The Landlab LandslideProbability Component User Manual

Ronda Strauch, Erkan Istanbulluoglu, Sai Siddhartha Nudurupati
January 2018

1. Background on LandslideProbability Component

The Landlab LandslideProbability component implements the infinite slope stability model using a Monte Carlo simulation to predict the probability of shallow landslide initiation based on local slope, specific catchment area, climatological triggers, and soil and vegetation parameters on Landlab's RasterModelGrid. A for loop inside the component executes the Monte Carlo simulation at each node of the RasterModelGrid. The LandslideProbability component is executed by a user-written driver script that parametrizes, instantiates, runs, and plots data and results. This User Manual describes the LandslideProbability component and how to parameterize, instantiate, run, and plot data and results described in Strauch et. al., (2018) and using "Regional landslide hazard using Landlab - NOCA Observatory" containing two example model drivers available on https://www.hydroshare.org/. This document is a supplement to the manuscript in *Earth Surface Dynamics.*

The component is based on Mohr-Coulomb failure law using the infinite slope stability model that predicts the ratio of stabilizing forces due to friction and cohesion, mediated by pore-water pressure to destabilizing forces due to gravity, implemented on a failure plane parallel to the land surface, coupled with a topography-driven steady-state subsurface flow model (Montgomery and Dietrich 1994; Pack et al. 1998). The component executes a Monte Carlo solution of the coupled model by generating model forcing and parameters: daily recharge to subsurface flow that reflects local pore-water pressure through regulating the water table, soil internal friction angle, combined root and soil cohesion, soil transmissivity, and soil depth from assumed probability density functions. Local slope and specific catchment area are generated from a digital elevation model (DEM). Several options are offered for recharge. This component expands the capabilities of Landlab by providing a probabilistic shallow landsliding model that can also be used to develop watershed sediment yield models.

Note: Currently, the LandslideProbability component can only operate on a structured grid; therefore, all the references to the grid below are referring to the Landlab RasterModelGrid object.

Prerequisites: A working knowledge of the Python programming language (any version) and familiarity with the Python libraries NumPy and Matplotlib are beneficial. Also, a basic understanding of the Landlab modeling framework (Hobley et al., 2017) and the RasterModelGrid module is recommended.

2. Model Description

2.1. Model Parameters (inputs)

Spatial model parameters listed here are primarily pre-processed by the model user in a geographic information system, such as Esri's ArcGIS rasters converted to ASCII format. Users may acquire and generate these parameters from a variety of sources. Details on where to find readily available data and how to process these data into parameter rasters are provided in Strauch et al., (2018). For example, root cohesion can be generated from reclassifying land use/land cover (LULC) rasters from USGS National Land Cover Data (USGS 2014; Jin 2013) based on reference to a lookup table that specifies the root cohesion for different LULC types, Table 1 in Strauch et al., (2018). The reclassified raster can then be converted to an ASCII file for import into Landlab, defined with number of rows and columns and cell size.

Parameters are assigned as *fields* to nodes in the RasterModelGrid. Most Landlab names follow the naming conventions of Community of Surface Dynamics Modeling System (CSDMS) (Peckham, 2014). Fields are accessed using Python's dictionary data structure where the field name, such as '*topographic__slope*' is a string keyword assigned and used to access the values array. Methods to import these parameters into Landlab are detailed below in section **3.3 Step 3**.

- **topographic__slope**: [-] – local elevation gradient slope of surfaces as represented by the tangent of hillslope slope angle.
- **topographic__specific_contributing_area**: [m] – specific contributing area calculated as upslope drainage area/unit contour length (e.g., grid cell width) using the multiple flow direction D-infinity approach of TauDEM. (http://hydrology.usu.edu/taudem/taudem5/index.html)
- **soil__internal_friction_angle**: [deg] Critical angle just before failure due to friction between particles.
- **soil__maximum_total_cohesion**: [Pa] Maximum of combined root and soil cohesion.
- **soil__minimum_total_cohesion**: [Pa] Minimum of combined root and soil cohesion.
- **soil__mode_total_cohesion**: [Pa] Mode of combined root and soil cohesion.
- **soil__thickness**: [m] Depth to restrictive layer (e.g., bedrock, low permeable layer).
- **soil__transmissivity**: [m^2/day] depth integrated saturated hydraulic conductivity; required input if conductivity is NOT provided.
- **soil__hydraulic_conductivity**: [m/day] rate of water transmitted through soil; required input if transmissivity is NOT provided to calculate transmissivity with soil depth
- **soil__density**: [kg/m^3] Wet bulk density of soil, which is set uniformly across model domain at 2000 kg/m^3 in the current application, but can be spatially distributed by user as a field on the RasterModelGrid.

The hydrologic driver of shallow landslide initiation in this model is a daily annual maximum rate of recharge [mm/d]. Given that users may acquire or identify recharge from various resources or approaches, the LandslideProbability component provides four options to parameterize recharge. These options control the amount of data used within the model and the distribution of the data, which is designed to represent uncertainty in recharge rate. The options are identified through a *'distribution'* specified by the user as listed below. The distribution and parameters are passed from the drive to the component. Additional details on setting up the recharge are provided in **Determine groundwater recharge** inputs in section 3.5.

- **groundwater__recharge_distribution**: [mm/d] distribution to use for representing recharge specified as one of four options:
  - **'uniform'** – user specifies parameters groundwater__recharge_min_value and **groundwater__recharge_max_value** used to generate uniformly distributed recharge applied uniformly over the RasterModelGrid.
  - **'lognormal'** – user specifies parameters **groundwater__recharge_mean** and **groundwater__recharge_standard_deviation** used to generate lognormally distributed recharge applied uniformly over the RasterModelGrid.
  - **'lognormal_spatial'** – user specifies parameter arrays for **groundwater__recharge_mean** and **groundwater__recharge_standard_deviation** for each node of the RasterModelGrid used to generate spatially variable lognormally distributed recharge. We recommend providing parameters from routed recharge to better represent the spatial hydrology as arrays of: **groundwater__recharge_mean** and **groundwater__recharge_standard_deviation**.
  - **'data_driven_spatial'** – user specifies three dictionaries for **groudwater__recharge_HSD_inputs** in a list using the format of [HSD_dict, HSD_id_dict, fract_dict] (*in that order*). HSD is the *'Hydrology Source Domain'* that provides the recharge from a hydrologic model, which is processed to generate routed recharge at each RasterModelGrid node using a nonparametric method.

2.2. Model Variables (outputs)

Variables listed below are calculated by the component at RasterModelGrid locations in the model domain.

- **soil__mean_relative_wetness:** [-] mean of the ratio of depth of subsurface flow over a restrictive layer to the depth of soil layer over the restrictive layer. The value has a range from 0 (dry soil) to 1 (saturated soil).
- **soil__probability_of_saturation**: [-] ratio number of times relative wetness is ≥1 out of number of iterations user selected
- **landslide__probability_of_failure**: [-] ratio of the number of failures simulated (i.e., FS ≤ 1) divided by the number of iterations in the Monte Carlo simulation, which calculates deterministic FS values.

3. Basic Steps of a Landslide Model

Here we first list the basic steps to develop a Landlab Landslide hazard model, followed by their detailed explanations.

1. Import necessary libraries: Only LandslideProbability from landlab.io is required. However, optional libraries that may be useful include: numpy, read_esri_ascii, write_esri_ascii, matplotlib, cPickle, os, collections, and pandas, depending on the data processing and outputs desired by the user. The SourceTracking _Utility is also helpful for routing.

2. Define model domain: The model computational domain of the LandslideProbability component can only work on a RasterModelGrid instances as of Landlab version 1.0.0.

3. Load model input data and parameters: Several data and parameter fields are required to run the LandslideProbability component. These input parameters, described above, can be typically read in using *read_esri_ascii()* if they are prepared in ESRI's ArcGIS software, or can

consist of arrays of the same length of the number of nodes, and assigned to the RasterModelGrid.

4. Set boundary conditions: Often only a portion of the model domain is desired for analysis, such as watershed boundaries, jurisdictional boundaries, or areas above a certain topographic criterion such as elevation. Additionally, input parameters often have missing data that are excluded from analysis. These regions can be defined by setting *set_nodata_nodes_to_closed()* (Hobley et al. 2017). http://landlab.readthedocs.io/en/latest/landlab.grid.base.html#boundary-condition-control.

5. Set Monte Carlo iterations value: The user set the number of iterations they wish for the Monte Carlo simulation.

6. Determine groundwater recharge inputs: Flexibility is provided for the user to define the groundwater recharge used to calculate relative soil wetness. The user must determine the recharge values and or parameters as well as the desired parametric probability distributions they wish to use for data generation.

7. Initialize LandslideProbability component: The instance of the LandslideProbability class is declared and parameters are set by the user.

3.1. Step 1. Import necessary libraries

To build a landslide model using the Landlab LandslideProbability component, first the necessary Landlab components and utilities, as well as any necessary Python packages and customized utilities must be imported. Standard Python style dictates all import statements belong in the top of the driver file or script, after the module docstrings. An example of a landslide model driver begins as follows:

```
##############################################################################
'''
Landslide_driver.py

Purpose: To model shallow landslide initiation probability by gathering
input data, running LandslideProbability component, and visualizing/storing
data and outputs.

Inputs: Data is provided by the user and consists of elevation
from a DEM and derived topographic traits such as slope and contributing area.
User also supplies soil characteristics derived from a soil survey,
land cover, or other sources, including transmissivity, cohesion,
internal angle of friction, soil density, and soil thickness.
Groundwater recharge is derived from post-processed hydrological data
provided from the VIC hydrological model at 1/16 degree resolution.

Outputs: mean relative wetness, probability of saturation, and
probability of failure based on Monte Carlo simulation.

Authors: R.Strauch, E.Istanbulluoglu, and S.Nudurupati
Univerity of Washington
Created on Thu Aug 20, 2015
Last edit June 9, 2017
'''
##############################################################################
# %% Import libraries and utilities
## Landlab components
from landlab.components.landslides import LandslideProbability

## Landlab utilities
from landlab.io import read_esri_ascii  # OR from landlab import RasterModelGrid
from landlab.io import write_esri_ascii
from landlab.plot.imshow import imshow_node_grid #plotter function optional

## Additional Python packages or customized utilities
import numpy as np
import os
import matplotlib.pyplot as plt
import matplotlib as mpl
import cPickle as pickle
import pandas as pd
##############################################################################
```

To run the model for the application here, only the LandslideProbability component is needed. This will calculate relative wetness and probability of failure across the model domain, which is at a 30-m resolution in this application. The domain will be defined by the DEM (see Step 2). Other Landlab utilities used in this example are the plotting library *imshow_node_grid*, which is a utility that can plot a Landlab grid instance and data field in map view (see Section 4). Finally, additional Python packages and user defined utilities are imported. In this application, the numpy and matplotlib are dependencies of Landlab, meaning they are installed as part of the Landlab installation and thus, are likely already available with the user's Python distribution. The scientific computing library numpy is used for mathematical operations, and the matplotlib library is for plotting model output. The *os* library is useful for handling files and folders when processing data, such as hydrologic flux files in the HSD. The cPickle library helps handle dictionaries that provide lookup tables of identifiers and data used in the SourceTracking_Utility and data_driven_spatial recharge option. Pandas Python package provides fast, flexible, and expansive data structures particularly useful for process data associated with dates such as flux files.

3.2. Step 2. Define the model domain

As mentioned above, the LandslideProbability component was designed to work on a gridded

landscape by using the RasterModelGrid instance in Landlab. The RasterModelGrid is square and composed of nodes, which are points in *(x,y)* space in the implementation of the LandslideProbability component. The grid nodes consist of rows and columns numbered from 0 at the lower left corner of the grid and ending at the upper right corner of the grid, looping left to right in number as one moves up the rows from the bottom. Calculations in the LandslideProbability component are made at each grid cell as represented by the central node. There are two methods to implement a RasterModelGrid in Landlab. A grid can be created by reading in data from an ASCII file created in Esri ArcGIS or using the RasterModelGrid class directly. An example using a DEM to establish a grid instance follows the syntax:

(grid, z) = read_esri_ascii('elevation.txt', name='topographic__elevation')

The second approach can be accomplished with the following code:

grid = RasterModelGrid((number_of_node_rows, number_of_node columns), Δx)

The first method demonstrated below is reading a DEM with *read_esri_ascii()*.

```
# load in DEM from ArcGIS
(grid, z) = read_esri_ascii(data_folder+'/elevation.txt',
                            name='topographic__elevation')
grid.at_node.keys()     # loads DEM grid with elevation
```

In this application, the 'elevation.txt' represents a square domain of 3217 rows and 2185 columns that includes the North Cascades National Park Complex in Washington, U.S.A. The path to the file was previously defined in 'data_folder' variable and is dependent on the user's file system. The second command assigns the elevation data to nodes. This particular DEM was pit-filled using TauDEM. The component does not use elevation in the calculations; however, loading it established the full model domain and can be helpful in visualizations.

The second method to establish an elevation grid sets the RasterModelGrid instance manually:

```
grid = RasterModelGrid((number_of_node_rows, number_of_node_columns),dx)
z = user_defined_elevation_data   # array length of rows*columns
grid.at_nodes['topographic__elevaton'] = z
```

This application assumes that model user knows and defines the number of grid rows, number of grid columns, the grid resolution *(dx)* and elevation data at each node. The *user_defined_elevation_data* must be the same length as the number of nodes in the RasterModelGrid, which can be found by using the command: *grid.number_of_nodes()* after the first line. The first row must match the lower left position and end with the upper right position of the RasterModelGrid.

3.3. Step 3. Load model input data and parameters

Ten model inputs, including topographic attributes and hydrologic forcing variables as well as soils and vegetation related parameters, are required by the LandslideProbability component. Nine of these are assigned to the RasterModelGrid and thus, provide the ability to spatially vary the parameters across the model domain. These are typically pre-processed in ArcGIS and read in similar to the DEM using *read_esri_ascii('parameter')* using the example code block below for slope. Note that in the development of specific catchment area, a larger domain is used to capture the regional boundary of the watersheds that contribute flows into the boundaries of the study area if it does not follow watershed boundaries.

```
# load slope from esri ascii file
(grid1, slope) = read_esri_ascii(data_folder+'/slope_tan.txt')
grid.add_field('node', 'topographic__slope', slope)
```

If the user has an array of values, they can assign the values to the RasterModelGrid similar to the alternative method described above for establishing an elevation field. The user may also want to set a parameter uniformly across the model domain, such as soil density using the syntax below.

```
# set soil density value and assign to all nodes
grid['node']['soil__density'] = 2000*np.ones(grid.number_of_nodes)
```

This approach can also be used to set **soil__minimum_total_cohesion** and **soil__maximum_total_cohesion** if only a **soil__mode_total_cohesion** is provided and the users wished to set the minimum and maximum as a fraction of the mode.

3.4. Step 4. Set boundary conditions

Often there are gaps in data or areas of a model domain that a user wished to exclude from analysis. These areas are handled through the establishment of boundary conditions on the grid nodes. Node boundary status can be set to *boundary* or *core*; boundary nodes can be further defined as *open* or *closed*. Grid perimeter nodes are open and interior nodes are core nodes by default. Interior nodes can be set to closed boundary conditions by the user for nodes that have no data, commonly represented by value *-9999*. The core nodes are the nodes where the component operates and calculates probability of failure. Landlab has several methods to set and update boundary conditions at *node* elements. Data imported for model parameters in **Step 3** often have missing values usually defined as *-9999*. These can be set to closed nodes using the following command:

```
# set boundary conditions closed where no data
grid.set_nodata_nodes_to_closed(grid.at_node['soil__transmissivity'], -9999)
```

This is also an approach for using a *mask* set up in ArcGIS where locations to include in the analysis have node values set to 1 and areas to exclude set to -9999. To test a subarea before analyzing an entire modeling domain, the mask can be set up for a subarea and then commented out to run the model for the entire domain. Another way to test an area is to establish a subset of *core_nodes* and run the component only for this subset.

3.5. Step 5. Set Monte Carlo iterations value

The number of Monte Carlo iterations, *n*, is a parameter defined by the user. The simulated failure probability becomes more stable (i.e., consistent) with increasing *n* at the expense of computer time (Cho 2007; El-Ramly et al. 2002). Hammond et al. (1992) recommends at least 1,000 iterations for the Monte Carlo simulation. This is set by the user using the following syntax:

```
# number_of_iterations for Monte Carlo simulation
number_of_iterations = 3000
```

3.6. Step 6. Determine groundwater recharge inputs

Groundwater recharge is the hydrologic driving force that increases pore-water pressure within the soil layer, reducing the effective normal stress. The LandslideProbability component

provides several options for the user to provide recharge input. Within the driver, the user provides information on recharge, which is used to instantiate the component. This information includes a *distribution* and the *parameters* (Table 1). The distribution can be: 'uniform', 'lognormal', 'lognormal_spatial', or 'data_driven_spatial'. The parameters represent scalars, arrays, or lists, depending on the distribution specified. Options are shown in Table 1 and detailed descriptions with code snippets for each distribution type follow the table.

Table 1 – Recharge inputs accepted by LandslideProbability component

Distribution	Parameters	Input / Comments
'uniform'	• minimum • maximum	Minimum and maximum recharge is provided to generate a uniform sampling distribution over the model domain
'lognormal'	• mean • standard deviation	Mean and standard deviation of recharge is provided to calculate mu and sigma for generating a lognormal sampling distribution applied uniformly over the model domain.
'lognormal_spatial'	• [mean] • [standard deviation]	Mean and standard deviation arrays are provided for each node and used to calculate mu and sigma for a lognormal sampling distribution for each node such that recharge is spatially distributed. Each of these arrays needs to be the same length as RasterModelGrid nodes. Only core node values will be used in calculations.
'data_driven_spatial'	• {HSD_Re} • {HSD_ID} • {fractions}	HSD_Re dictionary is a unique array of recharge provided as arrays (*'values'*) for each of the Hydrologic Source Domain (HSD) (*'keys'*). HSD_ID dictionary has the model domain node ID as *'keys'* and HSD IDs in a list as *'values'*. The fractions dictionary assigns to each node ID as *'key'* a list of the fractions of each HSD draining to the node as *'values'*.

- **Recharge: 'uniform'**

In this option, the user specifies distribution as 'uniform' and defines the minimum and maximum recharge [mm/d] parameters to be used in the component to generate a uniform distribution that is applied uniformly over the model domain. This is the default option used in the component. A code snippet below demonstrates how to provide this information in the driver.

```
#Option - uniform distribution
distribution = 'uniform'
Remin_value = 20.
Remax_value = 120.
```

- **Recharge: 'lognormal**

Instead of providing minimum and maximum recharge values, in this option, the user specifies the distribution as 'lognormal' and provides the mean and standard deviation parameters of the recharge to be used to calculate mu and sigma for generating a lognormal distribution. This

distribution is then applied uniformly over the model domain. A code snippet below demonstrates how to provide this information to the driver.

```
#Option 2 - Lognormal
distribution2 = 'lognormal_uniform'
Remean = 30.
Restandard_deviation = 0.25
```

- **Recharge: 'lognormal_spatial'**

This option allows for spatial representation of recharge. Similar to previous option, this option applies a lognormal distribution to represent recharge from arrays of mean and standard deviation representing recharge at each node. Note that the mean and standard deviation is from the native form of recharge acquired by the user and not a log transformation of the recharge. Additionally, this array must be as long as the *length of all nodes* in the model domain, such that the first mean is for node ID 0. A code snippet below demonstrates how to provide this information to the driver using a random generation of integers for mean recharge from NumPy.

```
#Option - Lognormal-Spatial
distribution = 'lognormal_spatial'
Remean = np.random.randint(20,120,grid_size)
Restandard_deviation = np.random.rand(grid_size)
```

- **Recharge: 'data_driven_spatial'**

This recharge option provides an even more elaborate handling of recharge data. In this option, the user specifies the distribution as 'data_driven_spatial' and three *Python dictionaries* are provided or generated in the driver using the STA utility to make spatially-distributed arrays of recharge at each node while looping through nodes in the component. An example of the driver description using recharge from a HSD that is spatially variable is provided below:

```
#Option - Fully distributed
distribution = 'data_driven_spatial'
HSD_inputs = [HSD_dict,HSD_id_dict, fract_dict]
```

In the application of the model in Strauch et al. (2018), we used spatial recharge rates from the Variable Infiltration Capacity (VIC) macroscale hydrologic model (Liang et al., 1994) simulation run at a daily time step for a historical period (water years 1916 to 2006) on a 1/16th° grid resolution (Hamlet et al., 2013). In our application of the model, we focus on landslide risk over the contemporary climate. Assuming landslides would most likely initiate during high water-input (rain and snowmelt) events, we use only the maximum modeled recharge rate in each water year at each VIC grid cell. Using a single recharge value for a year in a Monte Carlo simulation gives an annual probability of shallow landsliding. The native resolution of the VIC hydrology, generically named as *'Hydrology Source Domain'* (HSD) is coarser than the resolution of the landslide model. A separate 'source tracking' algorithm (STA) written as Landlab utility (SourceTracking_Utility) is run in the model driver, or in a separate Python script, to map the HSD data to the finer model domain resolution. This algorithm tracks the HSD grid IDs draining to each node of the RasterModelGrid and calculates the fraction of the upstream drainage area contribution from each HSD. This tracking is packaged into two Python dictionaries with both *'keys'* as the RasterModelGrid node ID, and the *'values'* as lists of the HSD grid IDs and the corresponding fraction of each draining HSD grid ID, respectively. These two dictionaries along with a dictionary of the HSD grid ID (*key*) and a numpy array of recharge (*value*) are passed onto the component. The component uses the recharge data from each contributing HSD IDs and generates recharge using a non-parametric method while looping through each node of the RasterModelGrid. Using the other two dictionaries, the recharge at a node is weighted by the

upstream contributing fraction from different HSD grids and a 'routed' recharge is calculated for each node (Fig. 1).

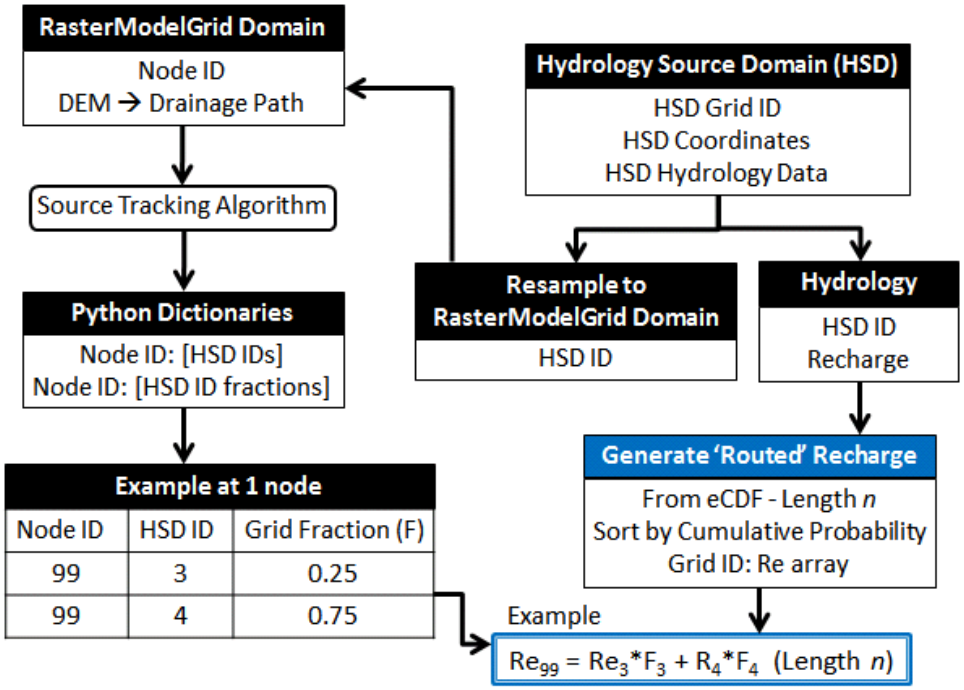


Fig. 1 – Workflow for source routing of recharge when downscaling from macro-scale hydrology to RasterModelGrid. Blue section at bottom right is work being performed inside the LandslideProbability component.

The STA is Landlab utility that traverses a RasterModelGrid, and records all upstream contributing core nodes for each core node. Unique HSD ids that represent the upstream core nodes, for each core node, will be returned as a Python dictionary. For more detailed explanation of the algorithm, refer to STA utility documentation on Landlab.

3.7. Step 7. Instantiate and run LandslideProbability component

Most Landlab components are structured as a Python class. These classes are imported (as seen in **Step 1**) and then the user must create an instance of the class as shown below. When the instance of the class is created (*e.g., LS_prob1*), parameters are passed as arguments to the class. All Landlab components take a grid as their first argument. The first argument passes on the grid as named by the user and all subsequent arguments are additional parameters (see section 2.1.) used to control the model behavior. The elegance of Landlab is that all the fields are attached to the grid object and are automatically passed to the component class at once through the 'grid' object. The recharge parameterization and number of iterations are also passed as arguments. Once the component has been instantiated, the component is run to calculate the probability of failure by calling the component's method *calculate_landslide_probability()* in a for loop that performs the calculations at each node. Example syntax are provided below for all four recharge options:

```
# Instantiate and Run the 'LandslideProbability' component
#Uniform
LS_prob1 = LandslideProbability(grid,number_of_iterations=n,
    groudwater__recharge_distribution=distribution1,
    groundwater__recharge_min_value=Remin_value,
    groundwater__recharge_max_value=Remax_value)
LS_prob1.calculate_landslide_probability()
#lognormal_uniform
LS_prob2 = LandslideProbability(grid,number_of_iterations=n,
    groudwater__recharge_distribution=distribution2,
    groundwater__recharge_mean=Remean,
    groundwater__recharge_standard_deviation=Restandard_deviation)
LS_prob2.calculate_landslide_probability()
#lognormal_spatial
LS_prob3 = LandslideProbability(grid,number_of_iterations=n,
    groudwater__recharge_distribution=distribution3,
    groundwater__recharge_mean=Remean3,
    groundwater__recharge_standard_deviation=Restandard_deviation3)
LS_prob3.calculate_landslide_probability()
#HSD
LS_prob4 = LandslideProbability(grid,number_of_iterations=n,
    groudwater__recharge_distribution=distribution4,
    groudwater__recharge_HSD_inputs=HSD_inputs)
LS_prob4.calculate_landslide_probability()
```

Probability of failure is calculated at each RasterModelGrid node by solving the infinite slope stability equation for the factor-of-safety index (FS) in a Monte Carlo simulation approach. The equation is solved deterministically by sampling from parameter distributions and calculating the FS for each iteration at each RasterModelGrid node. The probability is determined by the number of iterations where FS $\leq$ 1.0, count(FS $\leq$ 1.0)/n(), where n is sample size. The resulting output is a spatially distributed probability of failure over the modeled domain.

This methods within the LandslideProbability class (i.e., the component and not the driver) are described below with code snippets:
calculate_landslide_probability()

- Create arrays for storing output
- Execute a *for* loop that loops through each node, calculate FS, mean relative wetness, probability of failure, and a histogram of FS values for the last node to check distribution being created.
- Assign mean relative wetness and probability of failure as fields in the RasterModelGrid for later plotting.

```
def calculate_landslide_probability(self, **kwds):
    # Create arrays for data with -9999 as default to store output
    self.mean_Relative_Wetness = -9999*np.ones(self.grid.number_of_nodes,
                                               dtype='float')
    self.prob_fail = -9999*np.ones(
        self.grid.number_of_nodes, dtype='float')
    self.landslide__factor_of_safety_histogram = -9999*np.ones(
        [self.grid.number_of_nodes, self.n], dtype='float')
    # Run factor of safety Monte Carlo for all core nodes in domain
    # i refers to each core node id
    for i in self.grid.core_nodes:
        self.calculate_factor_of_safety(i)
        # Populate storage arrays with calculated values
        self.mean_Relative_Wetness[i] = self.soil__mean_relative_wetness
        self.prob_fail[i] = self.landslide__probability_of_failure
        self.landslide__factor_of_safety_histogram[i] = (
            self.FS_distribution)
        # stores FS values from last loop (node)
    # replace unrealistic values in arrays
    self.mean_Relative_Wetness[
        self.mean_Relative_Wetness < 0.] = 0.  # so can't be negative
    self.prob_fail[self.prob_fail < 0.] = 0.   # can't be negative
    # assign output fields to nodes
    self.grid['node']['soil__mean_relative_wetness'] = (
        self.mean_Relative_Wetness)
    self.grid['node']['landslide__probability_of_failure'] = self.prob_fail
```

calculate_factor_of_safety()

- This method is called within the previous *for* loop above. This method generates distributions of the parameters for the individual node and then calculates the FS for the number of iterations (*n*) specified.
- In the process, this method calculates the relative wetness *n* times and stores the mean. The mean FS and probability of failure are also calculated.
- The calculated variables and their summary variables are stored in arrays.

```
def calculate_factor_of_safety(self, i):
    # load parameter for node - EXAMPLE
    self.theta = self.grid['node']['topographic__slope'][i]
    # generate distributions to sample from, for some input parameters
    # currently triangle distribution using mode, min, & max
    # Tansmissivity (T) - EXAMPLE
    self.Tmode = self.grid['node']['soil__transmissivity'][i]
    Tmin = self.Tmode-(0.3*self.Tmode)
    Tmax = self.Tmode+(0.1*self.Tmode)
    self.T = np.random.triangular(Tmin, self.Tmode, Tmax, size=self.n)
    # recharge distribution based on distribution type
    if self.groundwater__recharge_distribution == 'data_driven_spatial':
        self._calculate_HSD_recharge(i)
        self.Re /= 1000.0  # mm->m
    elif self.groundwater__recharge_distribution == 'lognormal_spatial':
        mu_lognormal = np.log((self.recharge_mean[i]**2)/np.sqrt(
            self.recharge_stdev[i]**2 + self.recharge_mean[i]**2))
        sigma_lognormal = np.sqrt(np.log((self.recharge_stdev[i]**2)/(
            self.recharge_mean[i]**2)+1))
        self.Re = np.random.lognormal(mean=mu_lognormal,
                                      sigma=sigma_lognormal,
                                      size=self.n)
```

```
# calculate Factor of Safety for n number of times
# calculate components of FS equation
self.C_dim = self.C/(self.hs*self.rho*self.g)  # dimensionless cohesion
self.Rel_wetness = ((self.Re)/self.T)*(self.a/np.sin(
    np.arctan(self.theta)))                          # relative wetness
np.place(self.Rel_wetness, self.Rel_wetness > 1, 1.0)
# maximum Rel_wetness = 1.0
self.soil__mean_relative_wetness = np.mean(self.Rel_wetness)
self.Y = np.tan(np.radians(self.phi))*(1 - (self.Rel_wetness*0.5))
# convert from degrees; 0.5 = water to soil density ratio
# calculate Factor-of-safety
self.FS = (self.C_dim/np.sin(np.arctan(self.theta))) + (
    np.cos(np.arctan(self.theta)) *
    (self.Y/np.sin(np.arctan(self.theta))))
self.FS_store = np.array(self.FS)        # array of factor of safety
self.FS_distribution = self.FS_store
count = 0
for val in self.FS:                      # find how many FS values <= 1
    if val <= 1.0:
        count = count + 1
self.FS_L1 = float(count)     # number with unstable FS values (<=1)
# probability: No. unstable values/total No. of values (n)
self.landslide__probability_of_failure = self.FS_L1/self.n
```

4. Exporting, Plotting and Visualization

4.1. Exporting

There are several ways to assess results. Some users may elect to extract results into a dataframe and write the dataframe to a .csv file. This file can then be read and analyzed externally in other Python scripts, 'R', Matlab, etc. Landlab also provides a function to write field assigned to the RasterModelGrid into ASCII files that can be converted to rasters in a GIS program. Code examples for these options are shown below.

```
# %% Export data_____________________________________________________________
# extract data into a .csv file
data_extracted = {'prob_fail': np.array(
                  grid.at_node['landslide__probability_of_failure'][grid.core_nodes]),
                  'slope_tan': np.array(
                  grid.at_node['topographic__slope'][grid.core_nodes]),
                  'wetness': np.array(
                  grid.at_node['soil__mean_relative_wetness'][grid.core_nodes])}
headers = ['prob_fail', 'slope_tan','wetness']
df = pd.DataFrame(data_extracted, index=core_nodes, columns=(headers))
df.to_csv('LS_results.csv')  # creates a .csv file with headers
# extract data into a axcii file able to be converted to a raster in ArcGIS
write_esri_ascii('prob_fail.txt',grid,names='landslide__probability_of_failure')
```

4.2. Plotting and Visualizing

After the model is run, both inputs and outputs can be plotted using the matplotlib library. Below is an example for plotting a parameter (slope) assigned to the RasterModelGrid. For more customization options, the matplotlib.pyplot documentation is recommended: http://matplotlib.org/api/pyplot_api.html.

```
# %% Figures & Plots_____________________________________________________
# fields on grid
plt.clf
plt.figure('Slope (tan theta)')
imshow_node_grid(grid, 'topographic__slope', cmap='pink',
                 grid_units=('coordinates', 'coordinates'), vmax=2.,
                 shrink=0.75, var_name='Slope', var_units='m/m')
plt.savefig(outdir+'Slope.png', dpi=300)
```

The output for the above code is shown below. This is for a portion of North Cascades National Park Complex in northern Washington (Fig. 2). Black areas are excluded portions of the domain.

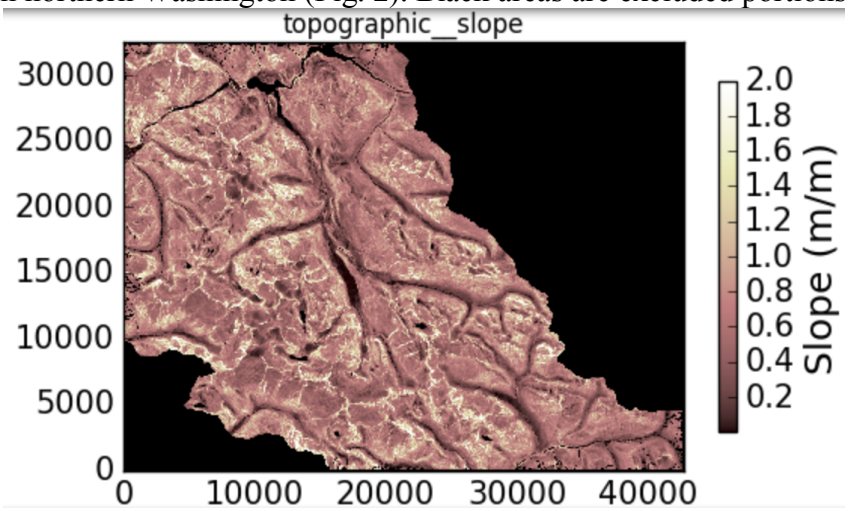
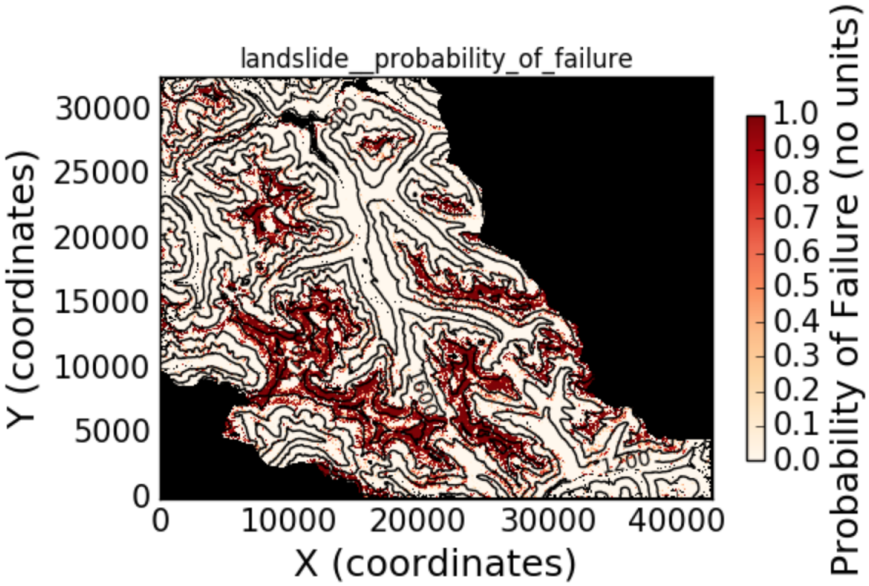


Fig. 2 – Slope (dimensionless) used as an input in the LandslideProbability component.

Adding contours helps interpret the inputs and results. These can be created using the *z* array containing elevation generated in **Step 2** above when creating the RasterModelGrid from a DEM. The example below is the probability of failure estimated the same area above with Monte Carlo simulation (Fig. 3).



```
plt.figure('Probability of Failure')
imshow_node_grid(grid, 'landslide__probability_of_failure', cmap='OrRd',
                 grid_units=('coordinates', 'coordinates'), shrink=0.75,
                 var_name='Probability of Failure', var_units='no units')
elev = grid.node_vector_to_raster(z)
# control contour extent with labels
cs = pylab.contour(elev, extent=[0,42500, 0,32500], hold='on', colors='black')
manual_locals = [(11000,11000, (14000,30000), (20000,8000), (36000,3000)]
pylab.clabel(cs, inline=True, fmt='%1i', fontsize=10, manual=manual_locals)
plt.savefig(outdir+'Prob_failure.png', dpi=300)
```

Fig. 3 – Probability of failure estimated for a portion of the park from the LandslideProbability component. Code used to generate plot is provided below the figure.

5. References

Cho SE. (2007) Effects of spatial variability of soil properties on slope stability. Engineering Geology, 92(3): 97-109.

El-Ramly H, Morgenstern NR, and Cruden DM. (2002) Probabilistic slope stability analysis for practice. Canadian Geotechnical Journal, 39(3): 665-683.

Hamlet AF, Elsner MM, Mauger GS, Lee S, Tohver I, Norheim RA. (2013) An Overview of the Columbia Basin Climate Change Scenarios Project: Approach, Methods, and Summary of Key Results. Atmos Ocean. 51(4): 392-415.

Hammond C, Hall D, Miller S, & Swetik P. (1992) Level 1 stability analysis (LISA), documentation for Version 2.0. USDA, For. Serv., Moscow, ID, Intermountain Res. Sta. Gen. Tech. Rep. INT-285.

Hobley DEJ, Adams JM, Nudurupati SS, Hutton EWH, Gasparini NM, Istanbulluoglu E, and Tucker GE. (2017) Creative computing with Landlab: an open-source toolkit for building, coupling, and exploring two-dimensional numerical models of Earth-surface dynamics, Earth Surf. Dynam., 5: 21-46, doi:10.5194/esurf-5-21-2017.

Jin S, Yang L, Danielson P, Homer C, Fry J, and Xian G. (2013) A comprehensive change detection method for updating the National Land Cover Database to circa 2011. Remote Sensing of Environment, 132: 159 175.

Montgomery DR, and Dietrich WE. (1994) A Physically Based Model for the Topographic Control on Shallow Landsliding, Water Resources Research, 30(4): 1153-1171.

Pack RT, Tarboton DG, Goodwin CN (1998) The SINMAP approach to terrain stability mapping. In: Proceedings of the 8th international congress of the international association of engineering geology and the environment, Vancouver, British Columbia, Canada, September 21–25, vol 2. AA Balkema, Rotterdam, pp 1157–1165.

Peckham SD. (2014) 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 Modeling and Software, edited by Daniel P. Ames, Nigel W. T. Quinn, A. E. R., International Environmental Modelling and Software Society (IEMSs), San Diego, California, USA.

Strauch RL, Istanbulluoglu E, Nudurupati SS, Bandaragoda C, Gasparina N, and Tucker G. (2018). Hydro-climatological approach to predicting regional landslide probability. Earth Surf. Dynam. 6, 1-26.

USGS. (2014) National Land Cover Data (NLCD) version Marched 31, 2014. National Map Viewer [Accessed November 25, 2014].