Documentation for transient pore pressure model for predicting slope failure constrained by remote sensing data

Simon M. Mudd, Boris Gailleton, School of Geosciences, University of Edinburgh
(b.gailleton@sms.ed.ac.uk)

Table of Contents

Chapter 1. Introduction

This document describes the installation, running, and outputs of a slope stability model developed for combining remote sensing and rainfall data, built with the support of the European Space Agency (ESA) as part of a General Support Technology Programme (GTSP).

1.1. Purpose and Scope

The GSTP project objective is to de-risk an operational methodology for determining the probability of slope failure (quantified using the 'factor of safety' approach) from rain events by developing a remotely sensed, rainfall driven landslide model that can be used in slope failure management. Therefore, the goal of the project is to develop a technical system designed to progress the current science of predicting rain-induced natural slope failure. It uses the concept of creating ‘virtual models’ of the physical environment and its characteristics from space observations which is used as a digital twin, on which various rainfall scenarios can be run to simulate the landscape behaviour.

The purpose of this document is document software developed in work package 4. This work package involves development of the landslide model, dividing the work in three different tasks: evaluation of model performances against a test site monitored in-situ; evaluation of the model performances on the same site solely using remote-sensing data; comparison of the results of the two approaches.

WP 4 aims to evaluate the performance of using remote sensing data to detect and anticipate landslide activity. Simulating slope failure involves a very high degree of uncertainty. A common approach consists in modelling steady state pore pressure in response to a threshold rainfall event to generate static safety maps. An alternative approach is to simulate the transient pore pressure to consider the time-evolution of the effect of precipitation and soil water content on slope stability. Such a model is highly case-dependent and requires a significant amount of expensive and time-consuming in-situ monitoring of soil hydraulic, mechanical and hydrogeologic features. The aim of this code is to use an already-monitored site to apply the Iverson (2000) model and compare the performance of the model constrained with in-situ data with the model constrained with remote-sensing derived data. The following documentation proposes a protocol to run the model from the different data sets and how to visualise and explore the outputs. The core of the modelling code is written in C++ to ensure computational efficiency. The automation for multiple simulations and I/O management is written in `python`.

Reference

Iverson, R.M., 2000. Landslide triggering by rain infiltration. *Water Resources Research*, 36, 1897–1910. https://doi.org/10.1029/2000WR900090

1.2. Parameters

A number of parameters are required by model:

- The model needs precipitation data input to simulate the transient pore pressure evolution in the modeled soil column. This data has to be formated as *precipitation intensity* in length unit per time unit, preferably in seconds.
- Hydraulic parameters from the modelled soil: the hydraulic diffusivity (*D0*), the Hydraulic conductivity (*Ksat*), the steady-state infiltration rates (*Iz/Kz*).
- The depth of the water table and of the substrate.
- The mechanical soil properties, soil cohesion (soil capacity to resist to motion), friction angle, weight of soil and weight of water.
- The landscape property: topographic slope.
- The model resolution.

1.3. Global approach

For a specific site, neither remote-sensed nor in-situ data would constrain discrete values for each parameters. We therefore run the model using a Monte Carlo sampling schemes on range of possible parameters. In the case of testing the model against in-situ data, ranges of parameters are determined by different mechanical and hydraulic tests. Calibration is achieved with field observation of ground motion. In the case of testing the model against remote-sensed constraints, the ranges of parameters are suggested from general ranges in the literature and calibrating failures are recorded from InSAR data that detect ground motion.

1.4. Model output

The main applicable output of the model is a factor of safety (FS) for a soil columns that determines when the slope is subjected to fail. This factor of safety is an absolute value defined by equation 18 in Iverson (2000). It suggests failure when the FS is ≤ 1. This factor of safety is calculated for a given time and depth of the modelled soil column, driven by a rainfall time series.

1.4.1. For a single simulation

The models outputs table-like `csv` files which include the factor of safety and the pore pressure as a time series. It also can output factor of safety maps for a given time and a given Digital Elevation Model (DEM) to identify the area at risk.

1.4.2. For Monte-Carlo Runs

We have developed `python` tools to control the Monte-Carlo simulations, where parameters are selected from a probability distribution. Because of the significant number of runs involved, all the `csv` outputs are concatenated into a `hdf5` data file, which is easily readable from `python`. Time series of factor of safety for all the simulations can be synthesised into a distribution of detected landslide failures as a function of time. These can be correlated with calibration data (e.g. observed or measured ground motion from InSAR or in-situ monitoring) to determine "successful" simulations. Once "successful" runs have been defined, visualisation of the parameter sets that leaded to failure detection can be used to constrain the model.

Chapter 2. Installation

All the code has been developed, tested and run on `UNIX` platform. It requires uses of a C++ compiler and a `python 3` environment.

2.1. The code

The code has been developed within the open-source LSDTopoTools research software suite. The deliverable has been packaged into a `zip` file with the final report and contains all the code required for running the analysis.

2.2. Compiling the model

The C++ model is in the subdirectory `cpp_model`. You must navigate into this directory and compile the code.

Compilations can be done with the `GNU g++` compiler, installed in most `UNIX` and `IOS` distributions. The `Analysis_driver` folder in the root folder contains a `make` file that automates the compilation. Navigate to the directory where this make file in a terminal window and compile with:

```
$ make -f iverson_model.make
```

It generates an executable file named `iverson_model.exe` in the same folder and is ready to be used.

2.3. Installing the `python` package

The `python` package which we use to manage our simulations (called `iverson_python`) can be installed in any `python` environment satisfying the following dependencies: `numpy`, `scipy`, `pandas`, `pytables` and `matplotlib`. The package is downloaded with the `github` repository in the folder `python_tools` and can be installed in the `python` environment using `pip install .` in the `python_tools folder`.

2.3.1. Optional: Getting a python environment with conda

If the user does not have `python` on their system, they will need to install an environment for the `python` scripts. We have written this documentation assuming use of the `miniconda` environment manager. Miniconda is an easy, clean and open-source solution. It can be downloaded from here. Choose any `python 3` installer compatible with your operating system. Once installed it offers a clean method to manage different `python` environments with different combinations of packages installed. Once install, run the following command to create an environment:

```
$ conda create -n iverson_python
```

This creates a `python` environment named `iverson_python`. Whenever you start a session you can activate this environment with:

```
conda activate iverson_python
```

When the environment has been successfully activated if `(iverson_python)` is stated at the beginning of the command line. After the first activation, the dependencies can be installed with:

```
conda install -c conda-forge numpy scipy pandas pytables matplotlib
```

This only needs to be run once. All installed packages will remain available when you reactivate the environment.

After you have done this, you into the `python_tools` directory and run the command

```
$ pip install .
```

Chapter 3. Preprocessing the precipitation data

Precipitation data drives the transient pore pressure model. The data can be from local instruments, global datasets or simulations. The data needs to be in a `csv` (Comma-separated-values) files containing the following columns:

```
duration_s,intensity_mm_sec
...,...
```

Where `duration_s` represent a duration in seconds and `intensity_mm_sec` is the corresponding precipitation intensity in mm per seconds. The intensity column usually requires unit conversion from common datasets. For example, `15 mm` of rain in 3 hours becomes `15/10800=0.001389 mm/second`.

Chapter 4. Running the model for a single soil column

The following instructions describe how to run the code and visualise the output for a single soil column simulated.

4.1. Simulation

Once compiled, the C++ executable can be run from the command-line and need to be directed to a **parameter file**. The **parameter file** is a text file containing a series of values telling the C++ model what parameters to use. The parameter files can be written as follow in a text file:

```
# This is a parameter file for the Iverson model
# Lines beginning with # are comments and are not read by the software

# The read path is where the data will be read
read path: /path/to/file/
# The write path is where the data will be written
write path: /path/to/file/
# Write fname is the prefix of the written files
write fname: 20150711_20160809_filtered

# Name of the csv file containing the preprocessed precipitation data
rainfall_csv: preprocessed_data.csv

# Parameters for the models
# See documentation for units
D_0: 0.000005
K_sat: 0.00000005
d: 2
Iz_over_K_steady: 0.2
alpha: 0.51
friction_angle: 0.38
cohesion: 12000
weight_of_soil: 19000
weight_of_water: 9800
depth_spacing: 0.1
n_depths: 35


#end of file
```

The executable needs to know the path and the name of the parameter file to read them correctly. From a terminal in the folder containing the compiled executable, the following command will run the model:

```
./iverson_model.exe /path/to/the/parameter/file/ name_of_the_parameter_file.param
```

Note that the parameter file is a text file and can be saved with any extensions (e.g., `.param`, `.driver`). The following parameters can be provided to the model.

Parameter	*Description*	*Units*
rainfall_csv	the name of the preprocessed `csv` file with the rainfall time series.	none
D_0	Hydraulic diffusivity	m^2/s
K_sat	Hydraulic conductivity	m/s
d	Depth of water table	metres
Iz_over_K_steady	Steady-state infiltration rate	dimensionless
alpha	Topographic slope	radians
friction_angle	Critical friction angle for the soil	radians
cohesion	Soil cohesion	Pa
weight_of_soil	Volumic weight of the soil	$kg/(s*m^2)$
weight_of_water	Columic weight of the water	$kg/(s*m^2)$
depth_spacing	Model resolution	metres
n_depths	Number of discretisation	nondimensional integer

4.2. Model outputs

The C++ model produces 6 `csv` files containing the evolution of various characteristics of the soil column. Four csv files contain the time series of the different components of the Factor of Safety (FS) functions of time and depth. The independent components of FS are described in equations 28 (a,b and c) of Iverson (2000) and their time series are in files `XXX_time_series_depth_Fcomp.csv` , where `XXX` is the `write fname` in the parameter file and `Fcomp` the corresponding component. It also outputs the final FS as well as the evolution of the transient pore pressure (Psi) in the file `time_series_depth_Psi.csv`. All of these files have the same structure:

- The first column is the depth below the surface (in metres)
- All the following columns are the corresponding values for each simulated time, where the first row is time.

4.3. Spatial analysis

Each model run predicts the evolution of the factor of safety for a single soil column through time. Once a time has been identified, a raster can be fed to the model to generate a risk map. The raster will be downsampled, the slope will be calculated and factor of safety is calculated for each soil column. The following lines can be added to the parameter file to enable the spatial analysis:

```
full_1D_output: false
spatial_analysis: true
reload_alpha: false
resample_slope: true
resample_slope_res: 6
topo_raster: insar_area_PP
polyfit_window_radius: 1
n_threads: 4
time_of_spatial_analysis: 19818000
```

Where:

- `full_1D_output: false` disables printing of full time series information of the entire depth profile.
- `spatial_analysis: true` enables the analysis.
- `reload_alpha: false` enables (true) or disables (false) the reloading of previously calculated slope map.
- `resample_slope` determines if the raster is down-sampled before slope calculation (for speed purposes) with a new resolution.
- `resample_slope_res` is the resolution of the resampled DEM
- `topo_raster` is the name of the DEM/DTM raster without its extension (it has to be an ENVI bil file).
- `polyfit_window_radius` is radius of the polynomial fitting window that is used to calculate slope gradient.
- `n_threads` is the number of threads to use for multithreading.
- `time_of_spatial_analysis` is the selected time in the precipitation time series. It produces a raster with the factor of safety for each soil column.

Note that most users will want to first run a 1D analysis to select the time of interest to produce the spatial map of the Factor of Safety (FS).

Chapter 5. Monte Carlo analysis to constrain the model

The Monte-Carlo analysis requires running a large (>10000) number of simulations in order to produce representative results. The software uses `python` as scripting language to automate the simulations and save/load/process the data. The installation section explains how to set up a `python` environment. The environment must be used to run python scripts. The Monte Carlo analysis is designed to define ranges of each of the 9 parameters in the model. It does this by running simulations that randomly sample these parameters within a predifined parameter space, and testing the resulting factor of safety time series against observed ground motion. The advantage is that running all the combinations would take several years of computation, whereas random sampling covers most of the spectrum in significantly less simulations.

5.1. Building the model

The model needs to be initialised with ranges of parameters as follow:

```
from plot_iverson_lsdtt import MonteCarloIverson

my_simulation = MonteCarloIverson(range_D_0 = [1e-8,1e-4], range_Ksat = [1e-11,1e-6],range_Iz_over_K_steady = [0.1,1.5], range_d = [0.5,3], range_alpha = [0.1,1.5], range_friction_angle = [0.1,1],
        range_cohesion = [100,5000], range_weight_of_soil = [15000,30000], range_weight_of_water = [9800,9800], depth_spacing = 0.1, n_depths = 35, OMS_D_0 = 1.,OMS_Ksat = 1., OMS_d = 0.1 , OMS_alpha = 0.1, OMS_friction_angle = 0.05, OMS_cohesion = 100,OMS_weight_of_soil = 500,OMS_weight_of_water = 100, OMS_Iz_over_Kz = 0.1,
        program = "", path_to_rainfall_csv = "./", rainfall_csv = "preprocessed_data.csv", path_to_root_analysis = "./", suffix = 0)
```

Most of the parameters are self-explanatory: `range_X` define the minimum and maximum values for the corresponding parameter; `OMS_X` define the precision step of the random sampling (`D_0` and `Ksat` are in log space); `n_depths` defines the number of depth to be investigated, `program` points to the path+name of the `exe` file (compiled model); `path_to_rainfall_csv` and `rainfall_csv` points to the rainfall csv file and finally `path_to_root_analysis` points to the root directory of the analysis (many subdirectories may be created below). The parameter `suffix` is set to `0` if you enter the full `csv` filename with extension. If this is `1` it assumes the extension is `csv`. Of the parameters, there should be no range for the `weight_of_water` as this is just the density of water times gravitational acceleration and does not vary.

5.2. Running the simulation

To run the simulation, simply add the following line of code on the script above that contains the ranges:

```
my_simulation.run_MonteCarlo(failure_time_s = 540460, n_proc = 4, n_tests = 1000)
```

Where `failure_time_s` is an optional feature if a specific time is chased: it would add a data to the output expressing the time of first failure compare to that one (it does not affect the other outputs); `n_proc` defines the number of processors to be used in parallel: the code is pleasingly parallel which means all the tasks are independent from each others and can be run at the same time; `n_test` defines the number of simulations to run. For example, we ran 70000 simulation on 24 cores on our server with a broad range of parameter taken from literature with the following script:

```
from plot_iverson_lsdtt.sensitivity_analysis import MonteCarloIverson

program =
"/home/s1675537/PhD/DataStoreBoris/dev_LSD/LSDTT_Development/src/Analysis_driver/Ivers
on.exe"
MySim = MonteCarloIverson(program = program, path_to_rainfall_csv = "./", rainfall_csv
= "preprocessed_data.csv", path_to_root_analysis = "./test_from_in_situ/",
depth_spacing = 0.1,
    n_depths = 32, range_D_0 = [5e-7,5e-5], range_Ksat = [5e-10,5e-
7],range_Iz_over_K_steady = [0.1,0.8], range_d = [1,3], range_alpha = [0.5,1.1],
range_friction_angle = [0.3,0.5],
        range_cohesion = [15500,16500], range_weight_of_soil = [15000,20000],
range_weight_of_water = [9800,9800],
        OMS_D_0 = 0.05,OMS_Ksat = 0.05, OMS_d = 0.05, OMS_alpha = 0.05,
OMS_friction_angle = 0.03, OMS_cohesion = 100,OMS_weight_of_soil =
100,OMS_weight_of_water = 100, OMS_Iz_over_Kz = 0.05)

MySim.run_MonteCarlo(failure_time_s = 0, n_proc = 24, n_tests = 200000, save_to_db =
True)
```

5.3. Outputs

While the model is running, a significant amount of temporary output data will be generated. This is due to the fact that the model write some results to the disk before appending them every `n_proc` runs in a final file. Do not delete it as the software does it automatically.

The most important output is a `csv` file (that can be opened by any common software) named `failure_global.csv`. It contains a synthesis of each simulation with (i) the parameters tested and (ii) the time of failure. It allowed us to produce the histograms of time of failure and isolate successful parameters.

Another file is generated containing more details about each run (time series): because of the significant amount of data produced by the simulations, the software uses the `hdf5` data structure via `pandas` to manage it. We strongly advice to use `pandas` in `python` to explore the data. One can open a file with the list of keys as follow:

```
import pandas as pd

glob = pd.HDFStore("test_from_in_situ/db_of_failure.hd5")
print(glob.keys())
glob.close()
```

It will display a lot of keys and might take time...

To access the dataframe of one single run and potentially save it for more data mining:

```
import pandas as pd

glob = pd.HDFStore("test_from_in_situ/db_of_failure.hd5")
df = glob["name_of_my_key"]
df.to_csv("/path/plus/name.csv", index = False)
glob.close()
```

5.4. Performance

Assessing performance can be difficult: due to the high number of simulations run within the Monte Carlo framework, there are many unsucessful simulations, but the general pattern still highlights the failure quite well. The following code produces first order performance metrics based on the file produced by the `python` package. It takes the time of observed failures and an interval to calculate the ratio of successful simulations. It can be used to isolate the successful combination of parameters that lead to real failures.

```
import pandas as pd

# Load the dataframe
df = pd.read_csv("failure_global.csv")

# Gater the interval
sttest = []
endtt = []
interval = 3600 * 24 * 5 # 5 days to seconds
for i in [1.1e7,1.48e7,1.85e7,2e7]: # this array contains the times of observed
failure in seconds
    sttest.append(i - interval)
    endtt.append(i + interval)

# number of elements
n_total = df.shape[0]

# Sorting my failure times to isolate the successful ones
lsofdf = []
for i in range(len(sttest)):
    lsofdf.append(df[(df["first_failure"]>sttest[i]) & (df["first_failure"]<endtt[i])]
)
QDF = pd.concat(lsofdf)

# printing the results

print("Total number of run: %s " %(n_total))
print("Total number of successful run: %s " %(QDF.shape[0]))
print("Total number of run showing constant failure: %s "
%(df[df["first_failure"]==0].shape[0]))
print("Ratio of success: %s " %(QDF.shape[0]/n_total ))
print("Ratio of success ignoring 0: %s " %( QDF.shape[0] / (
df[df["first_failure"]!=0].shape[0]) ) )
```