Earth Surf. Dynam., 5, 21–46, 2017
www.earth-surf-dynam.net/5/21/2017/
doi:10.5194/esurf-5-21-2017

Earth Surface Dynamics Open Access EGU

Creative computing with Landlab: an open-source toolkit for building, coupling, and exploring two-dimensional numerical models of Earth-surface dynamics

Daniel E. J. Hobley[1,2,3], Jordan M. Adams[4], Sai Siddhartha Nudurupati[5], Eric W. H. Hutton[6], Nicole M. Gasparini[4], Erkan Istanbulluoglu[5], and Gregory E. Tucker[1,2]

[1]Cooperative Institute for Research in Environmental Sciences (CIRES), University of Colorado, Boulder, USA
[2]Department of Geological Sciences, University of Colorado, Boulder, USA
[3]School of Earth and Ocean Sciences, Cardiff University, Cardiff, UK
[4]Department of Earth and Environmental Sciences, Tulane University, New Orleans, USA
[5]Department of Civil and Environmental Engineering, University of Washington, Seattle, USA
[6]Community Surface Dynamics Modeling System (CSDMS), University of Colorado, Boulder, USA

Correspondence to: Daniel E. J. Hobley (hobleyd@cardiff.ac.uk)

Received: 20 August 2016 – Published in Earth Surf. Dynam. Discuss.: 14 September 2016
Revised: 24 November 2016 – Accepted: 14 December 2016 – Published: 16 January 2017

Abstract. The ability to model surface processes and to couple them to both subsurface and atmospheric regimes has proven invaluable to research in the Earth and planetary sciences. However, creating a new model typically demands a very large investment of time, and modifying an existing model to address a new problem typically means the new work is constrained to its detriment by model adaptations for a different problem. Landlab is an open-source software framework explicitly designed to accelerate the development of new process models by providing (1) a set of tools and existing grid structures – including both regular and irregular grids – to make it faster and easier to develop new process components, or numerical implementations of physical processes; (2) a suite of stable, modular, and interoperable process components that can be combined to create an integrated model; and (3) a set of tools for data input, output, manipulation, and visualization. A set of example models built with these components is also provided. Landlab's structure makes it ideal not only for fully developed modelling applications but also for model prototyping and classroom use. Because of its modular nature, it can also act as a platform for model intercomparison and epistemic uncertainty and sensitivity analyses. Landlab exposes a standardized model interoperability interface, and is able to couple to third-party models and software. Landlab also offers tools to allow the creation of cellular automata, and allows native coupling of such models to more traditional continuous differential equation-based modules. We illustrate the principles of component coupling in Landlab using a model of landform evolution, a cellular ecohydrologic model, and a flood-wave routing model.

1 Introduction and motivation

Across a wide array of fields, researchers use numerical models to study processes that operate on and across the Earth's land surface and shallow subsurface. Science and engineering applications of these models of surface dynamics range from short-term flood forecasting (e.g. Horritt and Bates, 2002) to simulating the evolution of Earth's landscape over geologic epochs (e.g. Tucker and Hancock, 2010). Models may focus on a theoretical understanding of processes and their interaction, on management or engineering applications, or on predicting environmental responses to natural or human-made perturbations. Although the processes and temporal and spatial scales vary widely, the software behind these models is often quite similar. For example, most Earth-surface dynamics models manage data structures and algo-

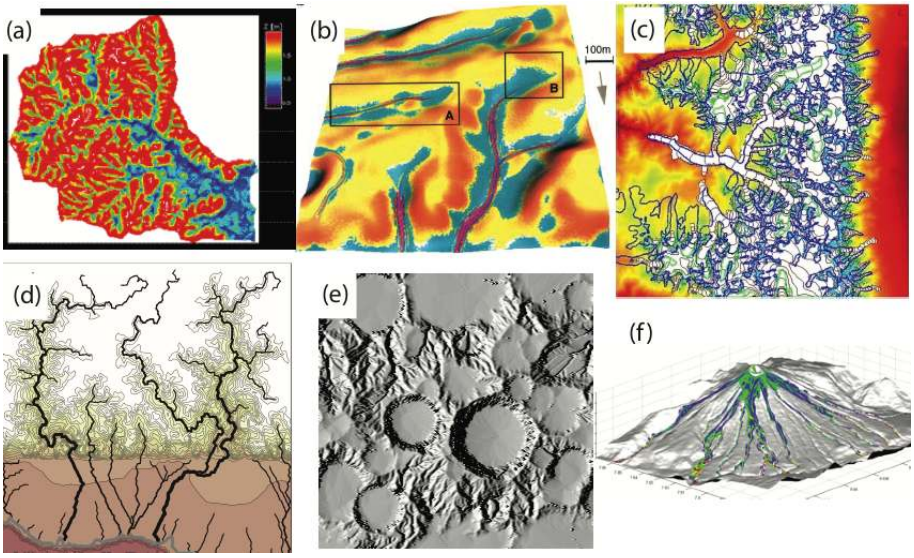


Figure 1. Examples of surface-process models. **(a)** Computed depth-to-groundwater, from the GSEM coupled groundwater–surface water model (Berger, 2000, image courtesy D. Entekhabi). **(b)** Computed patterns of soil erosion and sedimentation on agricultural fields, using the SIMWE soil erosion model (Mitas and Mitasova, 1998). **(c)** Model of ice-age glacier extent over the Sierra Nevada, USA, using the GC2D ice-flow model (Kessler et al., 2006). **(d)** Simulation of canyon erosion and fan-delta progradation in a region of active uplift (top) and subsidence (bottom), using the CHILD landscape evolution model (Tucker and Hancock, 2010). **(e)** Model of simultaneous cratering and fluvial erosion on the ancient Mars surface, with the MARSSIM model (Howard, 2007). **(f)** Simulation of pyroclastic flows at Tungurahua volcano, Ecuador, using the VolcFlow model (Kelfoun et al., 2009).

rithms to represent a terrain surface and its connectivity, and many include solution algorithms to compute flows of mass (such as ice, liquid water, sediment, or chemical nutrients) across terrain (Slingerland and Kump, 2011) (Fig. 1).

However, scientists who want to use an Earth-surface model often build their own unique model from the ground up, re-coding the basic building blocks of their model rather than taking advantage of codes that have already been written (Adams et al., 2014; Katz et al., 2015; Overeem et al., 2013). This undoubtedly does produce novel software capable of fulfilling its designer's needs, and can have advantages in helping the programmer to acquire a total understanding of the code base, but this approach also has many associated problems: many person hours are lost rewriting existing code, and the resulting software is often idiosyncratic, ad hoc, undocumented, and unable to interact with other software programs both in the same scientific community and beyond. In particular, models are often initially written to solve a very specific problem, rather than to provide a flexible and reliable platform for solving a general class of problems (Easterbrook, 2014). It may also become impossible for a single programmer to maintain their grasp of their code base once it exceeds a certain size. A result is that software development often acts as a bottleneck to progress, with frequent duplication of effort as research groups struggle to adapt existing software or develop new code from the ground up as each new research problem emerges.

The Landlab modelling framework described here seeks to mitigate these redundancies and lost opportunities and simultaneously lower the bar for entry into numerical modelling. The approach is to create a user- and developer-friendly modelling environment that provides scientists with the fundamental building blocks needed for modelling surface dynamics on the Earth, and potentially beyond. The framework takes advantage of the fact that nearly all surface-dynamics models share a set of common software elements, despite the wide range of processes and scales that they encompass (Peckham et al., 2013; Slingerland and Kump, 2011). Providing these elements in the context of the popular scientific programming language Python, and with strong user support and community engagement, would contribute to accelerating progress in the diverse sciences of the Earth's surface.

From the user's perspective, Landlab enables the following:

1. Rapid, easy creation of a number of distinct geometric *grids*, with all the connectivity between various elements already defined, and the ability to create two-dimensional data fields across a given grid.

2. Functions to operate on the values defined on such a grid, enabling the solution of time-dependent numerical algorithms across them (e.g. differential equations, cellular automata).

3. A mechanism for the control of boundary conditions across a grid;

4. Encapsulation of conceptual models for individual Earth-surface processes into reusable *components*, with a standard interface that allows operation across Landlab grids.

5. The ability to build a multi-process model by combining together components.

6. The ability to quickly and efficiently build new components, and to couple them with those components already in the library.

7. A straightforward and standardized input and output interface, including the ability to import from and export to common spatially distributed data formats such as NetCDF and ESRI ASCII, as well as a plotting module. This interface also enables coupling to third-party models and software.

2 Approach

2.1 Guiding design principles

The design principles for Landlab have been guided both by our observations of current software design practices in the surface-system modelling community and by white papers issued by existing organizations both within this community (Adams et al., 2014; Overeem et al., 2013; Peckham et al., 2013) and in the scientific software design community more widely (Becker et al., 2015; Chue Hong, 2014; Katz et al., 2015; NSF, 2012). Our key observations are as follows:

1. Many models exist that simulate Earth-surface processes, and many of these share a very similar underpinning in terms of the basics of grid construction and the suite of simulated processes. This set of models represents significant past duplicative effort in the surface process modelling community. Although the reasons for duplication are likely multiple and vary from group to group, we note that we are unaware of previous efforts to advertise a flexible, open-source programming framework.

2. Orphaned or unmaintained codes are common in the research community, having been built for a single purpose and then set aside.

3. Although standardized frameworks for model interoperability are now in place (such as the framework designed and maintained by the Community Surface Dynamics Modelling System, CSDMS, group; Hutton et al., 2014; Overeem et al., 2013; Peckham et al., 2013), many models are not compatible with these standards. We hypothesize this is largely due to the effort required by the original programmer to modify legacy code – which in many cases was written before the standards were established – to meet these new interoperability criteria.

4. Existing model software tends to have a high bar to entry. Many models are written in compiled languages, such as Fortran, C, and C++ (examples from the geomorphology and sedimentary stratigraphy communities include CHILD: Tucker et al., 2001b; Sedflux: Hutton and Syvitski, 2008; MARSSIM: Howard, 2007; Fastscape: Braun and Willett, 2013; DAC: Goren et al., 2014; SIBERIA: Willgoose et al., 1991a, b). This requires the prospective user be fluent in these languages before the code can be modified or, in many cases, even used efficiently. Because many legacy codes were not designed to be shared amongst the community, documentation, both in-line and external, tends to be idiosyncratic at best and missing at worst.

5. In several instances, scientific software with a broad user base exists but remains closed source. This includes both tools for data analysis (e.g. ArcMap, Matlab) and in some cases the modelling software itself (e.g. FLAC; Itasca, 2000; Dionisos, Granjeon and Joseph, 1999). Where software has to be purchased, this presents obvious barriers to wide uptake of modelling approaches using these tools in terms of financial cost for the user. More importantly, all closed-source software also presents significant barriers to code assessment in peer review and to reproducibility of the work (Crick et al., 2014; Katz et al., 2015).

These observations lead us to a set of key design principles that have governed our development of Landlab:

a. Landlab should be a community resource, and thus fully **open source**.

b. Landlab should provide a development environment that is **flexible, extensible, and highly reusable**.

c. Landlab should be written in a language that allows **rapid development** of new code.

d. Landlab should be fully compliant with the CSDMS model interoperability standards (Peckham et al., 2013) from the ground up, and this compliance should be built into the low-level development framework itself. Thus, for example, components written in Landlab will be automatically compliant with these standards.

e. Landlab should have a **low bar to entry** and be thoroughly **documented**. Tutorials should be present. It should be possible for a beginner to use Landlab without a full grasp of the underlying model architecture, in a "plug and play" fashion.

f. Landlab's code needs to be **sustainable**, as detailed below.

2.2 Low-level design choices

In turn, these guiding design principles directed early decisions in terms of Landlab's coding language, architecture, and distribution.

2.2.1 Open-source availability

Landlab is licensed under the MIT free software license, an approved license of the Open Source Initiative. This license allows a user to deal in the software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software. The source code and associated files are maintained in a Git version-control repository, for which the master repository is presently hosted on the GitHub website, https://github.com/landlab/landlab. Release versions are also freely available through the *pip* and *conda* Python package managers. The model repository maintained by CSDMS offers links to Landlab documentation and to the GitHub repository, increasing Landlab's visibility to the surface process modelling community in particular. Web-based documentation is hosted at http://landlab.github.io. This includes both developer-written summary documents and tutorials, as well as reference-level documentation that is automatically generated from inline comments and examples in the code itself.

2.2.2 Programming language

Landlab is written in Python and exploits and includes as dependencies a number of widely used scientific Python packages: numpy, scipy, matplotlib, nose, netCDF4, numpydoc, cython, six, pyyaml, setuptools, and libgcc. The decision to write in Python was explicitly made to lower the bar for entry to Landlab, to increase the flexibility and reusability of the code base, and to increase development speed both for the core development team and for future users. Informal canvassing amongst the surface process community, especially amongst graduate students and other early-career scientists less likely to already be strongly wedded to a certain development environment, revealed a marked preference for – and greater familiarity with – Python over C++ (other open-source languages were rarely mentioned). This changing preference for Python has also been noted for PhD students in general, beyond just the field of surface process modelling (Chue Hong, 2014). The choice of Python also means that developers using Landlab can take advantage of that language's affinity for rapid development (Prechelt, 2000). In particular, Python's dynamic typing and interpreted rather than compiled implementation remove the developer's need to deal explicitly with memory management (van Rossum and Drake, 2001). Other advantages of this choice include high portability between platforms, open-source language, numerous existing scientific libraries, and support for selective optimization of time-critical parts of the code base using Cython and/or compiled-language extensions. Cython is a compiled language that is a super-set of Python, and Cython extension modules interact seamlessly with pure Python. However, program modules written in Cython allow more granular control of memory management than is the case in pure Python, which can result in significant acceleration of code. Cython is already in use within Landlab for sections of the code that require long out-of-sequence iterations through arrays, and other sections where pure Python would tend to have poor performance. For example, Cython is used in the construction of some of the grid element connectivity arrays, in the FlowRouter and FastscapeEroder components, and in the CellLab extension to Landlab (Tucker et al., 2016).

2.2.3 Code sustainability

A key objective for Landlab from inception has been that the code base be sustainable (Adams et al., 2014; Becker et al., 2015; Katz et al., 2015; Stewart et al., 2010). Following other authors, we view sustainable software as that which is able to continue effectively, sustaining or improving its functionality through time while at the same time adding new users. Stewart et al. (2010) drew attention to a number of key features of sustainable software, which we have sought to implement:

- *Strong, consistent leadership.* The authors of this paper represent the core development team of Landlab.

- *Rapid prototyping and evolutionary design*. Landlab was initially developed to fill the immediate research needs of the core development team, giving it a strong and well-defined initial direction. In this initial development phase, we have emphasized long-term mountain belt evolution modelling; steady- and nonsteady-flow routing; eco-, surface, and shallow subsurface hydrology; hillslope dynamics; cellular automaton modelling; vegetation dynamics; and ecosystem dynamics. However, the explicitly modular nature of Landlab means that it can readily adapt to new scientific objectives and expand to meet new and as yet unforeseen demands in the future.

- *Modern and effective software engineering practices*. Landlab takes advantage of a number of best practice processes, including extensive and automated unit testing of key code functionality, a formal bug- and issue-tracking record implemented through GitHub, cross-team review of code changes before they are merged into the master branch, and thorough code documentation. A significant portion of our online documentation is created semi-automatically from inline code comments. This reduces duplication of information and aids maintenance and updating of the documentation as the code changes. Individual functions and classes are documented automatically using Python's docstring functionality. General descriptive documentation and

tutorials are created and maintained manually. Auto-generated documentation is updated and posted to the project website automatically as new code changes are committed to the GitHub repository using "webhook" functionality provided through the http://readthedocs.org website.

– *Sustained compatibility with underlying libraries, protocols ,and operating systems*. Landlab is compatible both with Python 2 and 3. The code base is tested automatically using Travis (Mac, Unix) and Appveyor (PC) continuous integration platforms, across Python versions 2.7, 3.4, and 3.5 (see also Sect. 4).

– *Dissemination and community understanding*. We have sought to publicize Landlab widely at a number of international conferences and workshops, classes, and through collaborative networks. We estimate that, as of mid-2016, approximately 330 potential users have now seen or participated in Landlab-based presentations or classes.

– *Encouraging collaborative software development*. Landlab enables users to tailor its functionality to their specific needs, through its modular design and flexible grid and grid functions. We are already aware of a number of groups outside the core Landlab development team working with Landlab for their own research purposes.

A secondary aspect to sustainability is the ability to have the software continue to be useable after the active development cycle has ceased (Stewart et al., 2010). We anticipate that the choice of Python, minimal system and extension package requirements, open-source availability of our code base, and thorough documentation will sustain our code for the foreseeable future.

3 Model architecture

Landlab has an essentially tripartite structure – a core grid module, a library of process components, and a set of supporting utilities (Fig. 2). The various subdivisions of the code behave as Python modules and can be imported and used within a Python environment independently.

3.1 Landlab's gridding engine

Landlab provides the ability to create a two-dimensional simulation grid of a user-specified size and shape, with a single line of code. Grids are represented as Python objects; a grid object includes data describing its geometry and topology, as well as a variety of methods and functions to manage data and perform common numerical operations. (In object-oriented programming parlance, a *method* is a procedure associated with an object; in this case, "method" means a function that is defined within the grid class, and that can be accessed with the "grid.method()" syntax typical of other class properties.)

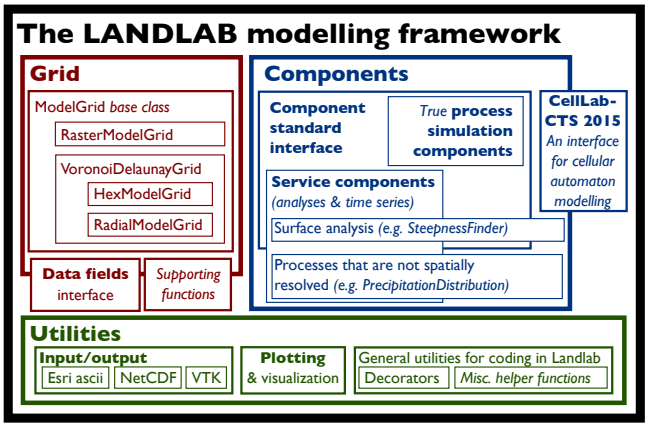


Figure 2. Schematic illustration of the structure of Landlab 1.0. The three main divisions of the code are the grid, the components, and supporting utilities. Structure within these three main divisions is discussed in the main text.

Although Landlab grids are inherently two-dimensional, in many cases it is nonetheless possible to create an effectively one-dimensional simulation by creating a 3-by-N regular grid and closing the nodes along the top and bottom edges (see Sect. 3.1.4). Three-dimensional grids are not possible in Landlab at this time, though they may be supported in a future release.

3.1.1 Grid types and elements

A Landlab grid is defined by a set of grid primitive elements: nodes, links, cells, corners, faces, and patches (Fig. 3). In terms of graph theory, these can be thought of as two interlocking and offset sets of points (nodes vs. corners), edges (links vs. faces), and areas (patches vs. cells). The entire grid can be generated from a description of the geometry of only one of these element types – typically, a user might specify the locations of the nodes, and the grid object's remaining elements are automatically placed according to this node framework.

Each element type shares unique one-to-one or one-to-many geometric mappings with the other elements. Were the grids to be infinite, these mappings would be perfectly reciprocal – the topology and connectivity of each element with respect to every other element would be identical everywhere it occurs. However, because these grids are finite, we must arbitrarily decide whether the bounding elements are the set of nodes, links, and patches or the set of corners, faces, and cells. We have chosen the former (see Figs. 4, 5), which means that for example, while all cells have nodes, not all nodes have cells – as the nodes at the grid perimeter cannot have cells defined around them. Table 1a lists the unique

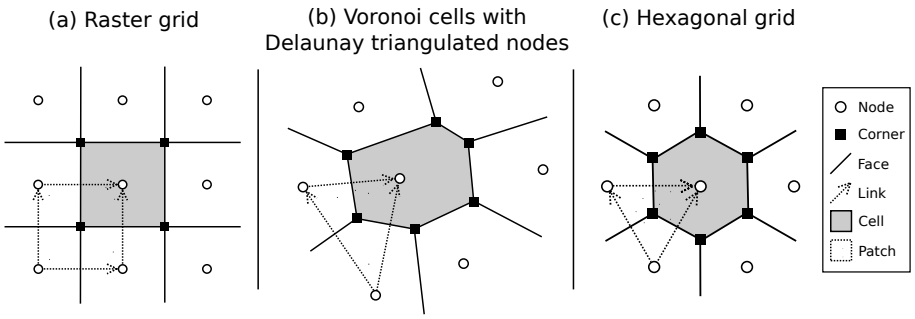


Figure 3. Geometry and topology of grid elements on various Landlab grids. Only one patch and its bounding links are shown for each example to prevent the diagram from becoming cluttered. Links always point into the upper right semicircle, as described in the text.

Table 1. **(a)** One-to-one mappings of Landlab grid elements. **(b)** Primary one-to-many mappings of Landlab grid elements.

(a) Element 1	Element 2	Behaviour at grid perimeter		
Node	Cell	Perimeter nodes lack cells		
Link	Face	Perimeter links lack faces		
Patch	Corner	Neither element defines the perimeter		
(b) Element	Connected elements	Number of each connected element by grid type:		
		Raster	Voronoi–Delaunay	Hexagonal
Node	Link, patch	1 : 4	Variable	1 : 6
Link	Node, patch	1 : 2	1 : 2	1 : 2
Patch	Node, link	1 : 4	1 : 3	1 : 3
Cell	Face, corner	1 : 4	Variable	1 : 6
Face	Cell, corner	1 : 2	1 : 2	1 : 2
Corner	Face, cell	1 : 4	1 : 3	1 : 3

one-to-one mappings of features, and emphasizes which element defines the grid edge in each case. Table 1b lists the primary one-to-many relationships defined for each element type, and lists the standard number of mapped elements (if well defined) for each of the primary grid classes. Note that this table only lists the most useful identities within the three-element groupings node-link-patch and cell-face-corner. The other identities also exist and can be reconstructed from the one-to-one identities in Table 1a.

Data can be assigned to any element of the grid (see Sect. 3.2, below). The grid classes also provide properties that define and describe the geometric interrelationships amongst these grid elements (see, e.g., Fig. 4). These mappings allow common geometric operations (such as calculation of gradients across the grid, finding maximum/minimum/mean values of neighbours, upwinding schemes, and flux divergences) to be achieved in typically one or two lines of code.

Landlab provides native support for both regular and irregular grids (Figs. 3, 4). Treating both grid types natively within Landlab allows the grid to be tailored to specific applications. For example, raster grids provide compatibility with digital elevation model data, and can in some cases allow better optimized process algorithms. Trigonal grids with hexagonal cells provide an additional axis of symmetry, and obviate the need for handling diagonal connections in certain types of numerical algorithm (such as flow routing; e.g. Jenson and Domingue, 1988). Irregular grids avoid some of the cardinal direction artifacts than can form on regular grids, such as linear networks and linear drainage divides, as well as consequent biases in measured channel metrics like drainage density, river length, and channel slope (Braun and Sambridge, 1997).

Regular grids with quadrilateral cells are implemented as rasters, and irregular grids and all other regular configurations (e.g. hexagons) are implemented as Voronoi–Delaunay interlocked meshes, as also used in the landscape evolution models CASCADE (Braun and Sambridge, 1997) and CHILD (Tucker et al., 2001b). Grid subtypes are defined within these broad families (Table 2). Landlab also implements a base grid class ("ModelGrid") from which both the raster and Voronoi–Delaunay grids are derived. This class describes the elements of the grid and allows their geometries and topologies to be set but defines no rules for how to do this. This base grid class is primarily intended as a frame-

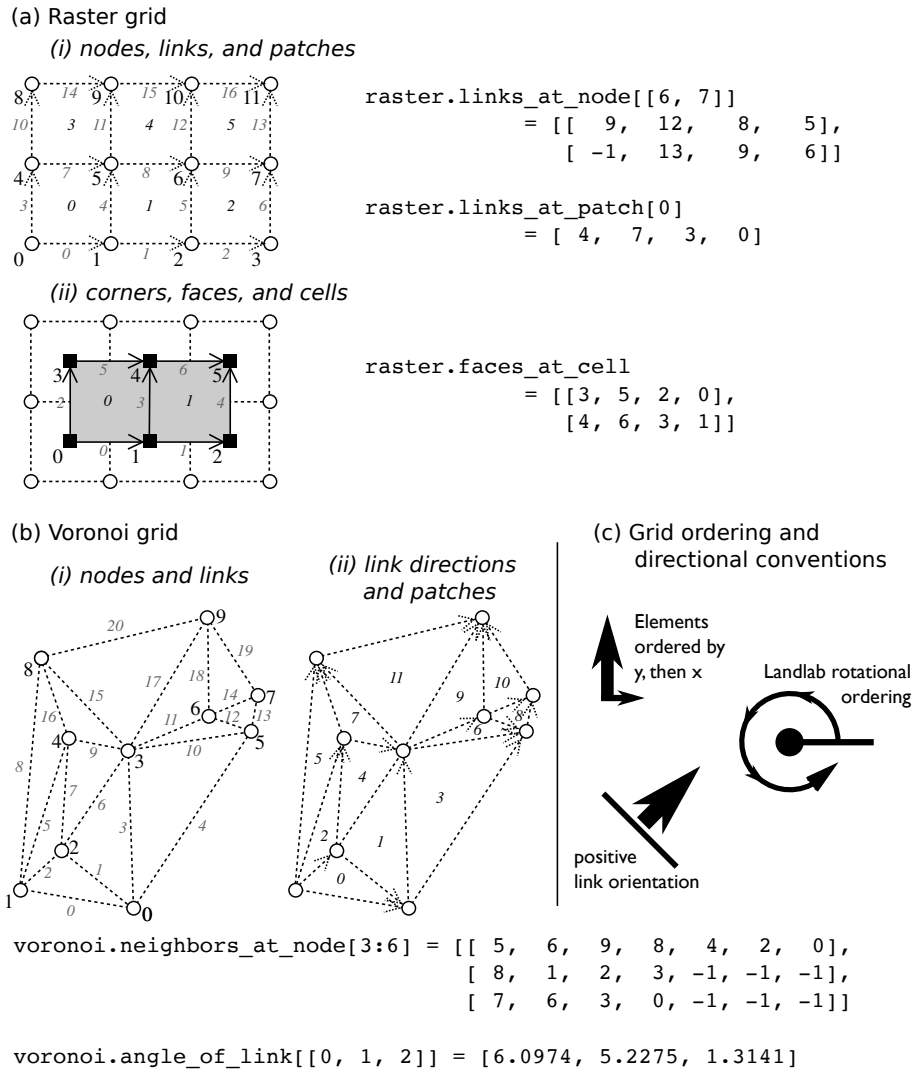


Figure 4. Standard ordering schemes and conventions in Landlab. Examples are shown for both a small RasterModelGrid **(a)** and a small VoronoiDelaunayGrid **(b)**. Point elements (nodes, corners) are numbered in black plain text, areas (patches, cells) in black italics, and linear elements (links, faces) in grey italics. Symbols are as in Fig. 3. In all grid types, elements are ordered by y then x according to their geometric centres. Directional elements (links, faces) always point towards the top right quadrant. Rotational ordering is always anticlockwise from the positive x axis (right/east). This includes angle measurements. Examples of calls to grid properties are shown alongside each grid type to illustrate the expression of these ordering rules in practice. Note that corners, faces, and cells are not shown in panel **(b)** for clarity.

work from which to derive new grid architectures, rather than as a usable grid type in isolation.

Although the grid primitive element set is shared between the various grid types, the implementation of the geometries is slightly different. For example, core nodes in a raster grid will always have exactly four links, whereas they may have any number of links in a Voronoi-centred irregular grid (Table 1b, Fig. 3). Similarly, methods defined for the grid may be polymorphic or overloaded to optimize functionality for each grid type.

3.1.2 Grid standardization and conventions

All Landlab grids share an identical scheme for the numbering of their elements. All elements are numbered from the bottom left of the grid, starting with an ID of 0. All features are ordered first by y coordinate, then by x, taking the midpoint (for linear features such as faces or links) or geometric centre (for areas such as cells or patches) for non-point elements as necessary (Fig. 4).

For rotational ordering, Landlab adopts the mathematical standard convention of *anticlockwise from the positive x axis* (i.e. the right-hand rule). This applies not only to almost all measured angles (unless otherwise explicitly noted)

Table 2. Currently implemented grid types in Landlab.

Grid type	Grid parent	Notes
Base	None	The base class; a grid defining the elements but without any internal geometry or topologic connectivity imposed.
Raster	Base	Regular grid with identical, square or rectangular cells.
Rectilinear	Raster	Regular grid with quasi-rectangular cells whose size can vary across the grid.
D8 raster	Raster	As for raster, but with diagonal connections between nodes.
D8 rectilinear	Rectilinear	As for rectilinear, but with diagonal connections between nodes.
Voronoi–Delaunay	Base	Irregular grid with polygonal cells and triangular patches. Each node has $n \geq 3$ links.
Radial	Voronoi–Delaunay	Irregular grid where nodes form concentric, evenly spaced rings around a central node.
Hex	Voronoi–Delaunay	Regular grid with identical, regular hexagonal cells and equilateral triangle patches. Each core node has exactly six links.

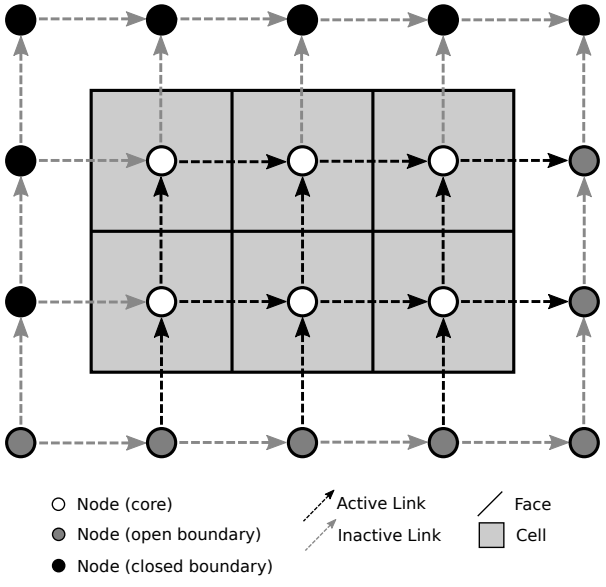


Figure 5. Interplay of node and link boundary conditions on a Landlab example grid. Because nodes rather than corners define the outer margin of the grid structure, the perimeter nodes lack cells, and the perimeter links lack faces (see main text). These aberrant nodes and links are automatically set as boundary elements. Landlab defaults to setting the condition of any such node to FIXED_VALUE_BOUNDARY and any such link to INACTIVE.

but also to the ordering of elements around other elements (such as links around a node), and to the ordering of grid edges where needed (i.e. the standard order is right, top, left, bottom edges). Simple ordering examples are illustrated in Fig. 4.

We extend this same rotational convention to define the directionality of all linear elements (such as links and, where necessary, faces), when such directionality is required. The positive direction is associated with the top-right (first) quadrant; in other words, the positive direction is the one that points more right than down or more up than left. This is shown in more detail in Fig. 4b. This kind of directionality is important for example in the definition of fluxes along links into and out of nodes. In the case of link directions, Landlab provides masking arrays that can describe the local orientation of each link with respect to another feature; for instance, `link_dirs_at_node` describes whether a link points into $(+1)$ or out of (-1) any given node. The use and utility of such data structures is illustrated in Sect. 5.

3.1.3 Mappings and grid characteristics

Landlab uses a standardized grammar to describe the methods and Python properties in the grid classes that provide information about the mapping of grid elements onto other elements, and to obtain information about the grid (e.g. areas, lengths, gradients). The intention of this standardization is to not only make it easier for users to quickly find the method they require but also provide information on the computational efficiency of the operation. Some of this information is summarized in Table 3.

Grid characteristics

Landlab grids provide Python properties to describe the geometric characteristics of the elements themselves, for instance position and dimension. These properties are denoted by the preposition "of", as in, for example, `width_of_face`, `length_of_link`, and

Table 3. Standard grid method and property naming conventions, listed in approximate order of operation speed.

Name contains	Refers to	Operation speed
at	Connectivity of grid elements	Lookup
of	Property of grid or grid element	Lookup (may require allocation on 1st use)
has, is, are	Logical test on grid property	Memory allocation
get, create	Memory allocation of grid property	Memory allocation
set	Update boundary conditions	Calculation; internal consistency checks
map	Map several pieces of data from several elements onto a single element to which they all connect	Several calculations & memory allocations
calc	Perform a calculation using data defined on grid elements	Several calculations & memory allocations

`area_of_cell`. Use of the word *of* tells the user that an array of floats (or, more rarely, integers) denoting a grid characteristic is the expected return. (See for example use of `angle_of_link` in Fig. 4b.) *Of* is also used to access many counted characteristics of the grid as a whole, such as `number_of_nodes`. All these properties return pre-allocated arrays or single values already stored in memory, and can be expected to be fast.

Grid element mappings

The grid also provides numerous Python properties that describe the connectivity and associations of elements with one another. These are denoted by the preposition *at*. Examples include `face_at_link`, `link_at_face`, `links_at_node`, `patches_at_node`, and `node_at_cell`. Use of *at* tells the user that an array of element IDs is the expected return (see Fig. 4 for examples of usage). The Landlab boundary condition interface also uses *at*; for instance, `status_at_node` returns an array containing the boundary condition status (as an integer code) of the grid nodes. All these properties return pre-allocated arrays, and can be expected to be fast.

"has", "is", and "are" methods

Use of *has*, *is*, or *are* in a method name indicates that the method in question applies a logical test to grid properties. These are not simple lookups, as in the case of *at* and *of* properties, but can still be expected to be fairly fast. The returned object will either be a Boolean or an array of Booleans. Examples include `is_boundary`, `are_all_core`, and `has_field`.

"get" and "create" methods

Landlab's design philosophy seeks a balance between speed of access of information about the grid, and memory usage. To this end, only the most commonly used arrays of grid characteristics accessed by *at* and *of* properties are created at grid instantiation. In other cases, these arrays are allocated in memory at the first time of usage in code, then referenced from that point on at subsequent calls of the property. Methods in the grid that begin with *get* or *create* are called by these properties the first time they themselves are used, and construct the necessary arrays in memory. These methods are typically intended for call only by a well-defined subset of other methods internal to grid, and not directly by the user; i.e. in programmer's parlance they are "private". We use the standard Python practice of beginning such methods with a leading underscore in the name, which tells the various Python user interfaces not to report them in standard lists of grid methods.

Computational methods

Landlab provides a large number of grid methods to allow easy completion of common and frequently repeated analyses of the values on the grid. These are denoted by names that begin with *calc*, to denote methods that calculate a new value from provided data, or *map*, which apply some standard rule to map multiple values for connected elements to a single value on the shared element to which they connect. For instance, *calc* methods might allow calculation of gradients at links from data defined at nodes (`calc_grad_at_link`), or flux balances at a node from fluxes defined at incoming and outgoing links (`calc_flux_div_at_node`). *Map* methods might return means of values at links around nodes (`map_mean_of_links_to_node`), or minima of node values attached to each link (`map_min_of_link_nodes_to_link`), or the maximum slope of links leaving each node (`map_downwind_node_link_max_to_node`).
More complex mapping schemes are also available, to allow for instance the mapping of data from topographically upwind or downwind elements only (for example, `map_value_at_upwind_node_link_max_to_node`). All these methods require active calculation and memory allocation of new values.

Boundary condition control

Grid methods that allow user control of boundary conditions use the word "*set*". Boundary condition handling is described further in Sect. 3.1.4, below.

General rules

Words are separated by single underscores. Nouns are typically singular, both describing the element and its characteristic, e.g. `area_of_cell`, not `areas_of_cells`. The exceptions are cases in which more than one thing is associated with each element, such as `links_at_node`, `faces_at_cell`. Any grid property can be expected to be a fast lookup operation if called repeatedly; methods may require additional memory allocation.

3.1.4 Grid boundary condition handling

Also provided are methods to facilitate boundary condition handling (Fig. 5). Nodes can have one of four boundary condition types: *fixed value* (Dirichlet), *fixed gradient* (Neumann), *looped*, or *closed*. A node that is not defined as a boundary is known as a *core* node. The boundary conditions defined on the nodes determine whether each connecting link is *active* (allows flux along it), *fixed* (allows flux, but flux value is fixed) or *inactive* (flux is forbidden), as shown in Table 4a. Each of these boundary conditions is associated with an integer value, which can be seen in the boundary condition arrays `grid.status_at_node` and `grid.status_at_link` (Table 4b).

We should emphasize that this framework is provided for user's convenience; it can be easily ignored if a user wishes to implement a different scheme for boundary condition handling. Further, the appropriate boundary conditions depend on the physical scenario that the user is modelling.

The edges of a Landlab grid are always defined by boundary nodes. Because perimeter nodes lack cells (Sect. 3.1.1), this means not every boundary node necessarily has a cell, and may also not have the standard number of links, patches, etc. (Table 1b). Conversely, any core node can always be expected to have a cell and a standard connectivity as described in that table. Likewise, inactive links at the grid perimeter lack faces, but each active link always intersects, and is uniquely associated with, a single face (Fig. 5). Thus, cells share the boundary conditions of nodes (core vs. boundary) and faces share the boundary conditions of links (active vs. inactive). Note also that nodes that are in the interior of a grid (i.e. not perimeter nodes) can also be assigned as boundary nodes, and that whether or not this occurs depends on the shape of the area that the user is modelling. For example, a user may wish use a grid that represents a drainage basin, with the basin's interior consisting of core nodes, a single node representing the outlet (flagged as a fixed-value or fixed-gradient boundary), and the remainder of the nodes flagged as closed boundaries.

Table 4. **(a)** Link boundary condition status as dictated by node boundary condition status. **(b)** Integer values associated with each boundary condition status.

(a) Nodes at link ends	Link status	Carries flux?
Core – core	Active	Yes
Core – fixed value	Active	Yes
Core – fixed gradient	Fixed	Yes
Core – looped	Active	Yes
Core – closed	Inactive	No
Boundary – boundary	Inactive	No
(b) Element type	Status	Integer value
Node	Core	0
Node	Fixed value	1
Node	Fixed gradient	2
Node	Looped	3
Node	Closed	4
Link	Active	0
Link	Fixed	2
Link	Inactive	4

The grid itself is responsible for keeping track of and ensuring internal consistency between boundary condition properties. The standard numpy setters and getters are overridden for the boundary condition data structures to ensure this internal consistency without the user's involvement. For example, if a user changes a node's status from core to fixed-value boundary, the gridding engine will automatically update the status of the relevant links.

3.2 Spatially distributed data and data fields

A key element of any model of surface processes is a description of how the state variables and surface characteristics vary across the domain. Such data can include both scalar measurements at a point or over an area (such as topographic elevation, water depth, sediment cover fraction, vegetation type) and directional vector data, for instance, describing fluxes across the surface or gradients in scalar values. Landlab uses data constructs called *data fields* within the grid to store and handle this information.

A prominent advantage of the field system is that data may be associated with any of the grid elements: node, cell, link, face, patch, or corner. Data fields are one-dimensional numpy arrays whose length matches the number of elements in question. By indexing these arrays with the IDs of element subsets, the values at specific locations and on each element type can be recovered. This scheme readily allows the storage of both scalar and vector data by exploiting the geometric relationships between the node–link–patch (and cell–face–corner, if desired) groupings, as in a traditional staggered-grid scheme (Harlow and Welch, 1965; Slingerland et al., 1994). Scalar data can be stored at nodes. Because

links describe the connectivity between nodes, vector information describing fluxes or gradients between nodes is readily stored on links; the link's orientation provides an implied unit vector, while the associated value represents the vector's magnitude. There are also a number of use cases in which values can usefully be stored on patches, for instance, in representing resolved means of vector values at the bounding links. This data structure also lends itself to the implementation of some cellular automata. For instance, pairwise transition automata (Narteau et al., 2001, 2009) represent the states of cells on a grid as paired "doublets", with rules prescribed to govern the rates of transition between each doublet type. These are readily implemented in Landlab by mapping the pair states onto the links of a Landlab grid, and representing the corresponding automaton cell states at grid nodes (Tucker et al., 2016).

In terms of implementation in the code, Landlab fields are represented as a dictionary of Python dictionaries within the grid object. The keys to the first dictionary are strings of the names of the grid elements (viz., "node", "link", "patch", "cell", "face", "corner"); the keys to the dictionaries that these return are Landlab *field names*. Users are free to create field names as they wish. However, Landlab maintains a standard format and name list which is widely used by the Landlab component library (Table S1 in the Supplement), and users are strongly encouraged to adopt this scheme to enhance standardization and interoperability throughout the software. Our standard naming scheme echoes that of the community standards adopted by the Community Surface Dynamics Modeling System (CSDMS). Our rationale follows theirs, aiming to remove ambiguity in the identification of different types of numerical information (Peckham, 2014; Peckham et al., 2013). However, given the potential for high frequency of name usage in Landlab code, and our ability to easily assess potential ambiguities between different components, we place more value on name brevity at the expense of total unambiguity as compared with the formal CSDMS Standard Names (https://csdms.colorado.edu/wiki/CSN_Searchable_List). Nonetheless, we maintain one-to-one mappings with the CSDMS Standard Names to enable automated implementation of the CSDMS Basic Model Interface (BMI; see Sect. 3.4.1).

The general format for Landlab names is "thing_described__quantity_described". This approach is more generally known as the object–attribute–value paradigm: the first word or phrase describes the object, the second word or phrase describes the attribute, and the variable's content is its value. A double underscore separates the object from the attribute. An example might be "surface_water__discharge". A full list of names used in Landlab components as of version 1.0 can be found in the Supplement as Table S1. A version of this list up to date with the current release version can be found on the Landlab website.

Units can be attached to grid fields. They are recorded in a further dictionary-like structure, which is a property of the element container. This means they can be accessed with syntax like `grid['node'].units['field__name']`.

Landlab offers some degree of "syntactic sugar" for its field name interface – i.e. the field interface is made more user-friendly by the addition of more readable grid properties to query the fields at each element type, rather than requiring the user to access the both dictionaries directly. For instance, `grid.at_node['my_field_name']` is equivalent to `grid['node']['my_field_name']`. In addition, Landlab also provides convenient shortcuts to create new fields of ones (`grid.add_ones`), zeros (`grid.add_zeros`), and from existing data (`grid.add_field`).

3.3 Components

Components are Python objects that simulate processes within Landlab. A typical Landlab component provides a numerical representation of a single process. For instance, a component might compute the flow of water across a terrain surface using a particular flow law and numerical solution method. Components also exist in Landlab that produce only spatially invariant time series, or that produce time-invariant steady-state solutions across a surface. A prominent example would be the FlowRouter component, which calculates the steady-state accumulation of water discharge and upstream total drainage area through a drainage basin. The latter category also includes a number of analytical tools that produce spatial statistics for a surface; for example, components to calculate the steepness (Wobus et al., 2006) or chi index (Perron and Royden, 2012) for a channel network.

Multiple components can be used together, allowing the simulation of multiple processes acting on a single grid. For example, components simulating hillslope processes and fluvial geomorphic processes can be easily implemented together to create a "custom" landscape evolution model. In some cases, the output from one component may form the input to another, as for example when combining flow routing and sediment transport components, or soil moisture and vegetation growth components. The design of each component is intended to work in a "plug-and-play" fashion, where each component couples simply and quickly to others. This is permitted by a standardized interface for each component, as described in Sect. 3.3.1. Examples of coupled component systems can be seen in Sect. 5.

Landlab provides a suite of existing components that can be deployed by users. Future versions of Landlab will add further components designed by the core development team. However, we anticipate that users of Landlab will also devise new components of their own, allowing the exploration of new processes within Landlab. In keeping with the open-source ethos of the project, we would encourage such users to in turn commit their work back to the master fork of Land-

lab, for the use of others. Documentation and advice for this process can be found on the Landlab website.

3.3.1 Component standard interface

Landlab components have standardized interfaces, which are designed to enhance interoperability both internally to Landlab (between components, or between components and Landlab utilities) and between Landlab and external interfaces like the CSDMS Basic Model Interface (Peckham et al., 2013) (see also Sect. 3.4.1). The Landlab standardized component interface consists of the following:

- An initialization method, with the standard argument signature `__init__(self, grid, x=a, y=b, z=c, ..., **kwds)`, where *grid* is a Landlab grid object; x, y, and z are component-specific keyword arguments with default values a, b, and c; and ***kwds* is an optional keyword argument dictionary. The grid object passed during instantiation is accessed during the running of the component, and its data fields are updated automatically. A component may have any number of component-specific keyword arguments. The variable names of these arguments are not standardized but rather generally unique to each component. The component-specific arguments are, however, required to have default values. The names of the keyword arguments make explicit the data requirements of the component in order to run. However, the ***kwds* argument alternatively allows these parameters to be set from a dictionary of model parameters. In other words, this component could be initialized in two equivalent ways:

```
>>> ld = LinearDiffuser(grid,
linear_diffusivity=1.0,
method='simple')
```

or

```
>>> paramdict = {
'linear_diffusivity': 1.0, 'method':
'simple'}
>>> ld = LinearDiffusivity(grid,
**paramdict)
```

- A run method, with the standard argument signature `run_one_step(dt, *args, **kwds)`, where *dt* is an interval of time over which to execute the component before returning a result, and **args* and ***kwds* are an argument list and dictionary respectively, specific to each component. These latter items allow any additional arguments necessary for the model to run to be passed in. If *dt* is not required for a component to run, it may be omitted.
- A standard set of properties for the component: *name*, *input_var_names*, *output_var_names*, *var_units*, *var_mapping*, and *var_definition*. These properties describe the fields that the component interacts with, the units of each, which element each field is defined on, and a brief summary of what each field represents.

All components inherit from the base class *Component*. This base class enables and regulates the standardized properties and interface that are available for every Landlab component. It also provides methods designed to streamline the creation of the output data fields when a component is instantiated.

Landlab version 1.0 provides a standard component library as part of its installation. A full list of components available in version 1.0 can be found in Table 5. Although these existing components are largely Earth-surface focused, we emphasize that Landlab permits modelling of the evolution of almost any two-dimensional system that lends itself to description by discretized systems of differential equations or cellular automaton rules.

3.3.2 Timestepping and interaction of components

For most existing Landlab components, the component is responsible for controlling its own internal numerical stability. A timestep parameter *dt* is passed to each component that operates in a time-dependent fashion; this timestep can be thought of as the "coupling timescale", and it represents the frequency of interaction between components if more than one is coupled (Fig. 6). However, it is not necessarily the stable timescale, which will vary between components. Each component is responsible for calculating its own stable timestep under the model run conditions, and internally subdividing the imposed *dt* in order to ensure the model run does not become unstable. The user is responsible for selecting an appropriate coupling timescale – too short, and a model run will take more steps than necessary for each component to be stable; too large, and information transfer between the components will be limited, possibly introducing an additional source of numerical error.

Note also that where components employ implicit solutions, there may be no internal limit to the timestep at all (e.g. the Fastscape algorithms of Braun and Willett, 2013, for stream power). In such cases, Landlab will make no check on the imposed timestep, and the user must ensure that the imposed *dt* is appropriate under the boundary and initial conditions that they are running. For instance, the Braun–Willett algorithm ceases to behave in a truly timestep-independent fashion under transient conditions, but in a way that still permits timesteps larger than would be imposed under an explicit Courant condition (for more details see their Appendix B). However, those authors did not propose an alternative scheme to limit the timestep in such cases, and consequently Landlab also does not. A user of this component

Table 5. Components available in Landlab v.1.0.

Component name	Process simulated/analysis performed	Key reference
LinearDiffuser	Linear diffusion of topography	Culling (1963)
PerronNLDiffuse	Nonlinear hillslope diffusion	Perron (2011)
Flexure	Simple lithospheric flexure under loading	Lambeck (1988), Hutton and Syvitski (2008)
gFlex	A more complex flexure model, utilizing gFlex	Wickert (2016)
FlowRouter	A convergent flow router, following the Fastscape algorithms	Braun and Willett (2013)
DepressionFinderAndRouter	A lake filler that can route flow across depressions	Tucker et al. (2001a)
SinkFiller	An algorithm to fill depressions in a surface	Tucker et al. (2001b)
OverlandFlow	A shallow overland flow approximation	de Almeida et al. (2012), Adams et al. (2016)
KinematicWaveRengers	A solution to the depth varying Manning equation for surface flow	Julien et al. (1995), Rengers et al. (2016)
SoilInfiltrationGreenAmpt	Infiltrate surface water into a soil following the Green–Ampt method	Julien et al. (1995), Rengers et al. (2016)
SoilMoisture	Compute local inter-storm water balance and root-zone soil moisture saturation fraction	Laio et al. (2001)
PotentialEvapotranspiration	Calculate potential evapotranspiration across a surface	ASCE-EWRI (2005), Zhou et al. (2013)
Radiation	Calculate total incident shortwave solar radiation	Bras (1990)
Vegetation	Calculate above-ground live and dead biomass, and leaf area index	Istanbulluoglu et al. (2012), Zhou et al. (2013)
VegCA	Cellular automata algorithm to simulate spatial organization of PFTs	Zhou et al. (2013)
PrecipitationDistribution	Generate a storm sequence of intervals and intensities	Eagleson (1978)
FireGenerator	Produces intervals between fire events, following a Weibull distribution	Polakow and Dunne (1999)
StreamPowerEroder	Implements fluvial erosion according to stream power, using the Fastscape algorithms	Braun and Willett (2013)
FastscapeEroder	An alternative implementation of the Fastscape stream power algorithms	Braun and Willett (2013)
DetachmentLtdErosion	An implementation of stream power erosion *not* based on Fastscape	Howard (1994)
SedDepEroder	Sediment-flux-dependent shear stress based fluvial incision	Hobley et al. (2011)
SteepnessFinder	Calculates steepness indices for a channel network	Wobus et al. (2006)
ChiFinder	Calculates the chi index along a channel network	Perron and Royden (2012)

is assumed to have read the component documentation and taken on board that this is potentially an issue, as well as taken steps to check that their output is behaving sensibly and is not highly sensitive to changes in the supplied timestep. We reiterate that it is ultimately the user's responsibility to check that the provided *dt* is appropriate to the modelling scenario at hand.

3.3.3 Parallelization

Together, the componentized nature of Landlab and the level of flexibility afforded to the user conspire to rule out the idea of Landlab as a whole being highly optimized through parallelization. However, there is great potential for parallelization of Landlab at the component level, since the run methods of each component are entirely self-contained. As proof of concept, the Flexure component has already been parallelized (see online code and documentation). Although in Landlab version 1.0 we have not had a compelling enough use case to invest significant time in such work, many of the components already in the library would be amenable to parallelization in this style, and this could be done in future releases.

3.4 Utilities and interfaces

In addition to the grid, which governs the topology and connectivity of spatial data, and the components, which describe how spatial data change with time, Landlab also offers tools that control input and output, including data input and export, translation between widely used data formats, plotting, and the BMI external model interface. Landlab can read and write data files in NetCDF4, VTK, and ESRI ASCII data for-

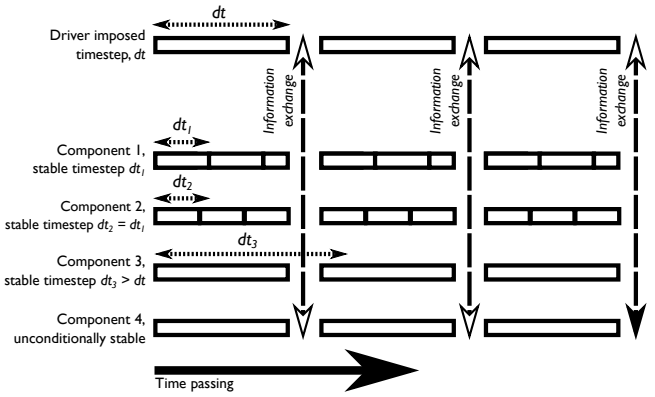


Figure 6. Interaction of timescales between a Landlab driver and a set of components. In this example, a driver that implements components 1–4 has a time loop of length *dt*, and *dt* is the timescale that is passed to the components. Components 1 and 2 implement numerical schemes that have maximum stable timesteps shorter than *dt*. In these cases, the imposed *dt* interval is internally subdivided to ensure the model remains stable. Here, we see two possible ways a component might do this, either always taking the largest timestep possible then a short timestep to finish (component 1) or by dividing the imposed timestep into the minimum number of equal length internal steps, dt_{int}, where $dt_{\mathrm{int}} < dt_{\mathrm{stable}}$ (component 2). Even if a component could run for a timestep longer than *dt* (e.g. components 3 and 4), under an explicit-time Landlab driver script like this, its steps will be truncated at *dt*. Once all the components have run for *dt*, they sequentially update their output fields in the grid with their changes. This is the only time that information can be passed actively between each component (and the driving script, if it also makes changes to the grid fields within the loop); each component cannot "feel" changes being made by any other until *dt* has elapsed. Hence *dt* is best thought of as the "coupling timescale".

mats. These options are intended to allow interoperability with third-party software, especially geographical information systems, and also to allow Landlab data to be manipulated in and displayed with specialized visualization software (such as ParaView).

Landlab's standard interfaces also allow it to interact more easily with software frameworks developed by the geoscience and hydroscience communities. For instance, Landlab is already embedded within the Hydroshare collaboration environment, http://www.hydroshare.org. This means that Landlab models can be created and run within the Hydroshare data and modelling environment and can take advantage of that environment's shared data platform and metadata systems.

3.4.1 Dynamic model interaction and the Basic Model Interface

As noted in previous sections, Landlab has been designed from conception to be fully compliant with the Community Surface Dynamics Modelling System's Basic Model Interface (BMI) (Peckham et al., 2013). The BMI concept allows any two models describing the changes caused by surface processes to be coupled together, regardless of the vagaries of model gridding schemes, programming languages, or other low-level design choices. It does this by means of a standard interface (the Basic Model Interface, sensu stricto), which is callable for any BMI compliant model or component and includes generically applicable functions such as `initialize`, `update` (i.e. run one timestep), and `get_current_time`. The interface allows information about the current state of a simulation to be passed back and forth between running models in a manner that is agnostic in terms of implementation details.

The Landlab framework is designed such that the Landlab standard component interface can also expose a full BMI interface; in other words, *all Landlab components are also BMI-compliant components*. This means that by choosing Landlab as their model development environment, users also gain the ability to couple their models immediately with any other model in the CSDMS repository of BMI-compliant codes. This choice will also enhance the utility of Landlab to users who wish to implement component functionality alongside some other model using the CSDMS BMI or Web Modeling Tool (WMT) (Piper et al., 2015).

4 Validation, testing, and documentation

Landlab makes extensive use of Python's native documentation and code testing systems in order to test and validate the code base and to keep our documentation up to date. The development team exploits a combination of this Python "doctesting" and unit testing techniques to simultaneously test and document the code base. Doctests are code examples that can be included in the docstring that describes each Python method, and they list the expected output from each line of code as part of the documentation. Crucially, this code is then actually run whenever testing of the code base is triggered (for instance, by calling `landlab.test()`), and any doctests for which the output does not match the expected solution are recorded as either an error (tested function does not run cleanly) or a fail (output does not match). Because doctests are part of each function's docstring, they are also then automatically scraped from the code and included in the online documentation as examples for the user. In this way doctests allow us to help ensure Landlab functionality does not break as the code base evolves, while at the same time documenting for the user the way in which a given method, function, property, or component can be used.

Landlab also includes suites of unit tests. These are test scripts written specifically to exercise particular aspects of the code, and to check the output of that test against known correct solutions. Examples of when this is useful can occur in longer or more involved code, especially in components, where various different configurations of grid types and initial and boundary conditions need to be tested to ensure the component is robust under various different conditions. Unit tests differ from doctests in that they are not intended to be user-facing, although they are run alongside them when testing of the code base is triggered.

Almost all core Landlab functionality of both grid methods and components is now tested in this way. As of this version, around 1400 separate tests are run on the code each time testing is triggered, and the tests cover 80 % of the code base. Most of the remaining uncovered code is either challenging to adequately test (for example, plotting functions), not part of the core Landlab functionality (such as helper scripts involved in building releases), or deprecated. Tests are triggered automatically and remotely through the web-based applications Travis (Mac/Linux) and Appveyor (PC) whenever a new commit is made to either a branch or the master version of the code repository on GitHub, or when a new release of the code is built. These tests are performed on a range of supported Python versions, including both versions 2 and 3. Tests can also be triggered manually on a local machine by running a testing script included with Landlab, or by calling `landlab.test()` from an interactive Python environment.

5 Creating models with Landlab

We here illustrate some of the key functionality of Landlab by example, demonstrating its applicability across a variety of types of problem. We hope to emphasize here that Landlab *is not a landscape evolution model* (although it can be used to create them) – rather, it presents a framework under which a wide variety of different models can be implemented using its tools, including hydrologic, ecologic, and sedimentological models, as well as landscape evolution models. This section illustrates four possible contrasting model designs that can be implemented within the Landlab framework: a very simple "toy" geomorphic diffusion code that demonstrates the core functionality of the grid; a coupled stream power–hillslope diffusion model driven with a stochastic sequence of storms, illustrating some of Landlab's components; a cellular automaton, demonstrating a fundamentally different style of model implementation that is also enabled by Landlab's design; and a flood wave routing model, run on real topographic data ingested by Landlab. We hope that these examples will also serve as an illustration of the potential power of the Landlab framework to enable novel or under-explored process interaction studies (e.g. of vegetation on landscape evolution; of surface hydrology on stochastic surface processes).

5.1 A simple diffusion model

Although Landlab provides "off the shelf" process simulation code in the form of the components, Landlab also facilitates the design of models without using the components. The Landlab grids provide mapping, gradient, and divergence functions to make implementation of, for instance, finite-difference or finite-volume methods both concise and straightforward.

Here we illustrate this functionality using a simple finite-volume diffusion scheme, which here is representing the downslope flow of soil on hillslopes (Culling, 1963). We wish to represent the evolving form of a diffusional hillslope that is undergoing a constant uplift ($1\,\mathrm{mm\,yr^{-1}}$) with reference to a relative base level. In this case, the grid is radial and so roughly circular in plan view. Use of this particular configuration is intended in part to demonstrate the flexibility of Landlab's design, although this radial grid arrangement could perhaps be thought of in terms of response to a rising volcanic mound or salt diapir or another similar scenario with a radially symmetric uplift field.

The governing equations for this example are

$$\frac{\partial \eta}{\partial t} = U - \nabla q_\mathrm{s}, \tag{1}$$

$$q_\mathrm{s} = -D\nabla\eta, \tag{2}$$

where η is land-surface elevation, t is time, U is the rate of vertical motion ("uplift") of rock relative to base level, q_s is volumetric sediment flux per unit slope width, and D is a transport coefficient with dimensions of length squared per time.

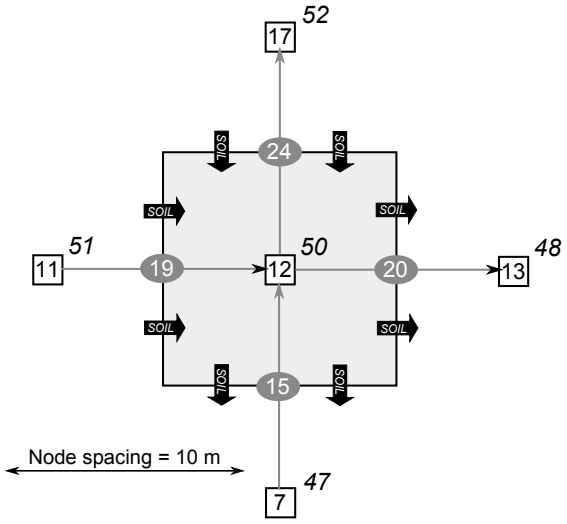
For our example model, Eq. (2) will be discretized and solved using a finite-volume solution scheme. Consider a cell of surface area a that is surrounded by N neighbouring cells (Fig. 7). We can integrate Eq. (1) over the surface area of the cell:

$$\int_a \frac{\partial \eta}{\partial t}\mathrm{d}a = \int_a U\,\mathrm{d}a - \int_a \nabla q_\mathrm{s}\mathrm{d}a. \tag{3}$$

Applying the divergence theorem to the last term on the right, and evaluating the other two integrals,

$$a\frac{\partial \overline{\eta}}{\partial t} = Ua - \oint_p q_\mathrm{s}(p)\boldsymbol{n}\mathrm{d}p, \tag{4}$$

where $\overline{\eta}$ is the average elevation within the cell, p represents position along the perimeter of the cell, and $\boldsymbol{n}$ is a unit vector normal to the perimeter and pointing outward. The last term is a line integral that represents adding up all the inflows and outflows of mass along the cell's perimeter. If the cell is a



Calculating gradients at links:

```
>>> grad = grid.calc_grad_at_link(elev)
>>> grid.links_at_node[12]
array([20, 24, 19, 15])
>>> grad[grid.links_at_node[12]]
array([-0.2,  0.2, -0.1, 0.3])
```

Calculating fluxes from gradients:

```
>>> q = -0.01 * grad
>>> q[20]
0.002
>>> q[15]
-0.003
```

Calculating flux divergence:

```
>>> divq = grid.calc_flux_div_at_node(q)
>>> divq[12]
0.0002
```

Figure 7. Schematic illustration showing how Landlab's grid geometry may be used to construct a finite-volume numerical scheme. White squares represent nodes, with example node IDs given for a 5×5 raster grid. Grey ovals show the centre points of the links, with the link IDs given. In this example, we assume that we have a node field called "elev" whose values represent the altitude of the land surface at various node locations (example values shown in italics next to each node). Black arrows indicate direction of soil flow (in the downhill direction). A finite-volume solution for a diffusion model can be implemented by (1) calculating the gradient at each pair of adjacent nodes and assigning it to the corresponding link (lines 1–3 in the code snippet below); (2) multiplying by a transport-rate coefficient (and -1) to obtain unit flux (lines 4–6); and (3) multiplying the unit flux at each cell face by the width of that face, adding up the inflows and outflows, and dividing by cell area to obtain flux divergence (lines 7 and 8).

polygon with N faces, this last term can be replaced by a summation:

$$\frac{\partial \overline{\eta}}{\partial t} = U - \frac{1}{a}\sum_{k=1}^{N} q_{sk} w_k \tag{5}$$

where q_{sk} is the unit flux at face k, positive outward, and w_k is the width of face k.

We will implement this solution in Landlab by assigning to each node i the value of the average elevation within its cell, $\overline{\eta}_i$ (for notational convenience, we will drop the use of the overbar below). To calculate the flux at each face, we first need to calculate the topographic gradient at each face. We will do this by taking the elevation difference between each neighbouring pair of nodes, dividing by the length of the link that connects them, and then assigning the resulting gradient value to the relevant link. The gradient at link j is therefore calculated as

$$G_j = \frac{\eta_{\mathrm{H}_j} - \eta_{\mathrm{T}_j}}{L_j}, \tag{6}$$

where η_{H_j} and η_{T_j} are the elevation values at link j's head and tail nodes, respectively, and L_j is the length of link j. In Landlab's gridding engine, the calculation of link-based gradients in a node-based scalar quantity like η is handled by the grid method `calc_grad_at_link`, which takes a node array or field name as an argument and returns a link array. Figure 7 illustrates how values of η defined at nodes can be used to calculate gradients at links, and then the gradients can be used to calculate the net flux into and out of a cell.

In our diffusion example, the summation of fluxes along the cell faces is calculated as follows:

$$\sum_{k=1}^{N} q_{sk} w_k = \frac{D}{a_i}\sum_{k=1}^{N} \delta_{ik} G_k w_k, \tag{7}$$

where δ_{ik} indicates the direction of link k relative to the cell i: if $\delta_{ik} = -1$, the link points outward from the cell; if $\delta_{ik} = +1$, the link points inward.

To calculate flux divergence using this finite-volume approach, Landlab provides the general grid method `calc_flux_div_at_node`, which takes a link-based array of unit fluxes as an input and returns a node array that contains the sum of in/out fluxes (divided by cell area) at each node (Fig. 7). Values at perimeter nodes, which lack cells, are ignored. In keeping with the standard definition of the divergence operation, the function returns positive values where the net flux is outward and negative values where it is inward.

In the diffusion example shown in Fig. 8, the time derivative is discretized using a simple forward-Euler explicit method, such that the values of elevation at the new timestep $t + 1$ are calculated from values at the old timestep:

$$\eta_i^{t+1} = \eta_i^t + \Delta t \left[U + \frac{D}{a_i}\sum_{k=1}^{N} \delta_{ik} G_k w_k \right], \tag{8}$$

where the superscript indicates timestep, and the quantity in brackets is evaluated at timestep t. The code to implement the model is shown in Fig. 8. Note the use of the `calc_grad_at_link` and `calc_flux_div_at_node` methods (and note also that $U = 0$ in this example).

Code to implement a simple diffusion model on a radial Landlab grid:

```
 1. >>> from landlab import RadialModelGrid, imshow_grid
 2. >>> from matplotlib.pyplot import show
 3. >>> mg = RadialModelGrid(num_shells=10, dr=10.)
 4. >>> z = mg.zeros('node')
 5. >>> qs = mg.zeros('link')
 6. >>> diffusivity = 1.e-2
 7. >>> dt = 0.2 * mg.length_of_link.min() ** 2. / diffusivity
 8. >>> for i in range(500):
 9. ...     z[mg.core_nodes] += 0.001*dt
10. ...     g = mg.calc_grad_at_link(z)
11. ...     qs[mg.active_links] = -diffusivity * g[mg.active_links]
12. ...     dqsdx = mg.calc_flux_div_at_node(qs)
13. ...     dzdt = -dqsdx
14. ...     z[mg.core_nodes] += dzdt[mg.core_nodes] * dt
15. >>> imshow_grid(mg, z, grid_units=('m', 'm'), var_name='Elevation (m)')
16. >>> show()
```

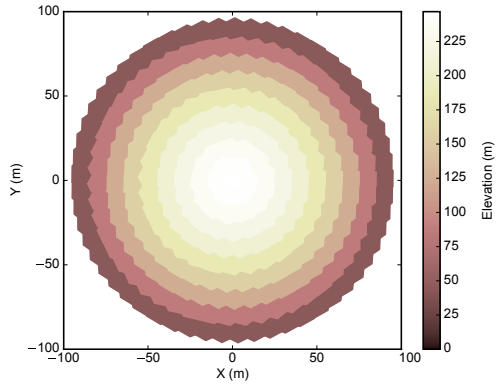


Figure 8. A simple finite-volume hillslope diffusion model implemented in Landlab. Values adopted here are within typical terrestrial ranges for hillslope length (~ 100 m, controlled from line 3), hillslope diffusivity ($0.01\,m^2\,yr^{-1}$, line 6) (Fernandes and Dietrich, 1997), total time of run (around a million years, since *dt* is ~ 1833 years, lines 7–8), and uplift rate relative to base level ($0.001\,m\,yr^{-1}$, line 9).

An advantage of the finite-volume approach is that it can be applied to cells of any shape. For instance, it can be used with hexagonal cells, or with Voronoi polygons as in the example in Fig. 8.

This model can be implemented in Landlab and plotted in as few as 16 lines of code (Fig. 8). Here, line 1 imports the Landlab classes and functions we will use, and line 2 imports the `show()` function from matplotlib that will let us display the plot. Line 3 instantiates the Landlab grid object. This example uses a RadialModelGrid, but the same code would work with any grid type. Lines 4–6 initialize data for the model run. z will be the land surface elevation at each node; qs will be the volumetric sediment flux per unit width along each link. Note that this implementation is consciously not using data stored as Landlab fields in order to illustrate that this is not a requirement; however, it would be trivial to modify lines 4 and 5 to create the data as fields on the grid, and the remainder of this script would be unchanged. Line 7 is the first line that actually begins the calculations that perform the diffusion. This line calculates a Courant–Friedrichs–Lewy (CFL) stability condition (Slingerland and Kump, 2011) for the maximum stable timestep for the finite-volume scheme we are about to implement.

Code to implement a simple diffusion model on a radial Landlab grid, using Landlab components:

```
 1. >>> from landlab import RadialModelGrid, imshow_grid
 2. >>> from landlab.components import LinearDiffuser
 3. >>> from matplotlib.pyplot import show
 4. >>> mg = RadialModelGrid(num_shells=10, dr=10.)
 5. >>> z = mg.add_zeros('node', 'topographic__elevation')
 6. >>> dt = 2000.  # no longer the stable timestep
 7. >>> ld = LinearDiffuser(mg, linear_diffusivity=1.e-2)
 8. >>> for i in range(500):
 9. ...     z[mg.core_nodes] += 0.001*dt
10. ...     ld.run_one_step(dt)
11. >>> imshow_grid(mg, z, grid_units=('m', 'm'), var_name='Elevation (m)')
12. >>> show()
```

Figure 9. Hillslope diffusion implemented in Landlab using a component. Compare to Fig. 8. Note that this version is more concise, and that timestep stability is now handled internally within the component.

Lines 8–14 implement a time loop, within which the diffusion occurs. The core (i.e. interior) nodes of the grid are uplifted at a rate of 0.001 length units per time unit relative to base level. Lines 10–14 implement the meat of the differencing scheme, where we use a staggered grid to solve the discretized diffusion equation (Eq. 8). The depth-integrated fluxes on the links are calculated as the product of the diffusivity parameter D and the topographic gradient at the links (lines 10, 11), taking care to calculate the flux only on active links. The flux divergence is then calculated at each node based on the fluxes on the links to which is it adjoined (line 12). Note that Landlab enables each of these operations to be performed with a single grid method. The final lines of the code invoke the standard Landlab plotter, then display the output. Although we have not specified any particular units in our calculation, in line 15 we assert that the length unit is metres and the time unit is years.

Note that this same result could have been achieved even more concisely using Landlab's in-built LinearDiffuser component. The equivalent code is shown in Fig. 9. Not only are the implementation details of the scheme now handled entirely within the component, but so also is internal subdivision of the provided timestep to meet the necessary stability conditions for the simulation. Additionally, the elevation data are now passed into the component as the field "topographic__elevation" – which is attached to the grid – rather than as a separate variable (lines 5, 7), as discussed in Sect. 3.2.

5.2 Coupling diffusion to stream power with a storm sequence

The next example illustrates a simple model for the evolution of an eroding and uplifting landscape, explicitly representing channel incision and hillslope processes. In this model, we also explicitly represent time variability of water input to the system (i.e. storms). In technical terms, the example is designed to show in more detail the use and coupling of several Landlab components: the FlowRouter, the StreamPowerEroder, the DepressionFinderAndRouter, the LinearDif-

fuser, and the PrecipitationDistribution classes. The aim here is to demonstrate how Landlab couples components and to illustrate several different component styles.

Here, channel incision processes are represented by the stream power law (Howard, 1994; Lague, 2014; Whipple and Tucker, 1999), which says that incision rate, E, of a stream is proportional to a product of powers of channel discharge, Q, and local channel bed slope, S. In this version, we also include an incision threshold, C, below which incision is forbidden:

$$\begin{aligned} E &= K Q^m S^n - C \quad \text{if } C < K Q^m S^n \\ E &= 0 \quad \text{if } C \geq K Q^m S^n. \end{aligned} \tag{9}$$

In this case $m = 0.5$, $K = 1 \times 10^{-5}\,\mathrm{m}^{-0.5}\,\mathrm{yr}^{-0.5}$, $C = 1 \times 10^{-5}\,\mathrm{m\,yr}^{-1}$, which are fairly typical and widely adopted values for a generic erosional upland landscape (Harel et al., 2016; Tucker and Whipple, 2002). Here we also adopt $n = 1$. This is primarily to maintain dimensionally sensible units for K while still honouring the widely observed ratio of $m/n \sim 0.5$, interpreted from channel concavities of natural rivers at apparent topographic steady state. Nonetheless, we note $n > 1$ in some global data compilations for stream power where $C = 0$, and suggest our incorporation of an explicit erosion threshold makes our choice of $n = 1$ reasonable (Harel et al., 2016). We shall see that this set of values together produces a plausible total landscape relief of order 1 km for catchments of maximum length ~ 5 km, which is within the range expected for real catchments of this scale in tectonically active regions. Other forms of stream-power-based incision rules are also possible using this component, but these are not illustrated here.

The Landlab StreamPowerEroder and FlowRouter components deployed here use the "Fastscape" algorithms of Braun and Willett (2013). This solution scheme is implicit and order n, and permits arbitrarily long, numerically stable timesteps to be taken. The Fastscape algorithm requires out-of-order (i.e. upstream order) iteration through the nodes, but pure Python code has relatively poor speed performance when executing explicit loops or iterations through arrays. For this reason, both the stream power and flow routing components also use compiled Cython (see Sect. 2.2) to accelerate these speed bottlenecks in the code. (The release version of Landlab distributes this code in pre-compiled form to users.) The run method of the component performs as order n, and as expected is unaffected by grid type (in this demonstration, raster vs. hex grids). The initialization of the grid and components adds a very small overhead which also increases close to linearly with grid size (Fig. 10; code in the Supplement as Script S2). This overhead reflects the calculations necessary to build the data structures describing the grid's connectivity, and is significantly greater for Voronoi grids compared to rasters, due to the iterative calculations required to assemble Voronoi grid-connectivity arrays.

The final topographies from the raster and hexagonal implementations of this pure stream power component are shown in Fig. 11. The code can be seen in the Supplement as Script S3. It conforms to a typical form for a Landlab driver script:

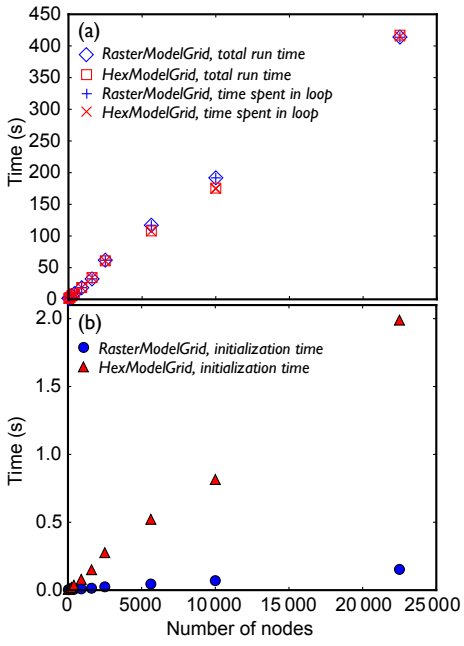


Figure 10. Performance of a Landlab-built model of landform evolution, using the StreamPowerEroder, FlowRouter, and PrecipitationDistribution components on grids of different types and sizes. Runs were performed on a mid-2014 MacBook Pro, and each data point represents the mean of five runs. **(a)** Total time for a simulation of 3 million years, implementing a stochastic storm sequence of around 3000 distinct stormy intervals. Both the total time to run and the time spent in the loop in the code that iterates forward in time are shown, and they are practically indistinguishable in most cases. The time to run the components is close to linear with number of nodes, as expected for the Fastscape algorithms (see main text). **(b)** The time spent initializing the grids and components in each case (i.e. the total time less the time spent in loop from panel **a**). Setting up a Voronoi-based grid is more computationally expensive than a raster, but both are quick in absolute terms, and both are close to linearly scaled with the number of nodes. In both graphs, small deviations from linear scaling occur, probably related to the interaction of Python's dynamic memory management with the size of the random access memory on the individual machine.

1. Import necessary Python libraries, including from Landlab.
2. Instantiate a grid object.
3. Create input fields and set the grid initial and boundary conditions.
4. Instantiate the components.
5. Perform a loop to run the components.
6. Finalize, plot, save, and/or export.

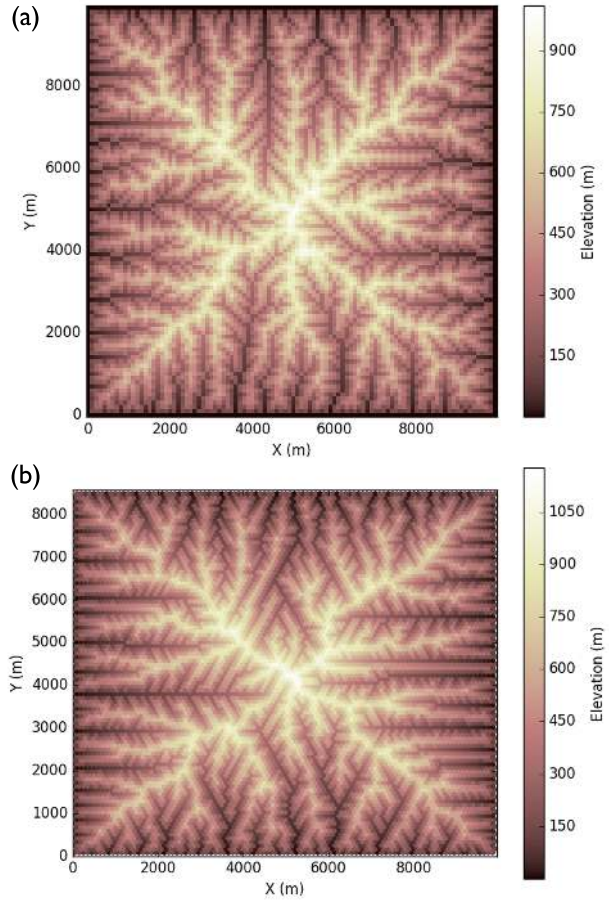


Figure 11. Simulated topographies produced from a simple stream power-based fluvial incision rule, combining the StreamPowerEroder, FlowRouter, and PrecipitationDistribution components. The same model set up is implemented on both a RasterModelGrid **(a)** and a HexModelGrid **(b)**, using the same random seed to generate the topography. Note the vertical–horizontal asymmetry in channel network planform visible in panel **(b)**, an expected outcome of the three axes of mirror symmetry running though a hexagonal grid. The linearity of these catchment planforms is enhanced by the presence of an erosion threshold.

In this case, the model is driven by a stochastic storm generator (the PrecipitationDistribution class), based on that suggested by Eagleson (1978) and similar to the one underlying the CHILD landscape evolution model (Tucker and Bras, 2000; Tucker et al., 2001b). Unlike CHILD, but in keeping with Eagleson's original derivation, here an explicit inverse relationship between storm length and intensity is built into the distribution, by calculating storm water depth as a gamma-distributed random variable and then deriving storm intensity as the quotient of depth and (exponentially distributed) duration. This approach prevents unrealistic long-duration, high-intensity events from being sampled (Eagleson, 1978). The PrecipitationDistribution class provides a method that yields tuples of interval durations and rainfall intensities as a true Python generator – in other words, the code block below the generator will repeat with fresh values for each iteration until the total time is elapsed, at which point the loop will cease (see lines 46–53 in the code). This makes the implementation of the "run" loop both efficient and concise, as well as being a classically "Pythonic" way to implement this kind of loop. In this instance, the parameters for the PrecipitationDistribution have been chosen to represent a mean annual rainfall rate of around $5\,\mathrm{m\,yr^{-1}}$, and with rainfall occurring around 10 % of the time.

The switch between grid types involves changing a single line of code (see the logical test at lines 15–18). Note that although the total number of nodes and the number of rows and columns is identical in both cases, the hexagonal grid is rectangular rather than square due to the single axis of mirror symmetry present in a tessellation of regular hexagons. (The HexModelGrid class provides flags allowing control both of the orientation of this symmetry axis, and also the shape of the perimeter of the grid – rectangular or hexagonal.)

The addition of the linear diffusion component, LinearDiffuser, is performed simply by creating an instance of that class and then incorporating its run method into the loop (code S4, lines 40 and 49). As in previous examples, each component is responsible for managing its own internal numerical stability – in this case, if the LinearDiffuser run method receives an input *dt* that exceeds the Courant–Friedrichs–Lewy stability limit, that timestep will be internally subdivided as necessary within the component.

In this example, because diffusion can occur independently of stream incision, it is possible that diffusion can sever the flow paths of the FlowRouter and create internal basins. Because of this possibility, this version of the code also includes a lake-filling algorithm, implemented as the component DepressionFinderAndRouter. The lake-filling algorithm identifies closed depressions in the topography then reroutes flow across them, and is based on the algorithm of Tucker et al. (2001b). The final topography of the coupled stream power and linear diffusion model is shown in Fig. 12.

5.3 Landlab as a cellular automaton

Much of this paper focuses on Landlab as a tool for the implementation of numerical solutions to two-dimensional partial differential equations, as many geomorphic process laws (sensu Dietrich et al., 2003) have been couched in the language of differential equations. However, Landlab can also act as a powerful environment for the implementation of cellular models. Landlab provides a set of tools for the construction of "continuous-time stochastic" (CTS) cellular automata (CA). This interface within the main body of Landlab is known as CellLab-CTS (Tucker et al., 2016). It enables efficient creation of CTS models: a user needs only to specify the states and transition rules and write a short Python script to initialize and run a CellLabCTSModel object. Fig-

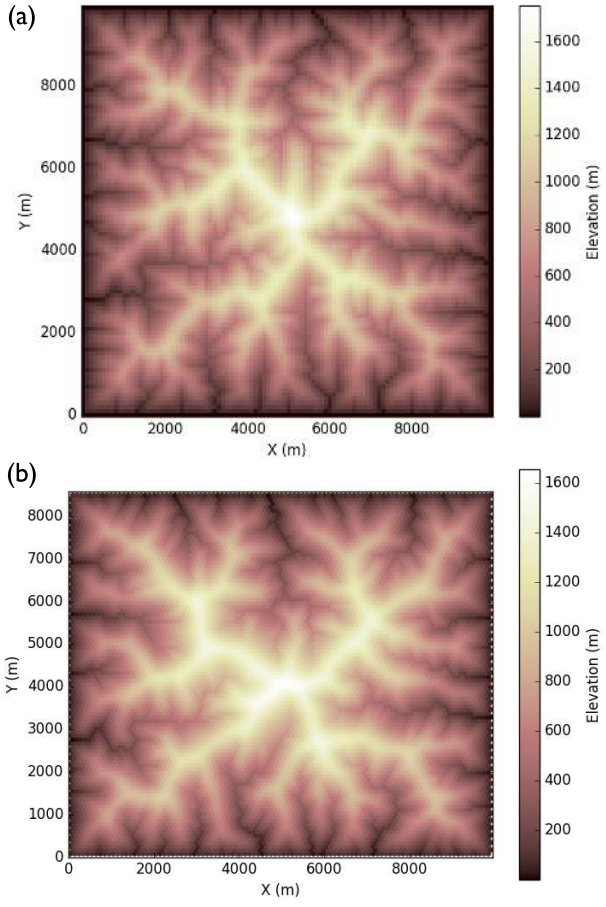


Figure 12. Simulated topographies produced from a coupled hillslope and channel evolution model, combining the StreamPowerEroder, FlowRouter, and LinearDiffuser components. A storm sequence is provided by the PrecipitationDistribution component, and discharge is routed across depressions in topography using DepressionFinderAndRouter. Stream power parameters are identical to those in Fig. 11. The same model setup is implemented on both a RasterModelGrid **(a)** and a HexModelGrid **(b)**, using the same random seed to generate the topography. Despite the differences in grid organization, planform drainage pattern remains fairly similar between the two cases.

ure 13 shows output from a CellLab-CTS model implementing a lattice-grain algorithm (Tucker et al., 2016).

Landlab can also be used to construct traditional discrete-timestep cellular automata. An example is provided by developing an ecohydrology model in Landlab (Fig. 14a, code S5), which is in part an implementation of the Cellular Automata Tree-Grass-Shrub Simulator (CATGraSS) (Caracciolo et al., 2016a, b, 2014; Zhou et al., 2013). CATGraSS couples local vegetation dynamics, which simulate biomass production based on local soil moisture and potential evapotranspiration, and plant establishment and mortality based on competition for resources and space at each cell of a gridded model domain. Each cell in the domain can be occupied by one plant functional type (PFT): each cell is flagged as Tree, Shrub, Grass or Bare (left unoccupied).

CATGraSS is driven by rainfall pulses and solar radiation. In Landlab, the model is implemented as a set of interacting components, each of which describes a different element of the coupled system: PrecipitationDistribution, Radiation, PotentialEvapotranspiration, SoilMoisture, Vegetation, and VegCA. This means that each process can also operate in isolation, outside the context of this example model. The PrecipitationDistribution component simulates the random arrival of storm pulses. Precipitation characteristics are based on the seasonal rainfall statistics of a region and characterized by exponential distributions of storm and inter-storm duration, and a gamma distribution of water depth as a function of storm duration. Storm pulses recharge the soil moisture storage, represented as a single bucket (Laio et al., 2001). The Radiation component calculates daily average extra-terrestrial and clear-sky shortwave radiation incident on a flat surface, based on latitude and day of the year (ASCE-EWRI, 2005). This component also calculates daily radiation ratio, defined as the ratio of cosine of solar angle of incidence for the true sloped surface to that for a flat surface (Bras, 1990). The Radiation component does not explicitly calculate diffused and reflected radiation. The PotentialEvapotranspiration component uses the radiation ratio to calculate spatial net radiation using daily maximum and minimum temperature, and potential evapotranspiration (ASCE-EWRI, 2005; Zhou et al., 2013). The SoilMoisture component models local root-zone soil moisture dynamics depending on the PFT that occupies the corresponding cell at a given time (Laio et al., 2001). The Vegetation component simulates temporal dynamics of above-ground live and dead biomass, as well as leaf area index (LAI). It does this by computing net primary productivity (NPP) based on the concept of water-use efficiency (WUE) that relates NPP to actual evapotranspiration (ET) and vegetation foliage loss due to water stress and senescence (Istanbulluoglu et al., 2012; Zhou et al., 2013). The VegCA component handles the spatial organization of PFTs, through plant establishment, competition, and mortality, by combining deterministic and probabilistic rules. Plant establishment is driven by seed dispersal and water stress, while mortality is related to water stress, plant age, and disturbances (Zhou et al., 2013).

This example ecohydrology model and its constituent components can work both on grids imported from a digital elevation model (DEM) using the `read_esri_ascii` utility (see also Sect. 5.4) and on synthetic grids created using the RasterModelGrid library. In the example illustrated in Fig. 14b and c, we use the example ecohydrology model (code S5) to simulate plant competition in a semi-arid basin in Sevilleta, New Mexico, USA, modelling the plant species found in this area (Zhou et al., 2013). Because of the stochastic nature of the simulations in this example, potential evapotranspiration is represented by a sinusoidal function of day of the year (Zhou et al., 2013). The domain is initialized with

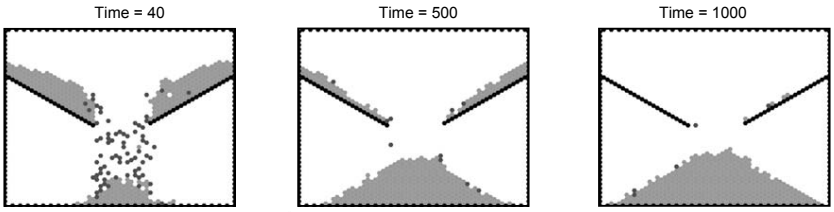


Figure 13. Example of a CellLab-CTS model. Here the CellLab-CTS framework has been used to implement a model of granular mechanics. The model has eight node states, representing air (white), a resting grain (light grey), and a grain moving in each of the six lattice directions (all coded as dark grey). Grid edges and immobile walls are treated as CLOSED_BOUNDARY Landlab boundary conditions (black). Transition rules are used to model grain motion, grain collision, and gravity (from Tucker et al., 2016).

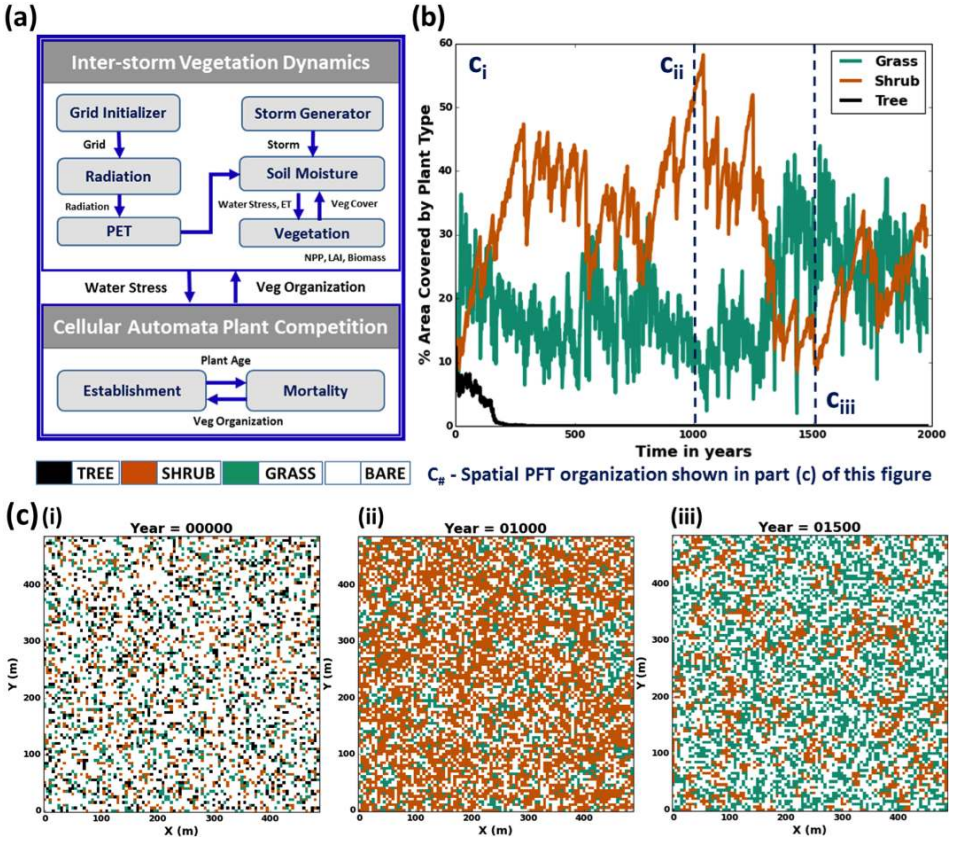


Figure 14. Implementation of an ecohydrology model in Landlab. **(a)** Schematic illustration of coupling among different Landlab components for the CaTGraSS application. **(b)** Demonstration of the model on a flat surface with semi-arid climate similar to that of Sevilleta, New Mexico, USA (Zhou et al., 2013). This figure plots percentage of space occupied by each PFT with time. **(c)** Spatial organization of PFTs at different times during the model run. These plots illustrate competition between different PFTs for space and resources. Trees die early within the first 300 years due to unfavourable climatic conditions and competition from shrubs and grass. The ecosystem swings between shrub-dominant and grass-dominant states for the next 1600 years.

randomly assigned PFTs with random spatial distribution of ages (Fig. 14ci). All PFTs initially have an identical cover fraction in the domain. Local vegetation dynamics are simulated at inter-storm timesteps, and plant competition is modelled at annual timesteps. In the simulations, trees are outcompeted by drought-tolerant shrubs and grasses in the first few hundred years, consistent with regional observations in central New Mexico (Zhou et al., 2013). Shrubs and grasses coexist in the modelled domain with alternating periods of shrub and grass dominance. Note that shrubs cluster as they propagate in space due to seed dispersal from mature shrub plants.

5.4 Landlab as a hydrological modelling environment

Landlab also contains several surface water flow generators, including an explicit two-dimensional solution for the shallow water equations. The OverlandFlow component has been adapted from the flood inundation model described by de Almeida et al. (2012). Their algorithm was derived for use on structured grids, and the Landlab implementation only works with the RasterModelGrid library. Water discharge is calculated on each active link within the model domain, simulating a hydrograph at each link location.

In many flood-wave routing models, a small timestep must be used to prevent instabilities, which often manifest as "checkerboard" patterns of water depth, from emerging. To maximize computational performance of the OverlandFlow component, an adaptive timestep is used to find the largest timestep that adheres to the CFL stability condition (Hunter et al., 2005). To further enhance the stability, the OverlandFlow component also contains stability criteria so that the component can operate not only on low-slope, urban areas but also on steeper terrain, such as mountainous watersheds (Adams et al., 2016). The OverlandFlow component was designed for structured grids, and it assumes water can only move in the four cardinal directions. This is easily accommodated within Landlab, and several other components (e.g. the FlowRouter and others in the example presented below) can also be optionally instructed using keywords to only use node neighbours in these cardinal directions.

An example script running the OverlandFlow component can be seen in the Supplement as Script S6. It follows a similar pattern to scripts outlined in earlier parts of this section, with import of the Landlab and other Python classes and functions needed, followed by grid creation, component instantiation, component execution in a loop, and then finalization and plotting. Notably, this script uses an imported digital elevation model (DEM) of a real landscape over which to route flow, which is ingested into Landlab using the `read_esri_ascii` function contained in Landlab's input and output utilities. Use is made of Landlab's native boundary handling system to designate nodes of the grid outside of the irregularly shaped catchment as closed, excluding them from the calculations.

This example combines the OverlandFlow component with the SinkFiller. The SinkFiller is run on the initial topography prior to the simulated storm, and fills any local depressions present in the surface. This has been done to enable full drainage of all the water from the network, and to permit evaluation of the full water budget at the outlet. However, in general the OverlandFlow component will happily run on landscapes that do contain pits. In this example, a rainfall rate of $25\,\mathrm{mm\,h^{-1}}$ was run over the watershed DEM for 1 h. The resulting hydrograph (water discharge over time) is plotted at the outlet. Water depth across the domain is also plotted to show the wave front propagating downstream (Fig. 15). As expected, the total hydrograph duration is several times the length of the storm, and the peak in the hydrograph lags behind the storm itself significantly, in this case by more than an hour.

6 Conclusions

Landlab is an open-source, Python-based software toolkit designed to accelerate the development of new process models. It consists of a gridding engine, a set of components describing individual surface processes, and a set of utilities for data input, output, and visualization. Landlab not only permits the creation of models by combination of existing components but is also optimized to aid in the design of new process components. The code base is thoroughly documented both online and within the code itself, and each release undergoes an automated testing procedure to ensure its robustness. A set of tutorials and examples to help learn about Landlab is also provided.

Landlab is explicitly designed to interface with other software, and in particular, with other models of surface processes. It exposes a CSDMS Basic Model Interface. It can serve as a platform to develop both continuum-based and cellular-automaton-style models, and potentially to have the two model styles interact on the same grid. We illustrate some of the functionality of Landlab and its existing components with a suite of examples drawn from geomorphology, ecology, and hydrology. The examples provided in this paper illustrate the wide diversity of scientific questions that can be addressed using Landlab-built models.

7 Code availability

This text describes Landlab version 1.0.2 ("Rapunzel"), which was released in November 2016. The source code for this version is maintained in a Git repository hosted on GitHub at https://github.com/landlab/landlab/releases/tag/v1.0.2 (the latest development version of Landlab is always available at http://github.com/landlab/landlab). Landlab can also be installed as a release version, including precompiled binary files containing Cython extensions, through the *conda* and *pip* Python package management systems, as described in the online documentation. Documentation and installation instructions for the most current release version of Landlab are provided at http://landlab.github.io. Software dependencies are listed at https://landlab.github.io under "Install". To the best of our knowledge, Landlab will operate on any system that meets these software requirements; as of the time of writing, Landlab is known to work on, and is tested for, recent-generation Mac, Linux, and Windows platforms running Python 2.7, 3.4, and 3.5. Landlab and its components are distributed under an MIT open-source license.

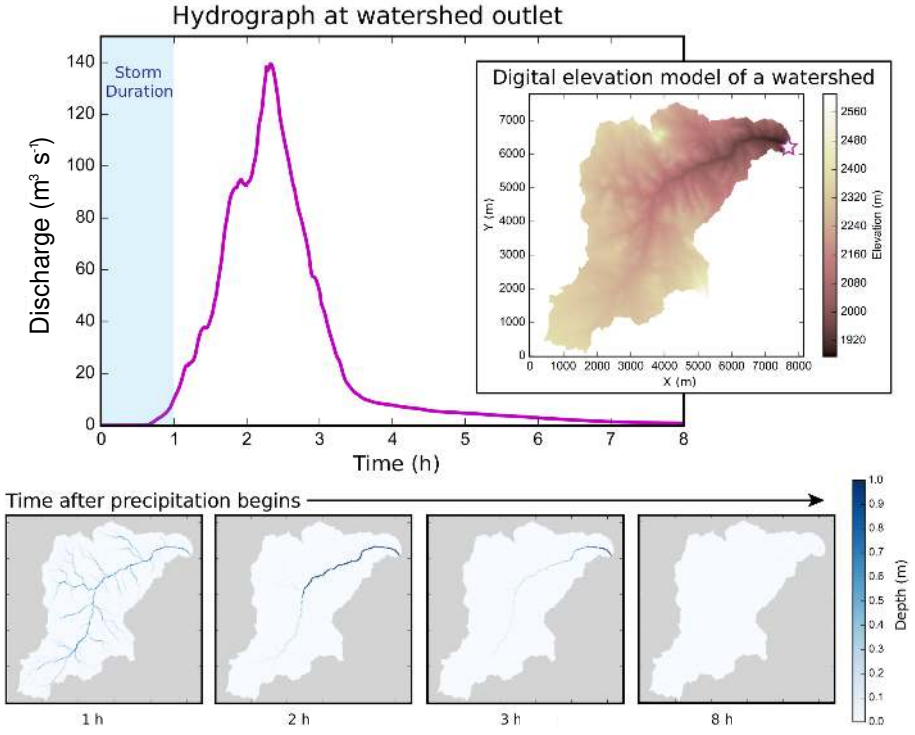


Figure 15. Demonstration of OverlandFlow component capabilities. The example shows development of a hydrograph in a catchment drawn from an airborne lidar-derived DEM of the Spring Creek catchment in central Colorado, USA. The run uses a constant rainfall rate of 25 mm h^{-1} and a storm duration of 1 h. The hydrograph persists for almost eight model hours, and water depth as plotted at several intervals after the start of the precipitation event: 1 (the end of the storm), 2, 3, and 8 h.

8 Data availability

Code to reproduce figures as found in this manuscript is either presented within this main text or can be found in the Supplement. The Spring Creek DEM data presented in Fig. 15 are a subset of the lidar dataset for Raleigh Peak, Colorado: May 2010 (NCALM, 2010). Lidar data acquisition and processing completed by the National Center for Airborne Laser Mapping (NCALM – http://www.ncalm.org). NCALM funding provided by NSF's Division of Earth Sciences, Instrumentation and Facilities Program. EAR-1043051. The lidar data have been cropped to a specific sub-basin and filled to remove pits, as described in the main text. The cropped and filled version of the data may be found as an ASCII-formatted text file in the dataset Adams (2016) as the file "SpringCreek_DEM.asc".

The Supplement related to this article is available online at doi:10.5194/esurf-5-21-2017-supplement.

Acknowledgement. This research was supported by the US National Science Foundation (ACI-1147454 (GET), ACI-1450409 (GET), ACI-1450338 (NMG), ACI-1147519 (NMG) ACI-1450412 (EI), ACI-1148305 (EI), and EAR-1246761 (through an NCED2 postdoctoral fellowship to DEJH)). We thank B. Campforts, W. Schwanghart, and A. Wickert for their helpful reviews of an earlier version of this paper, and S. Mudd for serving as editor on the manuscript. Landlab could not exist without the wider open-source software in science movement, and particularly open-source enthusiasts who are members of the surface process modelling community. We are particularly indebted to the members of the CSDMS Integration Facility for the best practices put forward and advice offered.

Edited by: S. Mudd
Reviewed by: A. Wickert and B. Campforts

References

Adams, J. M.: landlab/pub_adams_etal_gmd v0.2 (Data set), Zenodo, doi:10.5281/zenodo.162058, 2016.

Adams, J. M., Nudurupati, S. S., Gasparini, N. M., Hobley, D. E. J., Hutton, E., Tucker, G. E., and Istanbulluoglu, E.: Landlab: Sustainable Software Development in Practice, The Second Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2), New Orleans, LA, USA, 16 November 2014, doi:10.6084/m9.figshare.1097629.v6, 2014.

Adams, J. M., Gasparini, N. M., Hobley, D. E. J., Tucker, G. E., Hutton, E. W. H., Nudurupati, S. S., and Istanbulluoglu, E.: The Landlab OverlandFlow component: a Python library for computing shallow-water flow across watersheds, Geosci. Model Dev. Discuss., doi:10.5194/gmd-2016-277, in review, 2016.

ASCE-EWRI: The ASCE standardized reference evapotranspiration equation, in: Standardization of Reference Evapotranspiration Task Committee Final Report, edited by: Allen, R. G., Walter, I. A., Elliot, R. L., Howell, T. A., Itenfisu, D., Jensen, M. E., and Snyder, R. L., Technical Committee report to the Environmental and Water Resources Institute of the American Society of Civil Engineers from the Task Committee on Standardization of Reference Evapotranspiration, Reston, VA, USA, 2005.

Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., and Venters, C. C.: Sustainability design and software: the karlskrona manifesto, in: IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015, 467–476, doi:10.1109/ICSE.2015.179, 2015.

Berger, K. P.: Surface water–groundwater interaction: the spatial organization of hydrologic processes over complex terrain, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 242 pp., 2000.

Bras, R. L.: Hydrology: an introduction to hydrologic science, Addison Wesley Publishing Company, Boston, Mass., USA, 643 pp., 1990.

Braun, J. and Sambridge, M.: Modelling landscape evolution on geological time scales: a new method based on irregular spatial discretization, Basin Res., 9, 27–52, 1997.

Braun, J. and Willett, S. D.: A very efficient $O(n)$, implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution, Geomorphology, 180–181, 170–179, doi:10.1016/j.geomorph.2012.10.008, 2013.

Caracciolo, D., Noto, L. V., Istanbulluoglu, E., Fatichi, S., and Zhou, X.: Climate change and Ecotone boundaries: Insights from a cellular automata ecohydrology model in a Mediterranean catchment with topography controlled vegetation patterns, Adv. Water Resour., 73, 159–175, doi:10.1016/j.advwatres.2014.08.001, 2014.

Caracciolo, D., Istanbulluoglu, E., and Noto, L. V.: An Ecohydrological Cellular Automata Model Investigation of Juniper Tree Encroachment in a Western North American Landscape, Ecosystems, doi:10.1007/s10021-016-0096-6, in press, 2016a.

Caracciolo, D., Istanbulluoglu, E., Noto, L. V., and Collins, S. L.: Mechanisms of shrub encroachment into Northern Chihuahuan Desert grasslands and impacts of climate change investigated using a cellular automata model, Adv. Water Resour., 91, 46–62, doi:10.1016/j.advwatres.2016.03.002, 2016b.

Chue Hong, N.: We are the 92 %, The Second Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2), New Orleans, LA, USA, 16 November 2014, doi:10.6084/m9.figshare.1243288.v1, 2014.

Crick, T., Hall, B. A., and Ishtiaq, S.: "Can I Implement Your Algorithm?": A Model for Reproducible Research Software, The Second Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2), New Orleans, LA, USA, 16 November 2014, arXiv:1407.5981v2 [cs.SE], 2014.

Culling, W.: Soil creep and the development of hillside slopes, J. Geol., 71, 127–161, 1963.

de Almeida, G. A. M., Bates, P., Freer, J. E., and Souvignet, M.: Improving the stability of a simple formulation of the shallow water equations for 2-D flood modeling, Water Resour. Res., 48, W05528, doi:10.1029/2011WR011570, 2012.

Dietrich, W. E., Bellugi, D. G., Sklar, L. S., Stock, J. D., Heimsath, A. M., and Roering, J. J.: Geomorphic Transport Laws for Predicting Landscape Form and Dynamics, in: Prediction in Geomorphology, Geophysical Monograph-American Geophysical Union, Washington, DC, USA, 135, 1–30, 2003.

Eagleson, P. S.: Climate, soil, and vegetation: 2. The distribution of annual precipitation derived from observed storm sequences, Water Resour. Res., 14, 713–721, doi:10.1029/WR014i005p00713, 1978.

Easterbrook, S. M.: Open code for open science?, Nat. Geosci., 7, 779–781, doi:10.1038/ngeo2283, 2014.

Fernandes, N. F. and Dietrich, W. E.: Hillslope evolution by diffusive processes: The timescale for equilibrium adjustments, Water Resour. Res., 33, 1307–1318, doi:10.1029/97WR00534, 1997.

Goren, L., Willett, S. D., Herman, F., and Braun, J.: Coupled numerical–analytical approach to landscape evolution modeling, Earth Surf. Proc. Land., 39, 522–545, doi:10.1002/esp.3514, 2014.

Granjeon, D. and Joseph, P.: Concepts and Applications of a 3-D Multiple Lithology, Diffusive Model in Stratigraphic Modeling, in: Numerical Experiments in Stratigraphy Recent Advances in Stratigraphic and Sedimentologic Computer Simulations, SEPM Special Publications No. 62, SEPM, Tulsa, OK, USA, 197–210, 1999.

Harel, M. A., Mudd, S. M., and Attal, M.: Global analysis of the stream power law parameters based on worldwide 10Be denudation rates, Geomorphology, 268, 184–196, doi:10.1016/j.geomorph.2016.05.035, 2016.

Harlow, F. H. and Welch, J. E.: Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface, Phys. Fluids, 8, 2182–2189, 1965.

Hobley, D. E. J., Sinclair, H. D., Mudd, S. M., and Cowie, P. A.: Field calibration of sediment flux dependent river incision, J. Geophys. Res., 116, F04017, doi:10.1029/2010JF001935, 2011.

Horritt, M. S. and Bates, P. D.: Evaluation of 1D and 2D numerical models for predicting river flood inundation, J. Hydrol., 268, 87–99, doi:10.1016/S0022-1694(02)00121-X, 2002.

Howard, A. D.: A detachment-limited model of drainage basin evolution, Water Resour. Res., 30, 2261–2285, 1994.

Howard, A. D.: Simulating the development of Martian highland landscapes through the interaction of impact cratering, fluvial erosion, and variable hydrologic forcing, Geomorphology, 91, 332–363, doi:10.1016/j.geomorph.2007.04.017, 2007.

Hunter, N. M., Horritt, M. S., Bates, P. D., Wilson, M. D., and Werner, M. G. F.: An adaptive time step solution for raster-based storage cell modelling of floodplain inundation, Adv. Water Resour., 28, 975–991, 2005.

Hutton, E. W. H. and Syvitski, J. P. M.: Sedflux 2.0: An advanced process-response model that generates three-dimensional stratigraphy, Comput. Geosci., 34, 1319–1337, doi:10.1016/j.cageo.2008.02.013, 2008.

Hutton, E. W. H., Piper, M. D., Peckham, S. D., Overeem, I., Kettner, A. J., and Syvitski, J. P. M.: Building Sustainable Software – The CSDMS Approach, The Second Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2),

New Orleans, LA, USA, 16 November 2014, arxiv:1407.4106v2 [cs.SE], 2014.

Istanbulluoglu, E., Wang, T., and Wedin, D. A.: Evaluation of ecohydrologic model parsimony at local and regional scales in a semiarid grassland ecosystem, Ecohydrology, 5, 121–142, doi:10.1002/eco.211, 2012.

Itasca: FLAC: fast Lagrangian analysis of continua, Version 4, Itasca Consulting Group Inc., Minneapolis, USA, 2000.

Jenson, S. K. and Domingue, J. O.: Extracting Topographic Structure from Digital Elevation Data for Geographic Information System Analysis, Photogramm. Eng. Rem. S., 54, 1593–1600, 1988.

Julien, P. Y., Saghafian, B., and Ogden, F. L.: Raster-based hydrologic modeling of spatially-varied surface runoff, J. Am. Water Resour. As., 31, 523–536, doi:10.1111/j.1752-1688.1995.tb04039.x, 1995.

Katz, D. S., Choi, S.-C. T., Wilkins-Diehr, N., Hong, N. C., Venters, C. C., Howison, J., Seinstra, F., Jones, M., Cranston, K. A., Clune, T. L., De Val-Borro, M., and Littauer, R.: Report on the Second Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2), Journal of Open Research Software, 4, e7, doi:10.5334/jors.85, 2015.

Kelfoun, K., Samaniego, P., Palacios, P., and Barba, D.: Testing the suitability of frictional behaviour for pyroclastic flow simulation by comparison with a well-constrained eruption at Tungurahua volcano (Ecuador), B. Volcanol., 71, 1057–1075, doi:10.1007/s00445-009-0286-6, 2009.

Kessler, M. A., Anderson, R. S., and Stock, G. M.: Modeling topographic and climatic control of east-west asymmetry in Sierra Nevada glacier length during the Last Glacial Maximum, J. Geophys. Res, 111, F02002, doi:10.1029/2005JF000365, 2006.

Lague, D.: The stream power river incision model: evidence, theory and beyond, Earth Surf. Proc. Land., 39, 38–61, doi:10.1002/esp.3462, 2014.

Laio, F., Porporato, A., Ridolfi, L., and Rodriguez-Iturbe, I.: Plants in water-controlled ecosystems: active role in hydrologic processes and response to water stress II. Probabilistic soil moisture dynamics, Adv. Water Resour., 24, 707–723, doi:10.1016/S0309-1708(01)00005-7, 2001.

Lambeck, K.: Geophysical Geodesy, The Slow Deformations of the Earth, Clarendon Press, Oxford, UK, 718 pp., 1988.

Mitas, L. and Mitasova, H.: Distributed soil erosion simulation for effective erosion prevention, Water Resour. Res., 34, 505–516, doi:10.1029/97WR03347, 1998.

Narteau, C., Le Mouël, J. L., Poirier, J. P., Sepúlveda, E., and Shnirman, M.: On a small-scale roughness of the core–mantle boundary, Earth Planet. Sc. Lett., 191, 49–60, doi:10.1016/S0012-821X(01)00401-0, 2001.

Narteau, C., Zhang, D., Rozier, O., and Claudin, P.: Setting the length and time scales of a cellular automaton dune model from the analysis of superimposed bed forms, J. Geophys. Res.-Earth, 114, F03006, doi:10.1029/2008JF001127, 2009.

NCALM: Raleigh Peak, Colorado: May 2010, CO10_Tucker (Data set), doi:10.5069/G9TM782F, 2010.

NSF: A vision and strategy for software for science engineering and education, available at: https://www.nsf.gov/pubs/2012/nsf12113/nsf12113.pdf (last access: 24 November 2016), 2012.

Overeem, I., Berlin, M. M., and Syvitski, J. P. M.: Strategies for integrated modeling: The community surface dynamics modeling system example, Environ. Modell. Softw., 39, 314–321, doi:10.1016/j.envsoft.2012.01.012, 2013.

Peckham, S. D.: The CSDMS Standard Names: Cross-Domain Naming Conventions for Describing Process Models, Data Sets and Their Associated Variables, in: Proceedings of the 7th International Congress on Environmental Modelling and Software, 15–19 June 2014, San Diego, California, USA, edited by: Ames, D. P., Quinn, N. W. T., Rizzoli, A. E., ISBN: 978-88-9035-744-2, 2014.

Peckham, S. D., Hutton, E. W. H., and Norris, B.: A component-based approach to integrated modeling in the geosciences: The design of CSDMS, Comput. Geosci., 53, 3–12, doi:10.1016/j.cageo.2012.04.002, 2013.

Perron, J. T.: Numerical methods for nonlinear hillslope transport laws, J. Geophys. Res, 116, F02021, doi:10.1029/2010JF001801, 2011.

Perron, J. T. and Royden, L.: An integral approach to bedrock river profile analysis, Earth Surf. Proc. Land., 38, 570–576, doi:10.1002/esp.3302, 2012.

Piper, M., Hutton, E. W. H., Overeem, I., and Syvitski, J. P.: WMT: The CSDMS Web Modelling Tool, 2015 Fall Meeting, AGU, San Francisco, CA, USA, 14–18 December 2015, IN13B–1841, 2015.

Polakow, D. A. and Dunne, T. T.: Modelling fire-return interval T: stochasticity and censoring in the two-parameter Weibull model, Ecol. Model., 121, 79–102, 1999.

Prechelt, L.: An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl for a search/string-processing program, Technical Report 2000-5, University of Karlsruhe, Karlsruhe, Germany, 34 pp., 2000.

Rengers, F. K., McGuire, L. A., Kean, J. W., Staley, D. M., and Hobley, D.: Model simulations of flood and debris flow timing in steep catchments after wildfire, Water Resour. Res., 52, 6041–6061, doi:10.1002/2015WR018176, 2016.

Slingerland, R. L. and Kump, L.: Mathematical Modeling of Earth's Dynamical Systems, Princeton University Press, Princeton, NJ, USA, 231 pp., 2011.

Slingerland, R. L., Harbaugh, J. W., and Furlong, K.: Simulating Clastic Sedimentary Basins: Physical Fundamentals and Computer Programs for Creating Dynamic Systems, Prentice-Hall, Englewood Cliffs, NJ, USA, 220 pp., 1994.

Stewart, C. A., Almes, G. T., and Wheeler, B. C. (Eds.): Cyberinfrastructure Software Sustainability and Reusability: Report from an NSF-funded workshop, Indiana University, Bloomington, IN, USA, available at: http://hdl.handle.net/2022/6701 (last access: 24 November 2016), 2010.

Tucker, G. E. and Bras, R. L.: A stochastic approach to modeling the role of rainfall variability in drainage basin evolution, Water Resour. Res., 36, 1953–1964, 2000.

Tucker, G. E. and Hancock, G. S.: Modelling landscape evolution, Earth Surf. Proc. Land., 35, 28–50, doi:10.1002/esp.1952, 2010.

Tucker, G. E. and Whipple, K. X.: Topographic outcomes predicted by stream erosion models: Sensitivity analysis and intermodel comparison, J. Geophys. Res, 107, 2179, doi:10.1029/2001JB000162, 2002.

Tucker, G. E., Lancaster, S. T., Gasparini, N. M., Bras, R. L., and Rybarczyk, S. M.: An object-oriented framework for distributed hydrologic and geomorphic modeling using triangulated irregular networks, Comput. Geosci., 27, 959–973, 2001a.

Tucker, G. E., Lancaster, S. T., Gasparini, N. M., and Bras, R. L.: The Channel-Hillslope Integrated Landscape Development Model (CHILD), in: Landscape Erosion and Evolution Modeling, Springer US, Boston, MA, USA, 349–388, 2001b.

Tucker, G. E., Hobley, D. E. J., Hutton, E., Gasparini, N. M., Istanbulluoglu, E., Adams, J. M., and Nudurupati, S. S.: CellLab-CTS 2015: continuous-time stochastic cellular automaton modeling using Landlab, Geosci. Model Dev., 9, 823–839, doi:10.5194/gmd-9-823-2016, 2016.

van Rossum, G. and Drake, F. L.: Python reference manual, available at: http://www.python.org (last access: 24 November 2016), 2001.

Whipple, K. X. and Tucker, G. E.: Dynamics of the stream-power river incision model: Implications for height limits of mountain ranges, landscape response timescales and research needs, J. Geophys. Res, 104, 17661–17674, 1999.

Wickert, A. D.: Open-source modular solutions for flexural isostasy: gFlex v1.0, Geosci. Model Dev., 9, 997–1017, doi:10.5194/gmd-9-997-2016, 2016.

Willgoose, G., Bras, R. L., and Rodriguez-Iturbe, I.: A coupled channel network growth and hillslope evolution model: 1. Theory, Water Resour. Res., 27, 1671–1684, doi:10.1029/91WR00935, 1991a.

Willgoose, G., Bras, R. L., and Rodriguez-Iturbe, I.: A coupled channel network growth and hillslope evolution model: 2. Nondimensionalization and applications, Water Resour. Res., 27, 1685–1696, doi:10.1029/91WR00936, 1991b.

Wobus, C. W., Whipple, K. X., Kirby, E., Snyder, N. P., Johnson, J., Spyropolou, K., Crosby, B. T., and Sheenan, D.: Tectonics from topography: Procedures, promise, and pitfalls, in: Tectonics, Climate, and Landscape Evolution, edited by: Willett, S. D., Hovius, N., Brandon, M. T., and Fisher, D., Geological Society of America Special Paper 398, Geological Society of America, Boulder, CO, USA, 55–74, 2006.

Zhou, X., Istanbulluoglu, E., and Vivoni, E. R.: Modeling the ecohydrological role of aspect-controlled radiation on tree-grass-shrub coexistence in a semiarid climate, Water Resour. Res., 49, 2872–2895, doi:10.1002/wrcr.20259, 2013.