_probabilities”.

choices - A fully qualified name of the module for determining final choices (see Section 24.5.3). Default value is “opus_core.random_choices”.

interaction_pkg - This argument is only relevant if there is an explicit implementation of an interaction dataset that corresponds to interaction between agents and choices (such as those from Table 25.2). It then determines the package in which the module lives. Default value is “urbansim.datasets”.

filter - It is either a string specifying an attribute name of the filter for filtering out locations to be chosen from, or a 1D/2D array giving the filter directly, or a dictionary specifying filter for each submodel. If it is None (default), no filter is applied.

submodel_string - If model contains submodels, this character string specifies what agent’s attribute determines those submodels. If it is None (default), no division into submodels is applied.

location_id_string - A character string giving the fully qualified name of an agent attribute that specifies the location. It is only needed when the attribute is a variable (i.e. not a primary attribute). Default is None.
**run_config** - A collection of additional arguments that control a simulation run. It should be of class `Resources`. Default is None.

**estimate_config** - A collection of additional arguments that control an estimation run. It should be of class `Resources`. Default is None.

**dataset_pool** - A pool of datasets needed for computation of variables. Default is None.

The method calls the constructor of its parent class. Then, it creates a `Sampler` object from the argument `sampler` using `SamplerFactory`. It sets value of the argument `filter` as a class property `filter`.

### The Run Method

The **run()** method runs the simulation of location choice model on basis of a given specification and coefficients.

**Input:**
It takes the same arguments as the **run()** method of its parent (see Section 24.4.3 for more details):
*specification*, *coefficients*, *agent_set*, *agents_index*, *chunk_specification*, *data_objects*, and *run_config*.

**Algorithm:**
The model is processed in chunks. The **run_chunk()** method moves the agents out of their locations, i.e. the values of an `agent_set` attribute of the same name as the unique identifier of `location_set` is set to $-1$ for each agent of the currently processed chunk. If `agent_set` does not have this attribute, it is appended to it.

If an entry “compute_capacity_flag” is given in `run_config` and its value is True, an entry “capacity_string” is expected in `run_config` which gives the name of attribute/variable of `location_set` that determines capacity for each location. In such a case, after removing agents from their locations, the capacity is computed using method `determine_units_capacity()`. Note that by removing agents of only the current chunk from their locations, the capacity is influenced by only those agents. Each chunk then see the state of the world updated by all previously running chunks.

By default, the capacity values are used as weights of locations in the case of sampling. This can be changed by setting the entry “weights_for_simulation_string” of `run_config`, which should be a fully-qualified variable name of the location set that determines weights. The weights are multiplied by the filter given in the initialization. The model then invokes sampling of alternatives by calling the **run()** method of the sampler class, passing the possibly filtered weights. The parent class then takes care of creating the interaction set, for agents of the corresponding chunk and possibly sampled alternatives and of running the **upc_sequence**.

The location IDs that agents chose in the choice process are stored in the `agent_set` attribute specifying locations.

**Output:**
The method returns an array of size `agents_index`, representing the location IDs that agents (elements of `agent_set` determined by `agents_index`) made. Agents whose choice is less equal zero were not included in the choice process, for example because they do not belong to any submodels given in the specification.

### The Estimate Method

**Input:**
The **estimate()** method takes the same arguments as its parent class:
*specification*, *agent_set*, *agents_index*, *procedure*, *data_objects*, and *estimate_config*.

**Algorithm:**
As in the \texttt{run()} method, if “compute\_capacity\_flag” is given in \texttt{estimate\_config} and its value is True, an entry “capacity\_string” is expected in \texttt{estimate\_config} which gives the name of attribute/variable of \texttt{location\_set} that determines capacity for each location. In such a case, the capacity is computed using method \texttt{determine\_units\_capacity\_for\_estimation()}.

The weights for sampling alternatives are determined by an optional entry “weights\_for\_estimation\_string” in \texttt{estimate\_config} which should be an attribute/variable name of \texttt{location\_set}. They are multiplied by the given \texttt{filter} (if any) and as in the case of the \texttt{run()} method, sampling is performed using those weights. The parent class performs then the estimation.

\textit{Output:}

The method passes the return values of its parent method \texttt{estimate()}, i.e. a tuple of the created \texttt{Coefficient} object and a dictionary with entries for each submodel equals a dictionary returned by the \texttt{run()} method of \texttt{procedure} for that submodel.

Model Configuration

The arguments \texttt{run\_config} and \texttt{estimate\_config} are collections of parameters that control the \texttt{run()} and \texttt{estimate()} method, respectively. In addition to the mentioned entries “compute\_capacity\_flag”, “capacity\_string”, “weights\_for\_simulation\_string”, and “weights\_for\_estimation\_string”, they can contain entries “sample\_proportion\_locations” and “sample\_size\_locations”. Both entries control the size of sampled alternatives, the first one as a relative number, the latter one as an absolute number. The latter one has a priority over the first one.

\texttt{estimate\_config} can also contain an entry ‘chunk\_specification\_for\_estimation’, which should be an object of class \texttt{ChunkSpecification} (see Section 24.7.3). It controls the number of chunks in which the sampling is done for the estimation.

25.4.6 Development Project Location Choice Model

The Development project location choice model implemented in the \texttt{urbansim} package simulates a process where development projects of a specific type choose locations to be placed into (see Section 17.2.8 on page 103). The class \texttt{DevelopmentProjectLocationChoiceModel} is a child of \texttt{LocationChoiceModel} described in Section 25.4.5.

Initialization

The class is initialized by passing the following arguments:

\texttt{location\_set} - A dataset of locations to be chosen from.

\texttt{project\_type} - A character string determining the type of the projects for this model, such as ‘commercial’ or ‘residential’.

\texttt{units} - A character string giving the name of the attribute that determines sizes of projects.

\texttt{developable\_maximum\_unit\_variable\_full\_name} - A character string determining which Opus variable of the location set to use for computing the maximum developable units in each location.

\texttt{developable\_minimum\_unit\_variable\_full\_name} - A character string determining which Opus variable of the location set to use for computing the minimum developable units in each location. Default is None which means there is no restriction on the minimum number of units.
**model_name** - An optional argument giving the model name. Default is None.

... all remaining arguments from the constructor of `LocationChoiceModel`.

The arguments are set as class properties and the parent constructor is invoked.

### The Run Method

**Input:**

It takes the same arguments as the `run()` method of its parent: `specification`, `coefficients`, `agent_set`, `agents_index`, `chunk_specification`, `data_objects`, and `run_config`.

The `agent_set` is expected to be an instance of `DevelopmentProjectDataset`. It is assumed that initially all projects are unplaced.

**Algorithm:**

The model must be set to use capacity (via entries “compute_capacity_flag” and “capacity_string” in `run_config`). The capacity attribute is considered as a variable that is 1 for locations that are developable for this particular project type, otherwise 0. This means that no more than one project of the same type can occupy one location. The method `determine_units_capacity()` assures that locations taken in previous chunks are marked as taken.

The weight array for sampling is constructed in each chunk by combining information about project sizes and minimum and maximum developable units in each location. Thus, the weight array is a 2-d array which implies that different projects have different weights for the same locations, depending on their size. The dimension of the location axis of this array is for memory reasons reduced to only locations that are developable. No additional filtering is done (the model sets the class property `filter` to `None`).

The class overwrites its parent’s method `get_agents_order()` in a way that it sorts the agents according to their sizes in descending order. This means that larger projects enter the choice process first.

The `run()` method proceeds otherwise as defined in the parent class.

**Output:**

It returns results of its parent class.

### The Estimate Method

The `estimate()` method is identical to its parent method `estimate()`.

### Creators

The model can be created using a pre-defined creator in `urbansim`. The class `DevelopmentProjectLocationChoiceModelCreator` sets useful default values for arguments of the constructor. The model is by default initialized with:

- **sampler** - “opus_core.samplers.weighted_sampler”
- **utilities** - “opus_core.linear_utilities”
- **probabilities** - “opus_core.mnl_probabilities”
choices - “urbansim.first_agent_first_choices” (see Section 25.5.2)

submodel_string - “size_capacity”

filter - `'urbansim.gridcell.developable_%s' % units`

residential - False, project type is not of residential type

Entries of run_config are set as to:

```
{   "compute_capacity_flag": True,
    "sample_size_locations": 30,
    "capacity_string": "urbansim.gridcell.is_developable_for_%s" % units }
```

The variable units in the last line is taken from a model configuration that is passed to the creator in an entry “units” of the corresponding project type (see model configuration on page 253).

Entries of estimate_config are set as to:

```
{   "estimation": "opus_core.bhhh_mnl_estimation",
    "sample_size_locations": 30,
    "estimation_size_agents": 1.0 }
```

Model Configuration

This implementation is used only in the gridcell version of UrbanSim. The production run is configured with three development project types: “residential”, “commercial”, and “industrial”. Thus, there are three instances of the model, each initialized with the particular project type and a GridcellDataset as location_set. Each of the models runs only if the Development Project Transition Model returns some projects for the project type, i.e. the output value for this particular project type is not None. The run() method of each of the model is called by passing the following arguments:

**specification** - An EquationSpecification object created with data from table “%s_development_location_choice_model_specification”.

**coefficients** - A Coefficients object created with data from table “%s_development_location_choice_model_coefficients”.

**agent_set** - A DevelopmentProjectDataset returned by the Development Project Transition Model in its entry for this particular project type.

The “%s” in the above character strings are replaced by project type.

By default the model runs on submodels determined by the sizes of the projects. Specifically, the initialization method is called with the argument submodel_string=urbansim.development_project.size_category (see 25.4.14 for the definition of the categories). Note that n numbers in the categories list correspond to n + 1 submodels. For running the model on all projects without categorizing, set the sub_model_id field of the specification and coefficients table to −2.
25.4.7 Agent Location Choice Model

The class AgentLocationChoiceModel is a child of LocationChoiceModel described in Section 25.4.5. It can deal with exceeded capacity of locations.

Initialization

In addition to its parents’ arguments location_set, sampler, utilities, probabilities, choices, interaction_pkg, filter, submodel_string, location_id_string, run_config, estimate_config, and dataset_pool, it takes arguments:

- **model_name** - character string giving the name of the model.
- **short_name** - character string giving an abbreviation of the model name which is used only by the model logger.
- **variable_package** - Opus package used for computing variables/expressions given in run_config and estimate_config. Default is ‘urbansim’.

The constructor completes any unqualified variable names given in arguments, run_config and estimate_config to fully-qualified names, by using the dataset name of the given location_set and variable_package. Then the parent constructor is called.

The Run Method

*Input:*

In addition to its parents’ arguments specification, coefficients, agent_set, agents_index, chunk_specification, data_objects, and run_config, it accepts:

- **maximum_runs** - maximum number of iterations (see the algorithm below). Default is 10.

The method puts the parent method run() into a loop which is repeated whenever there is an overflow in the capacity after the last run. This behavior is activated only if the “compute_capacity_flag” entry of run_config is set to True. In such a case, run_config also should have entries “number_of_units_string” giving the variable name for computing number of available units in each location, and “number_of_agents_string” giving the variable name for computing the number of agents located in each location. Those variables are computed and the difference in their values determines the overfilled locations. If there are any such locations, an appropriate number of agents (from the set given by agents_index) are removed from their locations and the parent run() method is called again. The number of such iterations is given by the argument maximum_runs.

25.4.8 Household Location Choice Model

This model (class HouseholdLocationChoiceModel) is an Agent Location Choice Model that makes default settings of various arguments usable in the context of households choosing their locations. See also Section 17.2.6. The model is initiated with the arguments and their default values listed below. They are either directly passed to the constructor of the parent or packed into the run_config and/or estimate_config arguments under appropriate names.

- **location_set** - no default value
sampler - "opus_core.samplers.weighted_sampler"
utilities - "opus_core.linear_utilities"
choices - "opus_core.random_choices"
probabilities - "opus_core.mnl_probabilities"
estimation - "opus_core.bhhh_mnl_estimation"
capacity_string - "vacant_residential_units"
estimation_weight_string - "residential_units"
simulation_weight_string - None. If this argument is None, weights are proportional to the capacity.
number_of_agents_string - "number_of_households"
number_of_units_string - "residential_units"
sample_proportion_locations - None
sample_size_locations - 30
estimation_size_agents - 1.0
compute_capacity_flag - True
filter - None
submodel_string - None
location_id_string - None
demand_string - None. If this argument is not None, the aggregated demand for locations are stored in this attribute (see description of the run method in Section 24.4.3).

run_config - None.
estimate_config - None
dataset_pool - None
variable_package - "urbansim"

Model Configuration

Our gridcell version of UrbanSim configures the Household Location Choice Model with a GridcellDataset passed as argument location_set of the constructor. The zone version uses here a ZoneDataset object and the parcel version uses a BuildingDataset. Additionally, in all versions the default value of choices is overwritten by "urbansim.lottery_choices" (see Section 25.5.2).

The following arguments are passed to the run() method:
specification - An EquationSpecification object created with data from table “household_location_choice_model_specification”.
coefficients - A Coefficients object created with data from table “household_location_choice_model_coefficients”.
agent_set - A HouseholdDataset object used in Household Relocation Model.
agents_index - Results of Household Relocation Model (see page 261). Make sure that model is turned on when running Household Location Choice Model.
25.4.9 Agent Location Choice Model Member

This model is a child of AgentLocationChoiceModel (Section 25.4.7) that is used for running the parent class with different subsets of the agents.

Initialization

The class is initialized by passing the following arguments:

- **group_member** - An object of class ModelGroupMember (see Section 24.7.2) determining for which member is this model initialized.
- **location_set** - A dataset of locations.
- **agents_grouping_attribute** - An attribute of the agent set that is used for classifying agents into the member groups.

Other arguments of the parent class can be passed.

The Run and Estimate Methods

For both methods, the agents that correspond to the given member are selected and the model is run/estimated only on those agents. The return array of the run() method contains -1 for agents that do not belong to the group.

The prepare_for_estimate method appends the member name as prefix in front of the specification and coefficients tables.

25.4.10 Employment Location Choice Model

The class EmploymentLocationChoiceModel is a child of AgentLocationChoiceModelMember. Its default settings of various arguments makes the model usable in the context of jobs choosing their locations. See also Section 17.2.5. It is initialized by the following arguments and default values:

- **group_member** - no default value
- **location_set** - no default value
- **agents_grouping_attribute** - 'job.building_type'
- **sampler** - "opus_core.samplers.weighted_sampler"
- **utilities** - "opus_core.linear_utilities"
- **choices** - "opus_core.random_choices"
- **probabilities** - "opus_core.mnl_probabilities"
- **estimation** - "opus_core.bhhh_mnl_estimation"
- **capacity_string** - "vacant_SSS_job_space"
- **estimation_weight_string** - "total_number_of_possible_SSS_jobs"
simulation_weight_string: None. If this argument is None, weights are proportional to the capacity.

number_of_agents_string: "number_of_SSS_jobs"

number_of_units_string: "total_number_of_possible_SSS_jobs"

sample_proportion_locations: None

sample_size_locations: 30

estimation_size_agents: 1.0

compute_capacity_flag: True

filter: None

submodel_string: "sector_id"

location_id_string: None

demand_string: None. If this argument is not None, the aggregated demand for locations are stored in this attribute (see description of the run method in Section 24.4.3).

run_config: None.

estimate_config: None

dataset_pool: None

variable_package: "urbansim"

The initialization procedure replaces any occurrence of the string 'SSS' in the various arguments by the name of the group member. For example, if the group member corresponds to the 'commercial' building type of jobs, the value of argument capacity_string would be converted to "vacant_commercial_job_space". Arguments are then either directly passed to the constructor of the parent or packed into the run_config and/or estimate_config arguments under appropriate names.

Model Configuration

In the gridcell and zone version of UrbanSim, this model is called by default for members 'commercial', 'industrial', and 'home_based'. In the parcel version, we use only the group member 'non_home_based'. Note that these names must be contained in the table 'job_building_types'.

The following arguments are passed to the constructors of models for all members:

location_set: A GridcellDataset object in the gridcell project, ZoneDataset in the zone project, and BuildingDataset in the parcel project.

choices: 'urbansim.lottery_choices'

The default submodel_string is 'sector_id', which associates submodels with the employment sectors defined in the table 'employment_sectors'. Additionally, the member 'home_based' overwrites the arguments estimation_weight_string by the value 'residential_units' and number_of_units_string by the value 'residential_units'.

In all versions, the run() method is called with the following arguments:
specification - Uses data from “%s_employment_location_choice_model_specification” where %s is replaced by the member name.

coefficients - Uses data from “%s_employment_location_choice_model_coefficients” where %s is replaced by the member name.

agent_set - A JobDataset object used in Employment Relocation Model.

agents_index - Results of Employment Relocation Model (see page 261). Make sure that model is turned on when running Employment Location Choice Model.

25.4.11 Building Location Choice Model

The class BuildingLocationChoiceModel is a child of AgentLocationChoiceModelMember. Its default settings of various arguments makes the model usable in the context of buildings choosing their locations. It is an alternative model to the Development Project Location Choice Model (see Section 25.4.6) and it is tailored for the case when locations are single parcels. It should be run in combination with the Building Transition Model (see Section 25.4.17). It is initialized by the following arguments and default values:

group_member - no default value

location_set - no default value. An object of class ParcelDataset is recommended to be passed here.

agents_grouping_attribute - "building.building_type_id"

sampler - "opus_core.samplers.weighted_sampler"

utilities - "opus_core.linear_utilities"

choices - "urbansim.first_agent_first_choices"

probabilities - "opus_core.mnl_probabilities"

estimation - "opus_core.bhhh_mnl_estimation"

capacity_string - "is_developable_for_buildings_UNITS"

estimation_weight_string - None

developable_maximum_unit_variable - "developable_maximum_buildings_UNITS",

developable_minimum_unit_variable - "developable_minimum_UNITS". None means that there are no minimum restrictions.

number_of_agents_string - "buildings_SSS_space"

number_of_units_string - "total_maximum_development_SSS"

sample_proportion_locations - None

sample_size_locations - 30

estimation_size_agents - 1.0

compute_capacity_flag - True

filter - "developable_maximum_buildings_UNITS"

submodel_string - "size_category_SSS"
location_id_string - None
nrecords_per_chunk_for_estimation_sampling = 1000
run_config - None.
estimate_config - None
dataset_pool - None
variable_package - "urbansim"

The initialization procedure replaces any occurrence of the string 'SSS' in the various arguments by the name of the group member. For example, if the group member corresponds to the 'commercial' type of buildings, the value of argument submodel_string would be converted to "size_category_commercial". The group member object must have an attribute 'units'. Its value is used for replacing all occurrences of 'UNITS' in the arguments. For example, if the member units value is 'commercial_sqft', the value of argument capacity_string would be replaced by 'is_developable_for_buildings_commercial_sqft'. The various attribute names are also transformed to fully-qualified names (using the name of the location dataset and variable_package). Arguments are then either directly passed to the constructor of the parent or packed into the run_config and/or estimate_config arguments under appropriate names.

The Run Method

The run() method only differs from its parent in a way of constructing the array of weights for sampling locations. It is a 2-d array of ones and zeros, where one corresponds to locations where the particular building would fit into.

25.4.12 Household Transition Model

The class HouseholdTransitionModel creates and removes households to and from given household set according to the joint distribution of their characteristics. The total number of households is controlled by given annual control totals. See also Section 17.2.2.

Initialization

The class is initialized by passing the following arguments:

location_id_name - name of a household attribute that determines locations of households. Default value is 'grid_id'.

The argument is set as a class property.

The Run Method

Input:

tyyear - an integer indicating the simulated year.

household_set - an instance of Dataset that contains some specific attributes (see description of the algorithm bellow).
control_totals - an instance of HouseholdControlTotalDataset. It must have at least attributes “total_number_of_households” and “year” which give the control totals for specific years. Optionally, it can contain other attributes whose combinations determine groups for the control totals. These are e.g. “age_of_head”, “income”, “persons”, “cars”, “children”, “race_id”, “workers”. Each value of such attribute is the corresponding group index (starting from zero) for this characteristics from the argument characteristics (see below). The unique identifier of this dataset should consist of all attributes but the “total_number_of_households” (for details see Section “annual_household_control_totals” in 18.8.1).

characteristics - an instance of class HouseholdCharacteristicDataset that specifies grouping of characteristics. It must have three attributes: “characteristic”, “min” and “max”. Values of the “characteristic” attribute should match the names of the existing optional attributes of control_totals. “min” and “max” determine group boundaries for each characteristics (for details see Section “household_characteristics_for_ht” in 18.8.4). If there is a characteristics missing that is contained in control_totals, the grouping is assumed to be [0, 1), [1, 2), [2, 3), etc.

resources - additional Resources for controlling the simulation run. The argument is not used in this version of the model.

Algorithm:
The optional attributes in control_totals are called ‘marginal’ characteristics. The given household_set must contain the marginal characteristics as its primary attributes.

The combination of all marginal characteristics and the grouping within each of them determines distinct marginal bins. The method iterates over those bins. In each iteration the number of households is determined whose properties match the characteristics of the bin. This number is compared to the control total for this bin. If the difference d is positive, new households are created, if it is negative, households are removed, if it is zero, nothing is done.

When creating households, the method samples d existing households from all households that belong to that marginal bin. For each sampled household, values of all attributes (except of the id attribute and the location attribute) are duplicated and saved as a new household. New ids are assigned to all new households and their location is set to ‘unplaced’. Households are considered as unplaced if their attribute given by the class property location_id_name is smaller equal zero. If there are no existing households a the marginal bin, no households for that bin are created. Note that this can lead to a mismatch between the control totals and the number of households.

To remove households, first unplaced households from the bin are removed. If the number n_u of those unplaced households is larger than d, only d households are randomly sampled for deletion. If n_u is smaller than d, then d – n_u households from the set of placed households of that bin are randomly sampled and deleted.

Output:
The method returns the total difference between the sizes of the household dataset after and before the model run. Thus, a positive value means that in total, there were more households added than removed, a negative value means the opposite.

Example

Consider the following example. We have two marginal characteristics defined in the control totals - persons and income. The dataset characteristics defines two categories - persons (3 groups) and income (2 groups). A combination of those groups divides the space of households characteristics into 6 groups in total.

The table for the control totals dataset can be defined as follows:
The characteristics table defines the groups of each characteristic:

<table>
<thead>
<tr>
<th>characteristic</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>persons</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>persons</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>persons</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>income</td>
<td>0</td>
<td>49999</td>
</tr>
<tr>
<td>income</td>
<td>50000</td>
<td>-1</td>
</tr>
</tbody>
</table>

Note that \(-1\) stands for \(\infty\). The values in the control totals table denotes the index (starting from 0) of the particular group within the characteristics table. E.g. the first line in the control totals table refers to the persons group \([0, 2]\) and income group \([0, 49999]\).

The model iterates over the 6 bins defined by the marginal characteristics. For each of them, it determines the number of households that belong to that group in terms of their characteristics and compares it with the control total for that group. If for example in one of the bins there are 10 households to be created, the model would randomly sample (with replacement) 10 existing households from that bin and duplicate them. If the difference between control total and the number of households in a bin would call for removing households, the model would randomly sample households belonging to that bin and delete them.

**Model Configuration**

In a production run of UrbanSim, the model runs with the following arguments:

- **year** - The current year of the simulation.
- **household_set** - A `HouseholdDataset` object created with data from table “households” (see Section 18.8 for the table structure).
- **control_totals** - A `HouseholdControlTotalDataset` object that reads data from table “annual_household_control_totals” (see Section 18.8 for the table structure).
- **characteristics** - A `HouseholdCharacteristicDataset` object created with data from table “household_characteristics_for_ht” (see Section 18.8 for the table structure).

**Troubleshooting**

Problem: The resulting number of households after running the model is different than the control totals.
One possible cause: There is a mismatch between data in the control totals table and the characteristics table. For example, the indexing of groups in the control totals table starts with 1 instead of 0. Check carefully these two tables.

Other possible cause: There are no existing households for some of the combinations of characteristics.

Problem: In the first simulation year the model deletes and creates a large amount of households.

Cause: The distribution of the actual households characteristics in the base year is very different from the control totals. Consider the example above and suppose that the actual number of households in the base year in the six marginal categories is 200000, 50000, 20000, 360000, 10000, 100000. The control totals for year 2006 are 100100, 230000, 10000, 150000, 250000, 5000 (as given in the table above). The model changes the household set according to differences in these categories, which are -99900, 180000, -10000, -210000, 240000, -95000. As a result, even if the total number of households changed only by 5100 households (from 740000 in the base year to 745100 in year 2006), there were 414900 households deleted and 420000 new households created, which is far more than the half of all households.

Problem: The model fails with a KeyError.

Possible cause: Some households contain values that do not fit in any of the groups given by the marginal characteristics. Check especially categories where you have non-zero minima or finite maxima. For example, one has the number 18 as minimum for “age_of_head” in the characteristics table, but the households table contains records with smaller values than 18. The model must be able to put each household into one of the defined groups.

Problem: The model fails with an error about a missing attribute.

Cause: There are marginal characteristics that are not contained in the households table. Compare the control totals table to the attributes of households.

25.4.13 Employment Transition Model

The class EmploymentTransitionModel creates and removes jobs to and from given job set according to a job distribution over employment sectors. The total number of jobs is controlled by given annual control totals. It distinguishes between home based and non-home based jobs. See also Section 17.2.1.

The algorithm is similar to the Household Transition Model. Here, the only marginal characteristics are sector identifiers.

Initialization

The class is initialized by passing the following arguments:

- **location_id_name** - name of a household attribute that determines locations of households. Default value is 'grid_id'.
- **variable_package** - Opus package where various sector variables are located (see below). Default is 'urbansim'.
- **dataset_pool** - dataset pool passed to variable computation. Default is None.

The arguments are set as class properties.
The Run Method

Input:

**year** - an integer indicating the simulated year.

**job_set** - an instance of Dataset that contains some specific attributes (see description of the algorithm below).

**control_totals** - an instance of EmploymentControlTotalDataset. It must have at least attributes “sector_id”, “total_home_based_employment”, “total_non_home_based_employment” and “year” which specify the annual control totals per sector and employment type. The unique identifier of this dataset should be “year” and “sector_id” (for details see Section “annual_employment_control_totals” in 18.7).

**job_building_types** - an instance of class Dataset that contains unique building types of jobs. It must contain an attribute ‘home_based’ determining which of the building types are home based (see “job_building_types” in 18.7).

**data_objects** - a dictionary containing other datasets and arguments needed for computing variables. Default is None.

**resources** - additional Resources for controlling the simulation run. Default is None. The argument is not used in this version of the model.

Algorithm:
The **job_set** is required to have a primary attribute “building_type” with values that correspond to values of the unique identifier of **job_building_types**. The algorithm invokes a computation of variables “is_in_employment_sector_n_home_based” and “is_in_employment_sector_n_non_home_based” implemented in the package given by the class property variable_package. $n$ in the variable names is the sector id. If you use the default urbansim implementation of those variables, they require a primary attributes “home_based” of the dataset **job_building_types** and a primary attribute “sector_id” of **job_set**. Each job is set to be home based if its building type is home based, otherwise the job is non-home based.

The method iterates over sectors given in **control_totals**. For each sector it determines the number of jobs of that sector that are home based (non-home based) using the above variables. It then compares those numbers to the control totals. If the difference $d_h$ ($d_{nh}$) is positive, new home based (non-home based) jobs are created, if it is negative, home based (non-home based) jobs are removed, if it is zero, nothing is done.

Jobs that are created get the corresponding values for the attribute “sector_id”. For each combination of (sector id, home based) and (sector id, non-home based) it is determined, if there are any jobs of this group previously available. In such a case, the distribution of “building_type” among those jobs is determined and the values for “building_type” of the new jobs are sampled from this distribution. If there are no existing jobs in this group, it is sampled from the “building_type” distribution obtained over all home based (non-home based) jobs, regardless of the sector id.

To remove home based (non-home based) jobs from a sector, first unplaced home based (non-home based) jobs of this sector are removed. If the number $n_{h_u}$ ($n_{nh_u}$) of those unplaced jobs is larger than $d_h$ ($d_{nh}$), only $d_h$ ($d_{nh}$) jobs are randomly sampled for deletion. If $n_{h_u}$ ($n_{nh_u}$) is smaller than $d_h$ ($d_{nh}$), then $d_h - n_{h_u}$ ($d_{nh} - n_{nh_u}$) jobs from the set of placed jobs of that sector are randomly sampled and deleted. Jobs are considered as unplaced if their attribute given by the class property location_id_name is smaller equal zero.

Output:
The method returns the total difference between the sizes of the job dataset after and before the model run. Thus, a positive value means that in total there were more jobs added than removed, a negative value means the opposite.
Model Configuration

In a production run of UrbanSim, the model runs with the following arguments:

\textbf{year} - The current year of the simulation.

\textbf{job\_set} - A \texttt{JobDataset} object created with data from table “jobs” (see Section 18.7 for the table structure).

\textbf{control\_totals} - A \texttt{EmploymentControlTotalDataset} object that reads data from table “annual_employment_control_totals” (see Section 18.7 for the table structure).

\textbf{job\_building\_types} - A \texttt{JobBuildingTypeDataset} created with data from table “job\_building\_types” (see Section 18.7 for the table structure).

Troubleshooting

Problem: In the first simulation year the model deletes and creates a large amount of jobs.

Cause: The distribution of the actual job sectors in the base year is very different from the control totals. See an analogous example in the Troubleshooting section of the Household Transition Model (25.4.12).

25.4.14 Development Project Transition Model

The model (implemented in the class \texttt{DevelopmentProjectTransitionModel}) creates development projects in order to match the desired vacancy rates. Each development project is for a single type of development, e.g. ‘industrial’ or ‘commercial’. The distribution of project sizes (amount of space, value of space) is determined by sampling from the projects obtained from the history.

The Run Method

\textit{Input:}

\textbf{model\_configuration} - a \texttt{Configuration} object that determines project types and settings for each type (see Section Model Configuration below).

\textbf{vacancy\_table} - a \texttt{Dataset} object that must have attribute “year” as the unique identifier and should contain attributes “target\_total\_residential\_vacancy” and “target\_total\_non\_residential\_vacancy”. Those attributes give vacancy rates for specific years.

\textbf{history\_table} - a \texttt{Dataset} object giving historical data of development events.

\textbf{year} - simulated year.

\textbf{location\_set} - a \texttt{Dataset} object on which vacancies will be determined.

\textbf{resources} - a \texttt{Resources} object for additional arguments. It can contain names of variables that compute the current vacancies per project type. Those entry names are composed from the project type and a character string ‘vacant\_variable’, e.g. for commercial project type the key for this argument is “commercial\_vacant\_variable”. Default values for those entries are “urbansim.gridcell.vacant\_units” where \textit{units} is given in the entry “units” of \texttt{model\_configuration} (see below for details), e.g. “urbansim.gridcell.vacant\_commercial\_sqft”.
**Algorithm:**

The method first determines the target vacancy rates. It distinguishes between residential and non-residential vacancy rate. For each project type given in the `model_configuration`, the method obtains current vacancy in each location (from `location_set` and the overall vacancy rate). By comparing the rate with the target vacancy rate, it determines if there is a need for development and how many units should be developed. If there are units to be developed, from the historical data it samples sizes of projects so that their sum gives approximately the desired number of units to be developed. It then creates an instance of class `DevelopmentProjectDataset` for this particular project type that has unique identifier called “project_id”. Each project corresponds to one of the sampled amount of units. This amount is stored in the new dataset as an attribute given in the entry “units” of `model_configuration` (see below for details). If an attribute called project type + “_improvement_value” is contained in the `location_set`, an average improvement value is computed over all locations and this project type. This value is then added to each project in the created `DevelopmentProjectDataset`. Additionally, an attribute of the same name as the unique identifier of `location_set` is added to the `DevelopmentProjectDataset` and set to 0’s (which means that the projects are unplaced).

**Output:**

The method returns a dictionary with an entry for each project type containing the corresponding `DevelopmentProjectDataset` object. If no projects were created for a project type, the corresponding dictionary value is `None`.

**Model Configuration**

The `run()` method of this class expects a dictionary `model_configuration` as an argument. This object must contain an entry “development_project_types” which is also a dictionary. The name of each element is a project type and value is again a dictionary. Entries on this level are “units”, “categories”, and “residential”. Entry “units” gives the name of the attribute that determines sizes of projects. An attribute of the same name must be contained in the `location_set` and in the dataset with the historical data. Entry “categories” determines categories of a project set. It can be an empty array if categories are not used. Entry “residential” is a boolean value determining if vacancy of this project type should be compared to the residential vacancy rate or not. In the latter case, it is compared to the non-residential vacancy rate.

The model is used in the gridcell version of UrbanSim only. Here, the method `run()` is called using the following arguments:

- **`model_configuration`** - The default configuration for project types in ‘configs/general_configuration.py’ is
vacancy_table - An object of class TargetVacancyDataset created with data from table “target_vacancies” (see Section 19.5.1 for the table structure).

history_table - An object of class DevelopmentEventDataset created with data from table “development_event_history” (see Section 19.3.2 for the table structure).

location_set - An instance of class GridcellDataset created with data from table “gridcells” (see Section 19.1 for the table structure).

25.4.15 Development Event Transition Model

The model DevelopmentEventTransitionModel creates for each location a development event, which is a collection of development projects of different type (e.g. commercial, residential, industrial) for this location. This model is not estimated and contains only a method run().

The Run Method

Input:

projects - a dictionary where key is the type of development project and value is an instance of DevelopmentProjectDataset for that particular project type.

types - a list of project types which the resulting event should consist of.

units - a list of attribute names (defined as dataset qualified names) that determine the units for each project type (e.g. “commercial_sqft” for commercial type or “residential_units” for residential type). The ordering in the list must correspond to the ordering in types.

year - an integer indicating for which year this events are created. Default value is 0.

location_id_name - name of the project attribute that determines locations. Default value is “grid_id”.
Algorithm:
It generates a distinct set of all locations in which at least one project is located. It creates a DevelopmentEventDataset of size of number of those locations. It has an attribute given in location_id_name with id values of the locations. Furthermore, it has all attributes given in the argument units. Values for each of these attributes are the sums of values of the attribute of the same name in the corresponding project dataset over the corresponding locations. It also has an attribute “scheduled_year” with the same value for each location, namely the one given in the argument year. Additional attributes type + ”_improvement_value” contain the sum of the project set attribute “improvement_value” over the corresponding locations.

Output:
The model returns an instance of DevelopmentEventDataset.

Model Configuration

The model is only used in the gridcell version of UrbanSim. The run() method is called by passing the following arguments:

projects - A dictionary that results from a run of the Development Project Transition Model, and is (possibly) modified by runs of all three Development Project Location Choice Models.

types - A list of project types for which the Development Project Location Choice Model has run, i.e. for which there are projects to be developed. Typically, ['residential', 'commercial', 'industrial'].

units - A list of “units” entries from model configuration in ‘general_configuration.py’ (see page 253) for which the Development Project Location Choice Model has run, i.e. that correspond to types. Typically, ['residential_units', 'commercial_sqft', 'industrial_sqft'].

year - The simulated year.

25.4.16 Process Prescheduled Development Events

This model (implemented in class ProcessPrescheduledDevelopmentEvents) is intended to be used in combination with the Events Coordinator (see Section 25.4.18). It reads exogeneous events from the storage (e.g. from database or cache) and returns them as an Opus object for further processing (e.g. by the Events Coordinator).

The Run Method

Input:

storage - An object of class Storage that holds the exogeneous events.

in_table - Table name of the events. Default is 'development_events'. (see Section 19.3.1 for the table structure).

out_table - Table name of writing the events. The argument is not used in the model.

Algorithm:
The method creates an object of class DevelopmentEventDataset, using the input arguments and returns it.
Model Configuration

The model is only used in the gridcell version of UrbanSim. It is run as the first model of each year. The returned object is passed as an input argument of the `run()` method of the Events Coordinator. The `storage` argument is configured with the base year cache storage.

25.4.17 Building Transition Model

The model (implemented in the class `BuildingTransitionModel`) creates buildings in order to match the desired vacancy rates. It is an alternative model to the Development Project Transition Model described in Section 25.4.14, tailored to the case when the geography unit is a parcel. The process of creating buildings is done per building type, e.g. ‘industrial’ or ’commercial’. The distribution of building sizes is determined by sampling from existing buildings.

The Run Method

**Input:**

- `building_set` - An object of class `Dataset` holding data about existing buildings.
- `building_types_table` - An object of class `Dataset` holding data about building types. It must have attributes “name”, “is_residential” and “units”.
- `vacancy_table` - A `Dataset` object that must have attribute “year” as the unique identifier. Furthermore, it should contain attributes “target_total_%s_vacancy” where %s stands for a building type contained in the column “name” of the `building_types_table`. Those attributes give vacancy rates for specific building types and years.
- `year` - Simulated year.
- `location_set` - A `Dataset` object on which vacancies will be determined. We recommend to use a parcel dataset.
- `building_categories` - A dictionary where keys are building types and values are arrays of categories of building sizes. Default is None.
- `dataset_pool` - A pool of dataset used for computation of variables. Default is None.
- `resources` - A `Resources` object for additional arguments. It can contain names of variables that compute the current vacancies per building type. Those entry names are composed from the building type and a character string ‘_vacant_variable’, e.g. for commercial building type the key for this argument is “commercial_vacant_variable”. Default values for those entries are “urbansim.location_set_name.vacant_building_type_units_from_buildings” for residential buildings, or “urbansim.location_set_name.vacant_building_type_sqft_from_buildings” for non-residential buildings, e.g. “urbansim.parcel.vacant_commercial_sqft_from_buildings”.

**Algorithm:**

For each building type given in the `building_types_table` (in column “name”), the method obtains current vacancy in each location (from a variable of `location_set` called “urbansim.location_set_name.building_type_space” and the overall vacancy computed using variable given in the `resources` entry “building_type_vacant_variable”). By comparing the rate with the target vacancy rate, it determines if there is a need for development and how many units should be developed. If there are units to be developed, a sample from the existing buildings of that building type is taken, so that their sum of units gives approximately the desired number of units to be developed. The name of the building set attribute that specifies units is given in the `building_types_table` (in
column “units”). Copies of these sampled buildings (with modified attribute giving their location and modified unique identifier) are added to the existing set of buildings.

Output:
The method returns the difference between the number of buildings before and after running this model.

25.4.18 Events Coordinator

The class `EventsCoordinator` updates a location set to reflect changes that are contained in the given event set. The event set can be an output of the Development Event Transition Model (see Section 25.4.15) or of the Process Prescheduled Development Events (see Section 25.4.16). There is no `estimate()` method for this model.

The Run Method

Input:

`model_configuration` - a `Configuration` object that determines project types and settings for each type (see Model Configuration of Section 25.4.14).

`location_set` - a `Dataset` object to be updated.

`development_event_set` - a `DevelopmentEventDataset` with development events to be used to update the locations.

`development_type_set` - an object of class `DevelopmentTypeDataset`.

`current_year` - an integer value determining the year of events by which the locations are updated.

`models_configuration` - a `Configuration` object that is used with `development_models` to determines project types and settings for each type. This argument is alternative to `model_configuration`.

`development_models` - a list of development project location choice model names defined in models configuration

Algorithm:
The method adds/replaces/removes development contained in `development_event_set` to/by/from `location_set`. It updates the location attributes given by 'units' of `model_configuration`. The type of change can be given in event set attribute 'type_of_change' as 'A' for add, 'R' for replace and 'D' for delete. The default type of change is 'A'. The method also determines a new development type for each location depending on the existing development and the definition of development types given by the `development_type_set` (see Section 19.2).

Output:
Returns a tuple of indices of locations that were modified, and indices of development events that were processed.

Model Configuration

The model is only used in the gridcell version of UrbanSim. It is run twice per year: First, it processes the output of the Process Prescheduled Development Events model, and second time it processes the output of the Development Event Transition Model. The method `run()` is called using the following arguments:
**model_configuration** - The model configuration for project types from ‘general_configuration.py’ as shown on page 253.

**location_set** - An instance of class `GridcellDataset` created with data from table “gridcells” (see Section 19.1 for the table structure) and (possibly) modified by previous models.

**development_event_set** - Result of a Development Event Transition Model run or a Process Prescheduled Development Events run.

**development_type_set** - An object of class `DevelopmentTypeDataset` created with data from table “development_types” and “development_type_group_definitions” (see Section 19.2 for the tables structure).

**current_year** - The current year of the simulation.

**Events Coordinator And Storing**

A child class of `EventsCoordinator`, called `EventsCoordinatorAndStoring`, runs the parent class and stores the development event set into the simulation cache. This is especially useful for tracking down annual development.

### 25.4.19 Scaling Jobs Model

The class `ScalingJobsModel` locates jobs to locations according to the distribution of job sectors in locations.

**Initialization**

The model is initialized by passing the following arguments:

- **group_member** - An object of class `ModelGroupMember` (see 24.7.2). Default is None.
- **agents_grouping_attribute** - The agents grouping attribute. Default is ‘job.building_type’.
- **filter** - A fully-qualified variable name for filtering out a subset of locations to place jobs in. Default is None.
- **dataset_pool** - A pool of datasets used for computing variables. Default is None.

All arguments are set as class properties.

**The Run Method**

**Input:**

- **location_set** - an instance of `Dataset` containing the set of all locations for locating jobs.
- **agent_set** - an instance of `Dataset` containing the set of jobs. It must have an attribute “sector_id” and attribute of the same name as the unique identifier of the `location_set`.
- **agents_index** - indices of members of the `agent_set` for which the model runs. If it is None (default), the whole `agent_set` is considered.
**data_objects** - a dictionary containing other datasets and arguments needed for computing variables. Default is None.

**resources** - additional Resources for controlling the simulation run. The argument is not used in this version of the model. Default is None.

**Algorithm:**
The `location_set` is expected to have the variable “number_of_jobs_of_sector_n” implemented in the package given by the class property `variable_package` (default is “urbansim”), \( n \) is an integer giving sector IDs.

The method filters out jobs that are supposed to be located using `agents_index` and the ’membership’ of jobs given by ’group_member’. If ’group_member’ is None the group membership is ignored. If the class property `filter` is given, it uses that variable for selecting a subset of locations that will be further considered. Only those locations are taken where the value of the filter attribute is larger than 0. The method then determines distinct sectors of the considered jobs and their count in each of these sectors. It iterates over those sectors. In each iteration it randomly samples (with replacement) the desired count of locations weighted by the number of existing jobs of that sector in each location. If there are no existing jobs in that sector, it samples with equal weights for all considered locations. It then modifies the location attribute of the job set by storing the sampled location IDs.

**Output:**
The method returns an array of the new locations, i.e. of size `agents_index`. For entries in `agents_index` that do not belong to this group member, the values are -1.

**Model Configuration**
In the gridcell and zone version of UrbanSim, the model is run for group member ’governmental’. Note that this name must be included in the table ’job_building_types’. We don’t call this model directly in the parcel version, but use its child class `DistributeUnplacedJobsModel` only (see Section 25.4.20).

The `run()` method is called with the following arguments:

- `location_set` - A `GridcellDataset` object in the gridcell project; a `ZoneDataset` in the zone project.
- `agent_set` - A `JobDataset` object used in Employment Relocation Model.
- `agents_index` - Results of Employment Relocation Model (see page 261). Make sure that model is turned on when running Scaling Jobs Model.

**25.4.20 Distribute Unplaced Jobs Model**

This model is a child of the `ScalingJobsModel`. It determines all unplaced jobs, i.e. jobs with location identifier smaller equal 0. Then it calls the parent model on those jobs, setting ’group_member’ to None. The `run()` method accepts an additional argument `agents_filter` which is a fully-qualified variable name of the agent set. It allows for additional filtering of jobs to be processed.

**Model Configuration**

The model is run for jobs that were not placed by any of the Employment Location Choice Models nor by the Scaling Jobs Model. In all versions of UrbanSim, we pass an object of class `JobDataset` as `agent_set` argument to the `run()` method. In the gridcell and zone version, respectively, the `location_set` argument is set to an object of `GridcellDataset` and `ZoneDataset`, respectively.
In the parcel project, the **location_set** argument is set to an object of `BuildingDataset`. Furthermore, the argument **agents_filter** is set to 'urbansim.job.is_in_employment_sector_group_scalable_sectors’ and the constructor argument **filter** is set to 'urbansim_parcel.building.is_governmental’. In this case, the model is used for placing governmental jobs into governmental buildings.

The argument **agents_index** is set to None for all projects.

### 25.4.21 Agent Relocation Model

The class `AgentRelocationModel` determines which members of the given set of agents should be relocated. This is done on basis of given probabilities. Additionally, the way how to choose the agents can be given by passing a `Choices` object. See also Sections 17.2.3 and 17.2.4.

#### Initialization

The class is initialized by passing the following arguments:

- **probabilities** - A fully qualified name of the module that implements computing relocation probabilities. The module should be a child of the `opus_core` class `Probabilities` (see Section 24.5.2). Default is None.

- **choices** - A fully qualified name of the module that implements choosing agents for relocation according to given probabilities. The module should be a child of the `opus_core` class `Choices`. Default value is “opus_core.random_choices” (see Section 24.5.3).

- **location_id_name** - Name of the attribute that specifies locations. It must be a primary attribute of the agent set. Default value is “grid_id”.

- **model_name** - Name of the model. Default value is “Agent Relocation Model”.

If the **probabilities** argument is given, the initialization method creates a class attribute **upc_sequence**, using the passed arguments **probabilities** and **choices**. It is an object of class **upc_sequence** where the utilities component is set to None (see Section 24.5.4). All arguments are set as class properties.

#### The Run Method

**Input:**

- **agent_set** - A `Dataset` object containing agents to be relocated.

- **resources** - An instance of class `Resources` which is a collection of additional arguments and objects to be passed to the `upc_sequence` run.

**Algorithm:**

If the class attribute **upc_sequence** exists, the method appends **agent_set** into **resources** and invokes the `run()` method of **upc_sequence**, passing **resources** as argument. The **upc_sequence** runs the probabilities component and the choices component. It is expected to return an array of zeros for agents not to be relocated, and ones for agent to be relocated. The distinct values of indices of the “to be relocated” agents joined with indices of all unplaced agents is the resulting array of agents to be relocated. As unplaced agents are considered agents that have value smaller equal 0 for the attribute given in **location_id_name** passed to the constructor.

---

260 Chapter 25. The `urbansim Opus Package`
If there is no `upc_sequence`, the resulting array corresponds to indices of unplaced agents only.

**Output:**
The method returns the resulting array of indices of all agents to be relocated.

**Creators**

There are two pre-defined creators in urbansim. Their method `get_model()` returns an instance of `AgentRelocationModel` with the following default settings:

- **HouseholdRelocationModelCreator** - sets the argument `probabilities` to “urban-sim.household_relocation_probabilities” (see Section 25.5.1) and `model_name` to “Household Relocation Model”.

- **EmploymentRelocationModelCreator** - sets the argument `probabilities` to “urban-sim.employment_relocation_probabilities” (see Section 25.5.1) and `model_name` to “Employment Relocation Model”.

**Model Configuration**

- To run a **Household Relocation Model** in a production run, the `HouseholdRelocationModelCreator` is used to create an instance of the `AgentRelocationModel`. Additionally, it creates an instance of `RateDataset` initialized with the argument `what=“households”` which reads data from table “annual_relocation_rates_for_households” (see Section 18.8 for the table structure). This object is put into `resources` which is passed to the `run()` method of the model. As `agent_set` the production code passes a `HouseholdDataset` object that was (possibly) used and modified by the Household Transition Model. It precedes the Household Location Choice Model.

- To run a **Employment Relocation Model**, the `EmploymentRelocationModelCreator` is used to create an instance of the `AgentRelocationModel`. Additionally, it creates an instance of `RateDataset` initialized with the argument `what=“jobs”` which reads data from table “annual_relocation_rates_for_jobs” (see Section 18.7 for the table structure). This object is put into `resources` which is passed to the `run()` method of the model. As `agent_set` the production code passes a `JobDataset` object that was (possibly) used and modified by the Employment Transition Model. It precedes the Employment Location Choice Model.

### 25.5 Model Components

#### 25.5.1 Probabilities Classes

This section describes `Probabilities` classes (see Section 24.5.2) that are implemented in urbansim.

- `household_relocation_probabilities`

This class is implemented to be used with a rate set (instance of `RateDataset` initialized for households) structured according to the description for “annual_relocation_rates_for_households” in Section 18.8. The `run()` method iterates over households and using the rate set, for each household it determines from the values of required primary attributes “age_of_head” and “income” the probability of relocation. It returns a 1D array of those probabilities. The
household set and rate set, respectively, are expected to be contained in the argument \texttt{resources} of the \texttt{run()} method under the keys ‘household’ and ‘rate’, respectively.

\texttt{employment\_relocation\_probabilities}

This class is implemented to be used with a rate set (instance of \texttt{RateDataset} initialized for jobs) structured according to the description for “annual relocation rates for jobs” in Section 18.7. The \texttt{run()} method iterates over jobs and using the rate set, for each job it determines from the values of required primary attribute “sector_id” the probability of relocation. It returns a 1D array of those probabilities. The job set and rate set, respectively, are expected to be contained in the argument \texttt{resources} of the \texttt{run()} method under the keys ‘job’ and ‘rate’, respectively.

25.5.2 Choices Classes

This section describes \texttt{Choices} classes (see Section 24.5.3) that are implemented in \texttt{urbansim}.

\texttt{lottery\_choices}

This class determines choices by taking capacity into account. The \texttt{run()} method takes as arguments \texttt{probability} which must be a 2D array (number of observations $\times$ number of (sampled) alternatives) and \texttt{resources} which must contain an entry “capacity”. This entry should be a numpy array or list giving capacity of each alternative of the universe. \texttt{resources} can contain an entry “agent\_units” which is a 1D numpy array or list giving for each agent the number of units to be removed from capacity when placing this agent. Default value is 1 for each agent. \texttt{resources} can also contain an entry “index” which is a 1 or 2D array of indices of alternatives for each agent. If it is a 1D array, it is assumed to be the same for all agents. If alternatives were previously sampled, it is the index of the sampled alternatives within the universe. The shape of “index” should correspond to the shape of \texttt{probability} or its second dimension if 1D. If no “index” entry is given, it is created for all alternatives given by “capacity” and appended to \texttt{resources}.

The \texttt{run()} method invokes the \texttt{opus\_core} module \texttt{random\_choices\_from\_index} which returns indices of agent’s choices made independently of one another. The indices are chosen values from the entry “index”. Using capacity, it is then determined for which choices there is an overflow in agent’s interest. If there is no overflow the method simply returns the obtained indices. Otherwise, from agents that selected choices with overflow, a number that corresponds to the overflow is randomly sampled. Those agents are marked for making a new choice. The probability cells that correspond to the not available alternatives are set to zero and the process is repeated. The maximum number of iterations of this process can be controlled by the entry “lottery\_max\_iterations” in \texttt{resources} which is by default 3. If there is still overflow after reaching the maximum number of iterations, in the resulting array there is a value -1 for agents that couldn’t find any choice.

\texttt{first\_agent\_first\_choices}

This class is a child of \texttt{lottery\_choices}. It allows only one agent per choice. In case of overflow, the first agent keeps the choice, all others must choose again.
Part VII

Information for Software Developers
Programming in Opus and UrbanSim

This chapter provides some practical details on developing programs in Opus and UrbanSim, including how to start a new Opus package, design guidelines, and Python coding standards.

26.1 Creating New Opus Packages

This section provides information on constructing a new Opus project.

An Opus package is a Python package that conforms to the Opus conventions. If you are only creating new child XML configurations and modifying them with the GUI, you don’t need to make a new package. However, if you’re writing Python code to define new variables as Python classes, to define new datasets, or to define new models, then you should construct a new Opus project to hold this code. The new package should be placed in your workspace, alongside of the Opus and UrbanSim packages such as opus_core and urbansim.

For instance, to create an Opus package for the fictitious “atlantis” MPO:

1. Use the following Python commands to create a new ‘atlantis’ Opus package pre-populated with the structure and files expected in an Opus package (replace “atlantis” with the name for your Opus package):

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

   This will create a new directory, e.g. ‘C:/opusworkspace/atlantis’, containing a set of commonly useful files and directories, though some may not suitable to your application. For instance, you might want to delete the following files that are useful in the University of Washington research infrastructure, but perhaps not in yours:

   `.project` Eclipse’s container for project-specific information (see http://www.eclipse.org/).

   `build.xml` Directives for the CruiseControl automated build system used by the UrbanSim developers.

   Each Opus package may have a different set of files and directories, depending upon what is needed. To illustrate what an Opus package may contain, consider the `psrc` package we created for the Puget Sound Regional Council’s (PSRC) application of UrbanSim. Here are some of the directories and files it contains. (This material is in flux — some of these are being superseded by information in the XML configuration file, e.g. the run configurations.)
26.2 Unit Tests

This section describes how Opus developers write unit tests for Opus code. One of the goals of the Opus project is to practice test-driven development, wherein we write tests before we write any code. This practice is becoming increasingly common, since it has many advantages including dramatically increasing the ability to change code without breaking code.

In general, each Opus module includes just two classes: a class defining the functionality of this module, e.g. MyClass, and a test class containing unit tests of MyClass. The contents of this module are structured like this:

```python
county/
__init__.py  <-- Makes this into a Python package
de_population_DDD.py  <-- An Opus variable
...
docs/
<-- PSRC-specific documents EMME\2/
<-- Miscellaneous stuff for PSRC’s EMME\2 use estimation/
Scripts for estimating PSRC’s models
__init__.py  <-- Makes this into a Python package
estimate_dm_psrc.py  <-- Script to estimate the developer model
estimate_elcm_psrc.py  <-- Script to estimate the employment location choice model
...
faz/
<-- Additional variables for the FAZ
dataset_gridcell/
<-- Additional variables for the gridcell dataset household_x_gridcell/
<-- Additional variables for the household_x_gridcell dataset
indicators/
<-- Scripts to generate indicators
large_area/
<-- Variables for the large_area dataset
run_config/
<-- Configurations for different simulation runs
baseline.py  <-- Configuration for baseline run
no_ugb.py  <-- Configuration for baseline without UGB
...
tests/
__init__.py  <-- Automated tests
all_tests.py  <-- Runs all of this Opus package’s tests
test_estimation_dm_psrc.py  <-- Automated tests for estimate_dm_psrc.py
...
utils/
<-- Miscellaneous utilities
zone/
__init__.py  <-- Additional variables for the zone dataset
opus_package_info.py  <-- Information about this Opus package
```
The file starts with the required GPL header. Then some imports. Then the MyClass class, followed by the test class whose name is the name of the primary class followed by “Tests”. Finally, the last two lines will cause the unit tests to be run if this module is invoked directly by Python.

Test classes should derive from `opus_unittest.OpusTestCase` in order to ensure that the test runs in a “clean” environment. In particular, `OpusTestCase` removes all singletons before and after each test, so that a singleton created by one test will not accidentally be used by another test.

`OpusTestCase` also provides the following helper methods:

- `assertDictsEqual(first, second, *args, **kwargs)`: This is like the normal `assertEqual`, except that it also works for Python dictionaries that may contain numpy data.

- `assertDictsNotEqual(first, second, *args, **kwargs)`: This is like the normal `assertNotEqual`, except that it also works for Python dictionaries that may contain numpy data.

Tests that are defined in this manner will be automatically found and run by the `all_tests` module when it is executed. Please ensure that the test case class (e.g. `MyClassTests`) is found outside of the `if __name__ == '__main__':` check, otherwise the tests will not be found.

For more examples, check out the files in the `opus_core` package.

### 26.2.1 Writing Unit Tests for a Variable

By convention, there are a set of “unit tests” for every Opus variable, and for every Opus model. These tests check that the variable or model is doing the right thing, and were useful to uncover bugs when we wrote the tests. In addition, the insides of the Opus system also include many unit tests to check that the framework works.
Here are some guidelines to writing unit tests:

- Create *just enough* data for the test. For instance, create three gridcells with two attributes on each gridcell. This makes it easier to write, understand and modify tests.
- Use data that has interesting differences; the test data does not have to be realistic.
- Use different test data for each test. Otherwise, changing the data to better serve one test can cause problems with a different test.

The *urbansim* package contains many examples of variables and their tests. For instance, see the Python modules in *urbansim.gridcell*.

When writing a variable, be careful about forcing the values of a variable’s computation to be within a specified range. This can hide errors. For instance, when we were developing UrbanSim, we initially used numpy’s `clip` function to force the `vacant_industrial_sqft` variable values to be at least zero. However, it turned out that there was a bug in our code that led to more industrial jobs being allocated to a gridcell than the gridcell’s industrial sqft could support. This sometimes led to a negative value of `vacant_industrial_sqft` for that gridcell. Since these negative values were being forced to zero, however, we saw an un-intuitive result where the number of industrial jobs was going up at the same time as the `vacant_industrial_sqft`, even though no new industrial sqft was being built. Removing the clip made it easier to diagnose the problem.

### 26.2.2 Stochastic Unit Tests

Testing stochastic systems presents challenges for the traditional unit testing framework, since in general either the tests are trivial, or else they will sometimes fail. A research project by our group developed a methodology to put stochastic unit tests on a firm statistical basis, and this is now used in writing such tests in Opus and UrbanSim. See Hana Ševčíková, Alan Borning, David Socha, and Wolf-Gideon Bleek, “Automated Testing of Stochastic Systems: A Statistically Grounded Approach,” in *Proceedings of the International Symposium on Software Testing and Analysis*, ACM, July 2006 (available from [http://www.urbansim.org/papers/sevcikova-istta-2006.pdf](http://www.urbansim.org/papers/sevcikova-istta-2006.pdf)).

A result of this is that, indeed, such stochastic unit tests may fail, even though the underlying code is correct. This formerly resulted in a periodic failure, which would go away when we ran the test again. To automate this process, the `run_stochastic_test` method in the `StochasticTestCase` class now includes a keyword parameter `number_of_tries` with a default value of 5, which simply tries again that number of times. Since introducing the `number_of_tries` parameter this issue has not come up.

### 26.3 Programming by Contract

Several of the types of Opus objects (variables, models) have optional `pre_check` and `post_check` methods that allow you to use a “programming by contract” model of software development. In this technique a piece of code, such as a variable definition, can define a “contract” that specifies that if a given set of pre-conditions are met, the code is guaranteed to meet a given set of post-conditions. This “contract” can help (a) reduce debugging time by checking for invalid inputs or outputs, and (b) document the assumptions and behavior of the code.

Opus implements this programming by contract mechanism by automatically invoking the `pre_check` and `post_check` methods of variables, models, and simulations. This is done only if the Resources being used contain the “check_variables” key.

**pre_check** This method for an Opus variable is called before the variable’s `compute` method. This is a place to put optional tests of “pre-conditions” that this variable assumes in order to operate properly.
post_check  This method for an Opus variable is called after the variable’s compute method. This is a place to put optional tests of “post-conditions” of what this Opus variable “guarantees” it will produce given that the “pre-conditions” tested in the pre_check passed.

26.4 Design Guidelines

These design guidelines are key rules that, if followed, will help you create code that is easy to develop and to maintain.

• Practice test-driven development. Before you write any code, write an automated test that tests if that code works. Use this for new features as well as bug fixes.

• Include automated unit tests. A unit test tests an internal part of the code, such as a method. A method is a contract: given certain assumptions about the data used by the method, the method is guaranteed to produce certain data. A good rule of thumb is to have at least one unit test for every contract of every method. As always, use your judgment as to what tests to write.

• Include automated acceptance tests.

• Never duplicate code. Avoiding duplication results in lots of good, such as decoupled designs.

• Favor composition over inheritance.

• If you can think of a reason you want to use different versions of a method, convert the method to an class and use composition.

• Relentlessly keep your abstractions current with your understanding of the system.

• Have an abstraction for every “real world” object that the modeler interacts with, e.g. “a run request,” “a queue of run requests,” etc.

• Create decoupled designs. A class C should only know about another class B when C’s abstraction requires knowledge of B.

26.5 Coding Guidelines

These draft guidelines reflect what we believe makes for readable, understandable code. (Note that not all of the code in the current Opus code base follows these conventions at this point.)

The Enthought coding standards https://svn.enthought.com/enthought/browser/trunk/docs/coding_standard.py is a good start. In addition, we have a number of other suggestions particular to Opus or our preferences.

26.5.1 Indentation

Use spaces rather than tabs. (Python interprets a single space as the same level of indentation as a single tab, yet 8 spaces can look the same as 1 tab — which can lead to some very confusing debugging sessions otherwise.)

If for some reason you need a tab inside a string, use ‘\t’. This makes it easy to automatically check for and complain about tabs in files.

Indent 4 spaces for each successive level of indentation.
Indent nested functions.

Use nested functions only for very minor, short ones.

26.5.2 Naming Conventions

Use the following naming conventions for class, method, and variable names:

- **Normal Case Sensitivity**
  - Class names: `ClassNamesLikeThis`
  - Constants: `CONSTANTS_LIKE_THIS`
  - Instance variables: `instance_variables_like_this`
  - Method variables: `method_variables_like_this`
  - Semi-private method variables: `__semi_private_method_variables_like_this`
  - Private method variables: `__private_method_variables_like_this`
  - Public method names: `public_method_names_like_this`
  - Semi-private method names: `__semi_private_method_names_like_this`
  - Private method names: `__private_method_names_like_this`

Exceptions are the class names for Opus variable classes (e.g. `total_land_value`), for model component classes (e.g. `linear_utilities`) and for pluggable model building blocks (e.g. `mnl_probabilities`). These all are lower case with underscores, since users specify these classes by strings and our experience is that it is much less confusing if strings are case-insensitive.

If a variable name won’t be used but is needed, such as when a method returns a pair of values of which only one is used, start the variable name with `dummy`, as in:

```python
coef, dummy = cm.estimate(specification, ...)
```

It is important for a class to clearly communicate its intended usage patterns. Part of having a well defined interface is being clear about which methods and which instance variables may be used from where. This helps others understand how to use the class, which helps prevent usage errors. Some methods, for instance, should only be called from within that class; perhaps they can only safely be called with data that is private to that object.

Python provides three types of method names that help span the public to private spectrum. Opus’ convention is to use these three types of method names, as follows:

- **Public Methods**
  - `def foo(self)`: This is a public method callable by anyone. Others objects can always call this and expect it to do the right thing. This corresponds to Java’s `public` methods.

- **Semi-Private Methods**
  - `def _foo(self)`: This is a semi-private method. It has a single leading underscore. It should only be called by another object that is closely related to this class, either by inheritance or by being in the same Python package. This corresponds to Java’s `protected` and `package` methods, respectively.

- **Private Methods**
  - `def __foo(self)`: This is a private method. It has two leading underscores. It should only be called from within this class. This corresponds to Java’s `private` methods.

Always use setters and getters to access another object’s instance variables. Accessing instance variables directly ties the outside object to the particular implementation of the instance variable. For instance, the object can no longer decide to compute the variable on each access. This leads to code that is harder to extend and maintain.
Class names should not start with an underscore.

Python has no special syntax or semantics for abstract classes, but good practice is to explicitly name a class as abstract, e.g. `AbstractModel`, if it is abstract.

Use lower case names with underscores for file names, e.g. `my_stuff.py`, for improved readability, and because some file systems are case sensitive while others are case insensitive. Beware that some file systems have limits on the length of file names, so don’t make them too long.

Variable names should be at most 64 characters long, since MySQL does not allow column names with more than 64 characters.

Historical naming issues:

- Method names `thatStartWithALowerCaseLetter` are deprecated, although it is not urgent to change them.
- Method names `ThatStartWithACapitalLetter` should be renamed, since this style is used for class names instead.

### 26.5.3 Methods and Side-effects

A common type of bug that can be very difficult to find is when a method changes the value of a variable that is defined outside that method. The next two guidelines help prevent these bugs.

Methods that return meaningful values should not have any side effects, except for benign side effects. In programming language jargon, they should act as functions. Benign side effects that don’t change the values produced, such as caching a value to speed up processing, are fine though be careful that the lifetime of the values in the cache is what you want.

Methods that have no side effects should return any values. The one exception is some cases where a status code is returned.

### 26.5.4 Documentation and Comments

Have a doc string for every public class and method. Doc strings for internal methods are optional – use your good judgment.

Include informative comments, of course. Comments should be at the right level — comment entire methods and potentially obscure bits of code; don’t comment obvious things.

### 26.5.5 Source Files and Packages

Python source files may contain more than one class, if the classes are closely related. If a file is getting too long, break it into several files however. Exception: files containing an Opus variable definition must contain just the one class defining the variable.

Use Python’s name space mechanisms, such as modules and packages, to handle name space issues. Use `import myfile` or `from myfile import MyClass`. Don’t use `import *` in code in the repository. (It’s convenient to use at the console for interactive debugging however.)
Avoid `import as`, since it makes it harder to figure out what a name refers to. If there is a name clash, use a qualified name instead. For example, `numpyp` and `numpyma` both have a log function. Rather than using `from numpyp import log as nlog, use import numpyp, and then within your code qualify the names, e.g. `numpyp.log`.

Do not use multiple-line imports, such as:

```python
from numpyp import array, repeat, transpose, ndarray, reshape, \ 
indices, zeros, float32, asarray, arange, compress, \ 
logical_not, cumsum, log, where, ones, strings
```

These multiple line imports make it hard to search for imports of specific functions, since the “import” is separate from some of the function names. This in turn increases the chance of missing some instances of things you might want to change.

### 26.5.6 Typing Issues

Python is type-safe but dynamically typed — no type declarations are needed. This gives flexibility and good support for rapid development, but it’s also easy to write code that is difficult to understand and maintain. Here is a general rule of thumb: it should be possible to write the type signature of any Python variable or method, in a way that lets you statically check that there aren’t any type violations. (Python doesn’t require or even support such declarations, of course; this is a rule.) There are times you will want to violate this guideline, but make sure you have a good reason to do so.

For non-computer-science types, here are some practical implications of this heuristic:

- Use an actual boolean (`True` or `False`) when you need a boolean value, not 0, 1, the empty string, or the other things that Python allows for booleans.
- If you have a method that takes a sequence of some kind, let it take sequences only. Don’t generalize the method to also allow single values that get coerced into a sequence — this is confusing, and also leads to bugs.
- If you are writing code that uses `type(x)` or `is`, there is very likely a better way to write it (using objects and inheritance).
- Don’t use the AND hack from [Dive Into Python](http://diveintopython.org).

The ubiquitous “resources” variable is a violation of this rule, of course (which is why there is something a bit off about it); but for now, we just let this be an exception.

There are also a few typing style recommendations.

Tuples should in general just be used locally, if you are counting on elements being at specific positions — if you need to pass around a tuple of this kind, define a small class instead. For example, this is a reasonable use of a tuple (for formatted text):

```python
'Unexpected value -- squid was %s but %s was expected' % (a,b)
```

Here the tuple is used quite locally, and is clear.
However, if you find yourself passing around a tuple and then writing things like `squad[4][2][8]`, see if there is a better way to do it. Similarly, lists used to package up a fixed number of elements, with the positions being significant, are also suspect. (An exception is picking apart the results from SQL queries — here you’re stuck with picking it apart.)

### 26.5.7 Strings

When constructing strings by combining string parts and values, use the `%` operator instead of the `+` operator, e.g.

```python
'Could not find %s in file %s' % (value, file_name)
```

not `'Could not find ' + value + ' in file ' + file_name.

When quoting strings, favor single quotes (`'`) over double-quotes, except where that doesn’t make sense.

### 26.5.8 Functional Programming Style where Appropriate

Use list comprehensions when feasible. Use functional programming style in lambda functions.

### 26.5.9 Classes and Methods

Always use “new-style” Python classes (see http://www.python.org/doc/newstyle.html), as they have some subtle and important benefits over the old-style Python classes.

Always have an `__init__` method for each class. Make it the first method of the class. (If there isn’t anything to do, just put pass in the body.) Name, and initialize, all instance variables in the `__init__` method.

Use keyword parameters in method calls, as it is more readable. In method definitions, use defaults as appropriate, but keep the design clear and understandable. If the parameter is required, just don’t put a default.

Ideally, a default for a given parameter should occur in just one place, and not be repeated in multiple method or function definitions. (In the other definitions, use a required parameter instead and pass along the default from the other method or function.) For example, suppose some method has a parameter that includes a default value: `replace_nulls_with=0.0`. Other methods that have the `replace_nulls_with` parameter should require it, and use the value passed along from the method in which it was given the default. Otherwise, if you want to change the default, there would be lots of places to change. If it turns out to be awkward to follow this rule, and you really do need to give a default in lots of places, give it a name and use that name in the parameter list, e.g. `replace_nulls_with=self.default_null_replacement_value`.

We’ll delete the following example in a bit — but there are quite a few places in the existing code that use keyword parameters in more complex ways than need be, e.g. by setting the default to `None`, and then having an if in the code that tests for `None` and if so sets the parameter to a default. So I put in the discussion for the moment.

For example, consider the following. (Any resemblance to actual Opus code is purely coincidental.)

---

272 Chapter 26. Programming in Opus and UrbanSim
This has various useful defaults for the parameters, but could be improved.

Note that the treatment of defaults is simplified — having optional parameters that set the values of other optional parameters should be avoided. Also, putting the defaults in the method header makes it clearer what is going on. Naming the parameter database and then naming the corresponding instance variable database_name is confusing. Finally, the comment in the old code about the database parameter is incorrect — if a parameter required, just don’t put in a default value. Python still allows using a keyword for it in the method call, and the keyword parameters can be in any order. You’ll get a failure if the required parameter is missing, which is the desired behavior.

(There are other issues as well, for example, why is there something about silage in opus_core, but we’ll ignore that for the moment.)

If a variable is the name of an object, include name in the object’s name, e.g. variable_name, to help distinguish it from a variable that contains the object itself, e.g. variable.
26.5.10 Continuation Lines

For readability, only use Python’s backslash (‘\’) continuation character when necessary. This character is used to split a statement or expression across more than one line. It is not necessary when the line break is inside a (), [], or pair, or in triple-quoted strings.

26.5.11 Open Issues

- Need to specify naming convention for items in resources; data model. *Note that the resources concept is being replaced by Traits-based configuration classes. This provides an explicit API for each configurable class, and avoids many of the problems associated with the prior use of resources.*

- Conventions for throwing exceptions. (We should make much more use of Opus-specific exceptions, rather than generic exceptions like StandardError.)

- Put all SQL strings into a single or small number of isolated files, so that it is easier to localize these strings to different databases.

- Follow the conventions in Python’s PEP 290: Code Migration and Modernization PEP 8 ([http://www.python.org/peps/pep-0290.html](http://www.python.org/peps/pep-0290.html)) For instance, when searching for a substring, where you don’t care about the position of the substring in the original string, using the in operator makes the meaning clear, e.g. `use string2 in string1 instead of string1.find(string2) >= 0 or string1.find(string2) != -1`. Also, when testing dictionary membership, use the ‘in’ keyword instead of the ‘has_key()’ method. The result is shorter, more readable and runs faster.

- Follow the conventions in PEP 8: Style Guide for Python Code PEP 257 ([http://www.python.org/peps/pep-0008.html](http://www.python.org/peps/pep-0008.html))

- See Python’s PEP 257: Docstring Conventions ([http://www.python.org/peps/pep-0257.html](http://www.python.org/peps/pep-0257.html)).
Writing Documentation

This chapter provides information and guidelines on writing Opus and UrbanSim documentation. This guide is written using \LaTeX. Other documentation includes a growing number of tutorials, and documentation on individual indicators. Tutorials can be written using a variety of formatters, including \LaTeX and wiki; guidelines for writing tutorials is on our trac wiki at http://trondheim.cs.washington.edu/cgi-bin/trac.cgi/wiki/ContributingTutorials. Documentation for individual indicators is written using XML files; guidelines for writing indicator documentation are given in Section 27.6.

We use \LaTeX for the guide and other major documents, since it produces elegantly formatted printable document, handles mathematics well, and includes facilities for cross-referencing and indexing. In addition to a pdf version of the documentation written using \LaTeX, we use the latex2html converter program to produce an html version.

27.1 The Reference Manual and User Guide

The Opus and UrbanSim 4 documentation in this guide follows the Python documentation standards (or at least it’s supposed to), and is written using \LaTeX and the Python documentation macros. The latest version of this Guide (as a pdf file), and the Opus tutorial, is automatically posted on the UrbanSim website at http://www.urbansim.org/ whenever a new version of any of the manual’s components is checked in to the subversion repository. There is also an html version produced using latex2html, also produced by the same build script.

The Guide includes a table of contents and a (currently rudimentary) index. The Adobe pdf reader also includes a nice full-text-search tool that can, of course, be used on this manual when reading the pdf version online.

See Also:

Documenting Python
(http://docs.python.org/doc/doc.html)

This document describes how to document Python using the \LaTeX macros developed for the purpose.

Python Documentation Download page
(http://docs.python.org/download.html)

The download page provides links for downloading the Python documentation in various formats, in particular \LaTeX source. The macros in the “texinputs” subdirectory there have been copied into the ‘docs/latex’ directory (and occasionally modified) for use in formatting this Guide. Other manuals there are also useful as examples of using the macros.
27.2 Using Python Macros

The Python macros include logical markup for the elements of the language and related concepts. Use the \LaTeX macros defined in python.sty to format source code, along with function, method, and file names, and so forth. This ensures a uniform appearance. Also, the Python documentation states that sometime in the future, the Python documentation is likely to be converted to XML or some other structured markup, and using the logical markup now will facilitate the transition. Here are some useful examples of such macros. See the “Inline Markup” section of “Documenting Python” for the complete list (http://www.python.org/doc/current/doc/inline-markup.html).

\module{spam}
\file{parrot.py}
\function{setup()}
\method{append()}
\member{sproket}
\program{pkgtool}
\url{http://www.conglomocorp.com}
\kbd{Control-D}
\samp{import sys; sys.exit()}
\code{sys.argv[0]}
\class{Exception}
\character{\n} % this is for \n
\samp{\program{python} \programopt{-m} \var{module} [arg] ...},
\keyword{kwd}
\exception{expt}
\class{Giraffe}
\pytype{string} % built-in Python type
\UNIX % the system
\Cpp{}

In addition to the standard Python macros, we’ve redefined the “package” macro to also add the package to the index, and added another macro “variable” for formatting Opus variables and expressions. These are defined in the file ‘opus_docs/manual/macros.tex’.

27.3 Writing Email Addresses in Documentation

The Reference Manual and User Guide is published on the UrbanSim website in both pdf and html form. To avoid serving as a spam magnet, please don’t put real email addresses in the document – instead use circumlocutions of some sort, e.g.

info (at) urbansim.org

27.4 Indexing

Ideally, terms should be indexed as documentation is written. Use \index{ ... } and related commands to accomplish this. Consider what would be useful for a reader of the manual in deciding what and where to place index commands. For example, for “primary attribute” there should probably be an index entry where this term is first used.
or defined, but not every place it is used. (There’s full text search for that after all.)

The `\index{...}` command produces no text where it is written, and it’s best to put the command next to the word it indexes, with no spaces between them, so that the page number isn’t off by one if the word begins or ends a page.

Some particular cases:

**Subentries** For index terms with a subentry, use this form: `\index{MySQL!column name lengths}`.

**Cross-references** Entries that are simply “See” and another entry can be created using the vertical bar with this command: `\index{IDE|see{Interactive Development Environment}}`. These commands are probably best placed in the glossary next to the given term.

**Font changes** Use the standard Python macros to get the correct font for file names, package names, and so forth. For example, LaTeX can be rendered using its traditional typography in the index using `\index{latex@LaTeX{}}`. For file names, classes and similar terms whose nature isn’t immediately obvious, put an additional appropriate term in the index entry. For example `\index{Model@class{Model} class}` generates “Model class” in the index.

The index is in two-column format, so for very long words in the index (e.g. class names), you can suggest hyphenation points to \LaTeX.

## 27.5 Some Other Guidelines

Here are a few other guidelines. *These are mostly noted as a result of noticing issues in the existing documentation ... later, we can have a more coherent list.*

- Use the `\begin{verbatim}` environment to format larger blocks of source code.
- Use logical rather than physical markup, for example the `\emph{...}` environment for emphasis (italics), rather than `\em` or `\it`.
- Use the `\ldots` macro to produce three dots (…) indicating ellipses — if you just write three periods in a row they are spaced too closely (for example: ...).
- The latex2html program handles the mathematical typesetting from \LaTeX by including the math as a bitmap. This is reasonable for actual equations, but looks a bit odd for simple symbols in the middle of a sentence. There should be a switch to turn these into ordinary text, or in the extreme we could write alternate latex or html macros. However, if this can’t be fixed, avoid using the math typesetting environments if you don’t need them. For example, to produce a backslash, use `\textbackslash` or `\verb|\\|` rather than `$\backslash$` (since in latex2html the first two options produce a character, the third a GIF image).
- Do not use underscores in `.tex` filenames. Use hyphens instead (i.e. use ‘-’ instead of ‘\_’).

Normally, a good heuristic is to copy the existing examples in the manual when writing new ones. However, much of the LaTeX here doesn’t yet take advantage of these macros. So look at the description linked above instead for now.
27.6 Writing Indicator Documentation

Documentation on an individual indicator is contained in an XML file, to make it convenient to browse to the indicator documentation on the web.

These XML files go in the opus_docs project, in the directory ‘opus_docs/docs/indicators’. The new indicator should also be added to the ‘predefined_indicators.xml’ file in an appropriate category. (At the moment we don’t support having multiple indicator directories, e.g. for specific applications of UrbanSim, but we will probably add this in the future.)

The indicator documentation is linked from http://www.urbansim.org/. There is also a link to a web page, “Reading Indicator Documentation,” which briefly describes each section of the indicator documentation. It is intended for people who are reading the documentation (as opposed to writing it). Each section of the documentation for an individual indicator also includes a link to the corresponding section of the “Reading Indicator Documentation” web page.

The technical documentation for each indicator should be relatively neutral. There is also an experimental Indicator Perspectives framework, which allows different organizations to present their perspectives on which indicators are important and how to interpret them. Support for this may be added to UrbanSim 4 in the future. The mechanism is outlined in the following paper: Alan Borning, Batya Friedman, Janet Davis, and Peyina Lin, “Informing Public Deliberation: Value Sensitive Design of Indicators for a Large-Scale Urban Simulation,” Proceedings of the 9th European Conference on Computer Supported Cooperative Work, Paris, September 2005. (Available from http://www.urbansim.org/papers/.)

The technical documentation for each indicator includes a definition, related indicators, limitations, and other information. The web display also includes links to the Opus variable or primary attribute for that indicator.

Required elements in the XML are marked as required; if these are missing a note in red (such as “specification missing”) will appear in the rendered web page. Other elements are optional; if omitted the heading will still appear in the rendered web page, with an appropriate note. The DTD file that formally specifies the syntax of an XML indicator definition is ‘indicator-declaration.dtd’; the file ‘indicators.xsl’ specifies how the indicator is rendered on a web page when browsing.

**Definition** *(required)* A short, accurate definition of this indicator. In the rendered web page, this appears directly under the indicator name (there isn’t a separate heading “definition”).

**Principal Uses** What are the principal uses of this indicator? (This element should say whether the indicator is useful for policy evaluation, or mostly diagnostic. An indicator is useful for policy evaluation if it is useful in evaluating whether particular policy goals are supported or not by different scenarios. An indicator is useful for diagnosis if it has a clear directionality given different test input scenarios.)

**Interpreting Results** Describes how to interpret the results from this indicator, either from a diagnostic perspective, a policy perspective, or both.

**Display Format** *(Appears as “Units of Measurement and Precision”)* This section provides a machine-readable specification of the default display for the indicator data. Since it is machine-readable, it must use a set of XML elements, rather than being free-form English text. The first element should specify the units used, e.g. `<units>square feet</units>` or `<units>persons/acre</units>`. If the data is unitless then this element should be `<unitless/>` (this would be the case for ratios, e.g. a vacancy rate). The next element should specify how numbers should be displayed, and how many decimal places of precision should be used. The options here are `<numberdigits="N"/>`, `<percentage digits="N"/>`, or `<scientific digits="N"/>`, where N in each case is the number of digits. For percentages, this is the number of digits after the decimal place, so that if N=2 the value 0.08352222 would displayed as 8.35%.

**Specification** *(required)* A human-readable specification of how the indicator can be computed. This is not necessarily the same as how it is actually computed.
Limitations  Known limitations or problems.

Related Indicators  A short discussion of related indicators. If this indicator could be computed as the composite of other indicators, list them here, and describe the relation (see “Jobs per capita” for an example). Another example of using this category is in the “Employment density”, “Household density”, and “Population density” indicators, which all show different aspects of density. However, if this indicator is just related to others in its category, by virtue of being in that category, then that information can probably be safely omitted.

How Modeled  This element should describe whether the value of this indicator is determined by the interactions of UrbanSim’s component models, or whether it is exogenous (that is, is determined outside the operation of UrbanSim). In some cases, the value of the indicator will be exogenous for the regional geography, but not for sub-regional geographies. If this element is omitted, the default output is “Not specified. (In this case, normally one can assume that the value of this indicator is modeled by UrbanSim itself, as opposed to being exogenous, that is, coming from an external source such as a control total.)”

Indicator Source, Evolution, and Examples of Use  Where did the definition and code for this indicator come from? How has its specification evolved? If known, provide examples of using this indicator (including citations).

Source  This element gives a link to the source code for the indicator if it is defined as an Opus variable. If it is a primary attribute, the attribute should be listed (with no link), followed by “Opus primary attribute” in parentheses. (See for example the documentation for “Residential units”.)


Part VIII

Appendices
Installation

A.1 Installing Opus

Opus has been run and tested on Windows, Linux and Macintosh OS X. (It should also run on any other platform that Python and its libraries run on, but we’ve not tested others.) Instructions for installing Opus, and UrbanSim, which is bundled with it, are kept online at http://www.urbansim.org/download/. There is also a copy of the instructions in the source code tree — open the file `opus_docs/installation/index.html` in a web browser.

We provide both stable releases and development versions of Opus/UrbanSim. Each stable release has a unique version number, such as 4.1.2 (see Appendix B). Generally, we recommend using the latest stable release of Opus/UrbanSim. The system is under continuing development, and the other possibility is using the latest development version from our source code repository (not always recommended for the faint-of-heart).

A.1.1 Windows Installer

For users with a computer running Windows, we currently recommend installing the latest development version using the Windows Installer, which is downloadable from http://www.urbansim.org/opus/installer/opus_installer.exe. Step by step details for using the installer are available at http://www.urbansim.org/docs/installation. Follow these instructions on the web for the installation, and return to this document afterward for recommendations for testing your installation (Section A.1.3). The installer launches other installers for supporting packages during the install process, so you need to be connected to the Internet for the installer to download files during the installation. Please be patient — the installation takes 20-30 minutes depending on your connection speed.

A.1.2 Installing on Other Platforms

Step by step instructions for installing on other platforms (as well as on windows by hand rather than with the installer) are available at http://www.urbansim.org/docs/installation. First select either the latest stable release or the latest development version, and then the web page for your particular platform.

A.1.3 Testing Your Installation

Once Opus is installed, it is a good idea to test whether it is installed correctly. The installation involves downloading and installing numerous components (all managed by the Windows installer, if you’re using that). There are times when a component does not download properly and fails to install, and particularly when using the Windows installer
you might not notice this. Here are a few simple tests to check whether Python and the key libraries are properly installed.

1. Start a command window or shell. On Windows, this can be done from the start menu run and entering cmd.

2. Type python and press Enter (for brevity, I will not repeat the press Enter at the end of each command - it is implied). A Python prompt should appear, as >>>.

3. Type import numpy. It should just generate another Python prompt, which means it successfully imported the Numpy library. If it fails, you will see an error, and need to exit Python and reinstall Numpy. To see what version of Numpy is installed, type numpy.__version__.

4. Repeat the import step with the following libraries: scipy, matplotlib, sqlalchemy, and PyQt4 (the capitalization matters). If all of these import without reporting an error, the main libraries are installed correctly. The version number for sqlalchemy should be at least 0.4.3 at the time of this writing.

5. Finally, type import opus_core. This verifies that the opus_core package has been downloaded and that Python can find it. (If it can find opus_core, it can probably find the others ...)

To exit an interactive Python session, type quit(). (As a shortcut for exiting Python, you can instead type control-z on windows, or control-d on Macintosh or linux.)

A.1.4 Updating the Software

The software is being continuously developed and refined, with bug fixes and new features being added. In order to get these updates, or to download and install other Opus Packages not initially installed, users will need an application that can retrieve updates and packages from the Subversion repository for the software. Subversion is a version control system used by many software development projects, and there are many free software tools available for accessing it. One of these, on the Windows platform, is Tortoisesvn, available from http://tortoisesvn.tigris.org/. Users can download and install this package, and it will extend the Windows Explorer file browser, allowing packages to be checked out of the repository (downloaded for the first time), or updated if they have already been checked out previously. Subversion compares the version on your local machine with the version on the server, and if there is a newer version of any component, it will download it and replace the version on your machine with the most current one available.

To check out (download for the first time) any package from the Opus repository, if you have installed Tortoisesvn, right-click on the opus/src folder in the Windows Explorer, and the menu will contain new entries for interacting with Subversion. Select the SVN Checkout option. For a stable release, say version 4.2.0, use the url https://trondheim.cs.washington.edu/svn/opus/tags/4.2.0, plus the name of any package you wish to check out. For a package from the latest development version use the url https://trondheim.cs.washington.edu/svn/opus/trunk, plus the name of any package you wish to check out. The target local directory should be under the Opus_HOME/src directory, and take the same name as the Opus Package you are checking out. An example of this checkout dialog is shown in Figure A.1. There are other Subversion clients available for Windows, OS X and Linux, and any of them should work equivalently well for obtaining Opus packages. Eventually, this functionality may be embedded into the Opus GUI.

Once a user makes changes on their local machine to any content that has been checked out from the repository, the changed file will be inconsistent with the version in the repository. In many cases, this is fine, but if the file is updated by someone else and checked into the repository, then the next time you update from the repository, you will have conflicting changes in the file you have modified. There are tools available to reconcile the conflicts, but the best approach for most users is to isolate the places you want to make changes to a location that will not have collisions with ongoing updates in the repository. A project_configs directory is intended for users to create and manage their own projects. If the user needs to modify Python modules within the various Opus packages, these can most easily be handled by creating another Opus package of your own that inherits from the package you want to change, and
Figure A.1: Checking Out a Package using the TortoiseSVN Subversion Client

contains only the modified versions of files. Note that Opus packages can be updated from the repository at any time by selecting the directory (or all of them at once), right-clicking, and selecting the menu item for SVN Update.
Version Numbers

Opus and UrbanSim use version numbers for the source code and for the XML project configuration files. You should use consistent versions of these, since the source code will expect that the XML files are laid out in a particular way.

A given source code version number corresponds to a specific version of the source code. For the XML configuration files, the version number identifies the expected layout of the XML, not the exact content of the configuration — if we were using DTDs or other XML schema descriptions, it would be the version of the DTD. (We plan to support such a schema in the near future.)

B.1 Source Code Version Numbers for Stable Releases

For stable releases, the version is of the form ‘4.2.1’, where 4 is the major release number (i.e. UrbanSim 4), 2 is the minor number, and 1 is the bugfix number. The version number of the source code is associated with the opus_core package, and can be found in the usual Python fashion by evaluating the following expression:

```python
opus_core.__version__
```

(In the future we may add the `__version__` attribute to all Opus/UrbanSim modules.)

B.2 Source Code Version Numbers for Development Versions

For development versions, the version for the source code is something like ‘4.2-dev5216’ where 5216 is the svn (subversion code repository) revision number for this version of the code. When this development version is first released as a stable release, it becomes ‘4.2.0’. The 4.2 series then enters bugfix and minor enhancement mode. Further stable releases in the 4.2 series will be bugfix versions (perhaps also with some minor enhancements), e.g. 4.2.1, 4.2.2, etc. When the code in the main trunk is ready to move to a major change in functionality, the version number for the main trunk will then move to the 4.3 series, for example 4.3-dev3377.
B.3 XML Version Numbers

XML version numbers are of the form ‘1.0’, where 1 is the major number and 0 is the minor number. These are independent of the source code version numbers.\(^1\) The version number for an XML configuration is given in an element with the `xml_version` tag, for example:

```xml
<xml_version>1.0</xml_version>
```

Currently this isn’t visible in the GUI itself — you need to view the contents of the XML file.

The system has a minimum and maximum XML version number for which the code is known to work, and checks that XML configuration version numbers are within this range when they are loaded. (If not, it raises an exception.) You can find the minimum and maximum XML version numbers that your code expects by evaluating

```python
import opus_core.version_numbers
opus_core.version_numbers.minimum_xml_version
opus_core.version_numbers.maximum_xml_version
```

B.4 Sample Data Versions

For the sample data for a stable release, the version number is built into the file name. For example ‘opus-4-2-0.zip’ is the sample data file for version 4.2.0. This file can be downloaded from the UrbanSim download page at http://www.urbansim.org/download/.

The download page will note whether this is still valid for the current development version; if there are changes to the sample data before a new stable release, there will be a new zip file with a name like ‘opus-4-2-dev-20jan2009.zip’ (which indicates that this is a sample data zip file for the 4.2 development trunk, posted on 20 January 2009).

The sample data file also includes starter XML configurations, so if the XML version number changes a new zip file will be posted.

**Information for Developers.** Please see the trac wiki for information on updating version numbers: http://trondheim.cs.washington.edu/cgi-bin/trac.cgi/wiki/VersionNumbersAndReleases.

\(^1\)We had a brief period of trying to keep them the same, but soon decided this wasn’t a good idea, since the code version numbers change more rapidly than the XML version numbers, and we didn’t want to require that users change the XML version numbers on all their configurations if there weren’t any other changes.
Introduction to Programming in Opus for Modelers

This chapter provides a gentle introduction to programming in Opus. It is intended not for software developers but for model users, who need to understand enough about the underlying software to be able to use all of the capabilities of the system, and to extend these capabilities.

C.1 Python

Python is a programming language developed initially by Guido van Rossum, and has become a very popular programming language with a large user and developer base. It has been adopted, for example, by ESRI as the main scripting language for ArcGIS. It is Open Source software, and is available from www.python.org. Python is an interpreted language, as compared to a compiled language. This means that when you start Python, you are launching the Python interpreter, which then interacts with commands typed in interactively, or with programs (scripts) loaded from disk.

Python can be launched in several ways:

- from the command shell by typing python, or from the start menu in Windows
- from IDLE, a light-weight Python Editor and Shell that comes with Python, which can be launched from the start menu
- using Scite, an editor that can also run Python programs
- using a sophisticated Integrated Development Environment (IDE) such as Eclipse, Wing, or Eric

To get started, launch IDLE from the Windows Start menu. It will launch a window as shown in Figure C.1. If you are not on a Windows computer, just start a command shell and type python to start an interactive Python session.

C.1.1 Expressions and Data Types

Python has several data types that are useful to understand as you begin to explore the language. These data types are:

- integers
- floats
- booleans
Figure C.1: The IDLE Python Shell
Begin with the basics. Python can do simple (or complex) calculations interactively, just as you might do with a calculator. How might you compute the sum of two numbers? Just type in a mathematical expression, and hit return. Try 2+2, and 3/2. Notice that the second answer is probably not what you want: the result is an integer, rather than a floating point, so it truncates. To get a floating point result, use a decimal place on at least one of the inputs.

```python
>>> 2+2
4
>>> 3/2
1
>>> 3/2.0
1.5
```

The results of expressions are easily assigned to a variable name, and used in further calculations. Notice that when you type an assignment of an expression to a variable, it does not print the value, but does assign it. You can print or use the value at this point.

```python
>>> a = 2+2
>>> a
4
>>> b = 3/2.0
>>> a+b
5.5
>>> (a+b)/2
2.75
```

Now examine what happens if we use operators that assert a statement that can be evaluated as True or False. This generates a Boolean data type.

```python
>>> a = 2
>>> b = 4/2
>>> a == b  #This asserts that a is equal to b, by using == instead of =
True
>>> a < b  #This asserts that a is less than b
False
```

We have seen so far three of Python’s data types: integers, floats, and booleans. Text is also managed in Python, using its string datatype. Strings can be assigned to variables, and used, for example to concatenate two strings, or to extract a portion of a string using the index. Note that Python uses index values starting from 0. If a single index value is used, then it identifies a single value in the string at the index position. If two values are used, the second identifies the ending index value. One behavior that is not intuitive is that the returned values are up to, but do not include the second index value.
There are three other data types in Python that are closely related: tuples, lists and dictionaries. Tuples contain a set of items that are indexed (like strings, above), but cannot be changed once defined. An example would be the days of the week, or the months of the year:

```python
>>> days = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
>>> days[3]
'Thursday'
```

Lists are almost the same as tuples, but they can easily be modified, and items can be added or removed. A To Do list would be an example:

```python
>>> todo = ['get groceries', 'do homework', 'paint wall', 'watch movie']
>>> todo[1]
'do homework'
>>> todo.append('read book')
>>> todo
['get groceries', 'do homework', 'paint wall', 'watch movie', 'read book']
>>> del todo[1]
>>> todo
['get groceries', 'paint wall', 'watch movie', 'read book']
```

Dictionaries are flexible data structures that store key:value pairs, like a standard dictionary stores words and their associated definitions. Entries can be looked up by their key. A phonebook provides a simple example. Note the syntax to add an entry. Also, pay attention to the use of different kinds of brackets for these different data types. They do matter, and will generate an error if the wrong one is used.

```python
>>> myPhoneBook = {'Mark':2439503, 'Julie':4309302, 'Jeff':3540693}
>>> myPhoneBook['Jeff']
3540693
>>> myPhoneBook['Mary'] = 3339999
>>> myPhoneBook
{'Julie': 4309302, 'Jeff': 3540934, 'Mary': 3339999, 'Mark': 2439503}
```
C.1.2 Python Modules, Packages, and Methods

Python commands can be used interactively, as we have just seen, but they can also be stored in a file, called a Python Module, provided that it ends with an extension of .py, and follows some basic formatting requirements, like the use of indentation to identify what statements belong in a block. More on this later.

Any Python statements you can execute interactively can be put into a Python module and then executed at the command line or loaded into IDLE or another program that can edit and execute Python modules (Scite is a good example). Say you have created the classic first program "Hello World" in a Python module, called hello.py. It would contain one line: print "Hello World", and would be executed by typing at the command shell prompt (not the Python prompt):

c:\> python hello.py
Hello World

Python contains many built-in methods, and we will explore some of them as needed. One of the most common and useful ones is the range method. It generates a list, with N entries in it, sequentially numbered, starting from 0. N is passed to the method as an argument, in parentheses, like this:

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

This is useful in programs in which you need to iterate over a list, or perform some function N times. Here is the first example in which formatting in Python is needed. We have to indent the line print i underneath the line for in in range(10): in order to make clear to the Python interpreter which lines of the script are to be repeated 10 times. If we put this into a loop.py module and want to print the word "Done!" at the end of the list, the script would look like this:

for i in range(10):
    print i
print 'Done!'

If we have saved this as loop.py, then we can run it at the command prompt by typing python loop.py, and would see the following output:

0
1
2
3
4
5
6
7
8
9
Done!
Now that we have used a built-in Python function, try writing one of your own. For example, you could compute the square of a number with a function like this:

```python
def square(n):
    return n*n

print square(111)
```

It would not be necessary to implement this particular method, however, since it is built into Python’s Math package. To use functions in the Math package, you have to import the functions or the whole package. Here are some options:

```python
>>> import math
>>> math.sqrt(9)
3
>>> from math import sqrt
>>> sqrt(9)
3
>>> from math import *
>>> log(9)
2.1972245773362196
```

Note the difference in usage depending on which way you choose to import a function. The last one, using a * to import all the functions, is acceptable for an interactive session, but is a poor choice in a complex program you write, because each imported function has a name that gets stored in a Python `Namespace` used to keep track of all the functions, and this can get cluttered or confusing if you happen to import from multiple packages and don’t realize that the same function name is used to do different things in two different packages. You will see imports of many packages in the Opus system. One of the main ones is covered in the next section: Numpy.

## C.2 Numpy

Numpy is a Python package (library) for processing multi-dimensional arrays. To set terminology, consider a spreadsheet in Excel or some similar package. A data value in a single cell would be a **scalar**, or a single-dimension **array** of length 1. A column of 10 numbers would be a single-dimensional array of length 10. A sheet of 10 columns by 15 rows would be a two-dimensional array of shape (10, 15). If we then use 5 separate worksheets with each one containing a 10 by 15 worksheet, we have a three-dimensional array of shape (10, 15, 5). Numpy is designed to create, manipulate, and efficiently compute on these arrays.

Let’s begin a tour of Numpy by importing it and creating a small array, and then finding its size and shape, and computing some built-in Numpy functions on it. Note that we can easily manipulate the shape of an array.
There are numerous ways to create arrays, including some built-in functions for frequently used arrays containing 0’s or 1’s, or loading data from a file or even a database (with a bit more work).

Notice that you can perform mathematical operations on arrays, and that the default is to compute results elementwise, or element by element, like you would do in a spreadsheet. You can also use matrix computations as in linear algebra, with a slightly different syntax. A matrix multiplication of arrays A and B would be: \( \text{dot}(A, B) \). In the examples below, note the use of assignment to a variable (using an \(=\)), as compared to the use of an assertion that two arrays are equal (using \(==\)). Also note the use of a where statement, which allows assignment of different values where the statement is evaluated as true or false. These examples cover some of the more commonly used manipulations of arrays in Opus and UrbanSim.
```python
>>> a = array([20, 30, 40, 50])
>>> b = arange(4)
>>> c = a - b
>>> c
array([20, 29, 38, 47])
>>> b ** 2
array([0, 1, 4, 9])
>>> 10 * sin(a)
array([ 9.12945251, -9.88031624, 7.4511316 , -2.62374854])
>>> a < 35
array([True, True, False, False], dtype=bool)
>>> where(a < 35, 1, 0)
array([1, 1, 0, 0])
>>> d = array([10, 30, 20, 50])
>>> a == d
array([False, True, False, True], dtype=bool)
```

Numpy provides sophisticated numerical capabilities for doing statistical or econometric modeling, with a very concise syntax. An example of a tutorial script to estimate the parameters of a multiple linear regression, using Ordinary Least Squares estimation, is shown in the Appendix to this chapter. The script is from http://www.scipy.org/Cookbook/OLS.

Numpy contains a very large number of built-in functions. The following, for example, are some built-in random number functions:

- `beta()`, `binomial()`, `gumbel()`, `poisson()`, `standard_normal()`, `uniform()`, `vonmises()`, `weibull()`
- `rand()`, `randint()`, `randn()`
- `random_integers()`
- `random_sample()`
- `ranf()`
- `sample()`
- `seed()`

For a more complete tutorial on Numpy, please refer to the one at http://www.scipy.org/Tentative_NumPy_Tutorial.
Selected Methods and Functions in opus_core

This chapter lists selected methods and functions implemented in the opus_core package. For more information and information about arguments see our Trac site at www.urbansim.org (Browse Source).

D.1 Dataset

The class Dataset is implemented in opus_core.datasets.dataset.py. It is a child of AbstractDataset (opus_core.datasets.abstract_dataset.py) and most of its methods is inherited from the parent class. Here we group the methods according to their use cases, thinking about a dataset as a table with rows and columns.

D.1.1 Selected Methods

Adding columns
add_attribute (data, name, ...)
add_primary_attribute (data, name)

Adding rows
add_elements (data, ...)

Obtaining columns
get_attribute (name)
get_attribute_by_id (name, id)
get_attribute_by_index (name, index)
get_multiple_attributes (names)
get_id_attribute()
get_attribute_as_column (name)

Obtaining rows
get_data_element (index, ...)
get_data_element_by_id (id, ...)

Obtaining column names
get_id_name ()
get_attribute_names ()
get_known_attribute_names ()
get_primary_attribute_names ()
get_computed_attribute_names ()
get_nonloaded_attribute_names ()
get_attributes_in_memory ()
get_cached_attribute_names ()
get_attribute_long_names ()

Modifying columns
modify_attribute (name, data, index=None)
set_value_of_attribute_by_id (attribute, value, id)

Deleting columns
delete_one_attribute (name)
delete_computed_attributes ()

Deleting rows
remove_elements (index)
subset (n, is_random=False)
subset_by_index (index, ...)
subset_by_ids (ids, ...)
subset_where_variable_larger_than_threshold (attribute, threshold=0, ...)

Computing variables
compute_variables (names, dataset_pool=None, ...)
compute_one_variable_with_unknown_package (variable_name,
                                           dataset_pool=None, package_order=None)

I/O methods
load_dataset (... attributes=None, in_storage=None, in_table_name=None, ...)
load_dataset_if_not_loaded (... attributes=None, in_storage=None,
                          in_table_name=None, ...)
write_dataset (... attributes=None, out_storage=None, out_table_name=None,
               ...)
flush_attribute (name)
flush_dataset ()
load_and_flush_dataset ()
ge_cache_directory ()
remove_cache_directory ()
Obtaining row indicies
get_id_index (id)
try_get_id_index (id, return_value_if_not_found=-1)
get_index_where_variable_larger_than_threshold (attribute, threshold=0)
get_filtered_index (filter, threshold=0, index=None, dataset_pool=None, ...)

Connecting two datasets
connect_datasets (dataset)
join (dataset, name, join_attribute=None, ...)
join_by_rows (dataset, ...)
aggregate_dataset_over_ids (dataset, function='sum', attribute_name=None, constant=None)

Data analysis
summary (names=[], ...)
size ()
attribute_sum (name)
attribute_average (name)
aggregate_all (function='sum', attribute_name=None)
correlation_matrix (names)
correlation_coefficient (name1, name2)
categorize (attribute_name, bins)
get_data_type (attribute, ...)

Plotting methods
plot_histogram (name, main='', filled_value=0.0, bins=None)
r_histogram (name, main='', prob=1, breaks=None)
plot_scatter (name_x, name_y, main='', npoints=None, ...)
r_scatter (name_x, name_y, main='', npoints=None)
r_image (name, main='', xlab='x', ylab='y', min_value=None, max_value=None,
       file=None, pdf=True)
plot_map (name, ...)

Memory management
items_size_in_memory ()
unload_attributes (names)
unload_all_attributes ()
unload_primary_attributes ()
unload_computed_attributes ()
unload_one_attribute (name)

Other methods
get_dataset_name ()
get_attribute_header (name)
filled_masked_attribute (name, filled_value=0)
get_version (name)
has_attribute (attribute_name)
get_coordinate_system ()
create_subset_window_by_ids (ids)

D.1.2 Useful Functions

Examples of creating and using datasets are given in Section 23.1. Two functions that support creating datasets from an flt storage and a tab delimited storage, respectively, are implemented in the module opus_core.misc:

get_dataset_from_flt_storage (dataset_name, directory,
    package_order=['opus_core'], dataset_args=None)
get_dataset_from_tab_storage (dataset_name, directory,
    package_order=['opus_core'], dataset_args=None)

If the dataset is defined in a package (for example, dataset ‘gridcell’ is defined in the urbansim package), put the package name into the package_order list. Otherwise, put all arguments that the Dataset’s constructor needs into dataset_args in form of a dictionary. Both functions return a dataset object.
GNU Free Documentation License

Version 1.2, November 2002


51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow
the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.
H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE
The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

**ADDENDUM: How to use this License for your documents**

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.
Glossary

This section defines important terms used in describing the Opus system, from a modeler’s viewpoint. In addition, there are a few technology-oriented terms that might otherwise be confused with these domain-specific terms.

**Agent** An object, such as a job or a household, that takes direct action on its environment, such as making choices among a set of alternatives. Typically, agents represent “real-world” objects from the model domain. Agents have state. Agents can engage in behavior, communicate, change state, etc. An agent’s behavior usually depends upon its context.

**Agent set** An object that contains a set of agents.

**Alternative** One of the objects that an agent may choose.

**Attribute** A value that can be attributed to a particular object. An inherent characteristic of an object. For instance, the UrbanSim job object may have attributes such as “id”, “gridcell_id”, and “is_in_scalable_sector_group”. Attributes may be “primary”, in that the jobs in the input database have values for that attribute. Or attributes may be computed by a variable definition. There are no “constant” attributes. Models may modify any attributes, including “primary” attributes.

**baseyear cache** The file-based storage for the attribute values read from the baseyear database. Access to a file-based cache is much faster than to a database. Data cached during model simulation is written to a simulation cache.

**Characteristic** Synonym for attribute.

**Choice** We avoid using this term because it is ambiguous: it can mean either a member of a choice set, or it can mean the single chosen alternative.

**Choice set** An object that contains a set of alternatives. By convention, we use this term instead of “alternative set”.

**Category** A single state of a nominal variable, such as a development type. Often used to group objects based upon their characteristics.

**Category set** A set of categories.

**Dataset** A set of objects, such as gridcells or jobs, of the same type. You can think of a dataset as a table, with one column for each attribute, and one row for each object. Each object in the dataset has a unique identifier (an integer) stored in an attribute whose name is referenced as “id_name”.

In practice, a dataset often is constructed from a single table read from a Storage object, though some datasets are constructed by joining information from multiple tables.
Logit Equation  Also known as a Logit Transformation, this transforms the collection of an agent’s computed direct utilities for all alternatives in the choice set into an estimated probability for each alternative:

\[ P_{ij} = \frac{e^{\lambda U_{ij}}}{\sum_{j' \in J} e^{\lambda U_{ij'}}}, \]

Estimated Utility Function  A set of paired variables and coefficients of the form \( \sum c_i * u_i \).

Graphical User Interface  A general term for an interactive, graphical user interface to a computer application. The Opus/UrbanSim GUI provides access to much of the system’s functionality without needing to use Python scripts.

Integrated Development Environment (IDE)  An application to assist software engineers in writing programs. An IDE might include support for code browsing and searching, debugging, and so forth. The Opus/UrbanSim developers usually use either the Eclipse or Wing.

Interaction Set  A dataset of variables describing the interaction between two different datasets. See “Interaction Variable”.

Interaction Variable  A variable that computes the interaction between two different datasets, such as that gridcell “number_of_households” variable that computes the number of households residing in each gridcell.

Logit Model  A Logit model is defined using an indirect utility equation, which includes the error term and specifies how that error is distributed. For example:

\[ U_{ij} = \theta X_{ij} + \epsilon_{ij} = \sum_{k \in K} \theta_k X^k_{ij} + \epsilon_{ij}, \]

where \( \epsilon_{ij} \) is i.i.d. distributed Gumbel Type I.

Model  An object that defines behavior and that has a run() method. Models have no state. They get their data from data objects, and store their results in data objects. A model can be a formula that calculates on the data provided to it. A model can be a sequence of actions to perform. A model also can act as an action. A model must have a run() method.

By convention, every model module contains a set of tests that test that model.

Model implementation  A model whose specification, or both specification and coefficients, have been estimated to a particular data set.

Model object  A model, or a model part. Model objects act on data objects.

Model component  An object that performs a logically distinct task that is part of how the model runs. Every model is formed by a sequence, and perhaps cycle, of model steps. Different models may combine the same model components in different ways to define different models.

Note that models can become components in other higher-level models, such as a land-price model and a neighborhood choice model that are parts of a household-location-choice model, which is part of an UrbanSim model.

Model specification  A definition of what variables are included in an econometric model. Coefficient values may, or may not, also be part of the specification.

Opus  The Open Platform for Urban Simulation, a new Python-based framework for writing urban and regional models.

Opus package  The term “Opus package” refers to any Python package built using the Opus framework and that follows the Opus package guidelines. The packages in the Opus base distribution will all be Opus packages in this sense, as will contributed Opus packages. See Section 26.1 for directions for creating your own Opus package.
Puget Sound Regional Council (PSRC)  www.psrc.org The Metropolitan Planning Organization for the Central Puget Sound Region in Washington State, which includes Seattle, Bellevue, Tacoma, and other cities, towns, and unincorporated areas. PSRC is one of the users of UrbanSim.

**Python package**  A way of structuring Python’s module namespace by using “dotted module names”. See the Python documentation at http://www.python.org/doc.

**Resources**  A Python dictionary object. It is used to contain parameters passed between model steps.

**Simulation cache**  The file-based storage for the attribute values for the datasets used by the models. Access to a file-based cache is much faster than to a database. The baseyear attributes are stored in the baseyear cache.

**Simulation cycle**  A set of sub-models that run in sequence before repeating. Typically, this refers to the outermost loop.

**Submodel**  A . . . need to supply this definition

**Time step**  The unit of simulated time between each simulation cycle. For many simulations, this will be a year.

**Variable**  An attribute that is computed by a variable definition. Each variable definition resides in its own Python module. The Python module for a variable, e.g. “is_industrial”, is the same as the variable name except with a “.py” extension, e.g. “is_industrial.py”.

**Work request system**  A system for filing and tracking work requests, including bug reports.
INDEX

‘.project’ file, 264
adding new tool set, 35
Adobe, 275
agent
definition, 304
Agent Location Choice Model, 242
Agent Location Choice Model Member, 244
Agent Relocation model, 260
agent set
definition, 304
AgentLocationChoiceModel class, 242, 244
AgentLocationChoiceModelMember class, 244, 246
AgentRelocationModel class, 260
aggregate variable, 80, 180
‘alias.py’ file, 178, 229
aliasing, 83, 178
AllocationModel class, 217
alternative
definition, 304
ArcGIS, 57
arguments to variables, 79
attributes, 156, 171, 196
computed, 171, 196
definition, 304
primary, 171, 196, 304
base_year_data, 31
baseyear cache, 146, 306
definition, 304
BHHH estimation, 221
‘build.xml’ file, 264
Building Location Choice Model, 246
Building Transition Model, 246, 256
BuildingLocationChoiceModel class, 246
BuildingTransitionModel class, 256
casting, 79
category
definition, 304
category set
definition, 304
characteristics, 156, 304
definition, 304
child project, 23
choice
definition, 304
choice set
definition, 304
ChoiceModel class, 211, 237
Choices class, 220, 262
ChunkModel class, 210
ChunkSpecification class, 227
class
Choices
first_agent_first_choices, 262
lottery_choices, 262
random_choices, 220
random_choices_from_index, 220, 262
EstimationProcedure
bhhh_mnl_estimation, 221
bhhh_mnl_estimation_with_diagnose, 221
bhhh_wesml_mnl_estimation, 221
bhhh_wesml_mnl_estimation_with_diagnose, 221
estimate_linear_regression, 221
estimate_linear_regression_r, 221
Probabilities
employment_relocation_probabilities, 262
household_relocation_probabilities, 261
mnl_probabilities, 220
Regression
linear_regression, 220
linear_regression_with_normal_error, 220
Sampler
stratified_sampling, 222
weighted_sampling, 222
upc_sequence, 220
clip_to_zero function, 79
coefficient values, 305
Coefficients class, 224
coefficients, 212, 305
compute_variables method, 172
computed attributes, 171, 196
Configuration class, 227
controlling the simulation, 49
Corrected Land Price Model, 232
CorrectedLandPriceModel class, 232
CorrectLandValue class, 233
correlation, 159
creating a Regression model, 38, 39
creating an Allocation Model, 37, 39
cross-platform, 21
CruiseControl, 264
Data Manager tab, 31
Data tab, 21
Database Server Connections, 29, 58
Dataset class, 196
dataset
definition, 304
dataset pool, 173, 181, 183, 226
dataset_name argument, 196
DatasetPool class, 226
DatasetSubset class, 199, 211
DBF file, 57
Demographic Transition Model, 97
dependencies, 205
development constraints, 105, 134, 140
Development Event Transition Model, 254
Development Project Location Choice Model, 103, 239, 246
Development Project Transition Model, 252, 256
development templates, 106, 138
DevelopmentEventTransitionModel class, 254
DevelopmentProjectLocationChoiceModel class, 239
DevelopmentProjectLocationChoiceModelCreator class, 240
DevelopmentProjectTransitionModel class, 252
disaggregate variable, 80, 180
Distribute Unplaced Jobs Model, 259
DistributeUnplacedJobsModel class, 259
Eclipse IDE, 264, 305
Economic Transition Model, 95
Employment Location Choice Model, 99, 244, 261
Employment Relocation Model, 98, 246, 259, 261
Employment Transition Model, 250, 261
employment_relocation_probabilities, 262
EmploymentLocationChoiceModel class, 244
EmploymentRelocationModelCreator class, 261
EmploymentTransitionModel class, 250
EquationSpecification class, 223
Estimated Utility Function
definition, 305
estimation, 221
BHHH estimation, 221
EstimationProcedure class, 221
Events Coordinator, 255, 257
EventsCoordinator class, 257
EventsCoordinatorAndStoring class, 258
exp function, 79
exporting Opus dataset, 32
expressions, 25, 77, 198
equality, 83
implementation, 207
principal dataset, 82
first_agent_first_choices, 262
flat-file database, 29
font preferences, 28
General tab, 21, 30
GeneralResources class, 227
Graphical User Interface (GUI)
Data Manager tab, 31
Data tab, 21
definition, 305
General tab, 21, 30
Model Manager tab, 37
Models tab, 21
Opus Data tab, 31
Results Manager tab, 49
Results tab, 21, 53
Scenarios tab, 21, 48
Tools tab, 34
GUI, see Graphical User Interface
hidden identifier, 196
histogram, 158, 166, 198
household agents, 156
Household Location Choice Model, 100, 242, 261
Household Relocation Model, 98, 243, 261
Household Transition Model, 247, 261
household_relocation_probabilities, 261
HouseholdLocationChoiceModel class, 242
HouseholdRelocationModelCreator class, 261
HouseholdTransitionModel class, 247
Housing Price Model, 234
HousingPriceModel class, 234
id_name argument, 196
IDE, see Integrated Development Environment
in_storage argument, 196
in_table_name argument, 197
indicator, 25
available indicators, 56
export results, 55
indicators in current visualization, 56
view indicator automatically, 55
indicator batch, 55
Indicator Framework, 21
indicator map, 25
indicator visualizations, 55
adding an indicator visualization, 55
collection, 56
dataset, 56
running an indicator batch, 55
visualization name, 56
visualization types, 56
map format options, 57
table format options, 70
inheritance, 30
installing Opus, 281
Integrated Development Environment (IDE)
definition, 305
Eclipse, 264, 305
Wing, 156, 305
interaction sets, 80, 179
definition, 305
interaction variable
definition, 305
InteractionDataset class, 199
JoinAttributeModificationModel class, 218
Land Price Model, 229
LandPriceModel class, 229, 232
latex2html, 275, 277
lazy evaluation, 198
linear_utilities, 219
linear_utilities_diagnose, 219
ln function, 79
ln_bounded function, 79
ln_shifted function, 79
ln_shifted_auto function, 79
Location Choice Model, 237
LocationChoiceModel class, 237, 239, 242
Logit Equation
definition, 305
Logit Model
definition, 305
lottery_choices, 262
Mapnik, 57
custom graduated color map example, 65
custom map example, 63
default map example, 62
diverging custom graduated color map example, 67
setting options, 58
True/False attribute map example, 68
matplotlib, 158, 198
Memory
Flushing attributes, 197
Lazy loading of attributes, 197
Loading a subset of a dataset, 175
Using separate process, 148
Memory management
datasets_to_cache_after_each_model, 151
flush_variables, 152
Flushing attributes, 151
tables_to_cache_nchunks, 150
Using multiple chunks per model, 186, 197, 210
When caching data from input store, 150
menu bar, 28
Model class, 210
model
coefficients, 224
component, 305
definition, 305
implementation, 305
Logit, 305
object, 305
opus_core model components, 218
specification, 223, 305
urbansim model components, 261
model group, 151–154, 226, 244, 246
model group member, 227
Model Manager tab, 37
ModelGroup class, 226
ModelGroupMember class, 227, 244, 258
models
opus_core models, 210
allocation model, 85, 217
choice model, 87, 211
regression model, 86, 214
simple model, 84, 217
urbansim models, 229
agent location choice model, 242
agent relocation model, 260
building location choice model, 246
building transition models, 256
development event transition model, 254
development project location choice model, 103, 239
development project transition model, 252
distribute unplaced jobs model, 259
employment location choice model, 99, 244
employment relocation model, 261
employment transition model, 95, 250
events coordinator, 257
household location choice model, 100, 242
household relocation model, 261
household transition model, 97, 247
housing price model, 234
land price model, 229
Index 311
SimpleModel class, 217
simulation
controlling a simulation, 49
running a simulation, 48
simulation cache, 112, 148
definition, 306
simulation cycle
definition, 306
Specification, 223
SpecifiedCoefficients class, 225
SQL database, 57
sqrt function, 79
starting a simulation, 147
Stochastic unit tests, 267
Storage, 196, 197
dict, 170, 181
tab, 157
Storage class, 200
stratified sampling, 222
subexpressions, 27
submodel
definition, 306
time step
definition, 306
Tool library, 34
Tool sets, 35
Tools menu, 28
Result Browser, 54
export results of indicator, 55
view indicator automatically, 55
Tools tab, 34
troubleshooting
urbansim models
  Employment Transition Model, 252
  Household Transition Model, 249
tutorial
urbansim, 156
Unit tests
stochastic, 267
Utilities class, 219
utilities, 219
Variable class
compute() method, 205
creating variables, 205
dependencies() method, 205
variable library, 25
variable names
arguments in, 79
variable values
aggregation, 80, 180
disaggregation, 80, 180
VariableFactory class, 205
get_variable() method, 205
variables, 25, 171
definition, 306
version numbers, 284
viewing datasets, 32
weighted sampling, 222
Wing IDE, 156, 305
work request system
definition, 306
XML configuration files, 22

Index 313