Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Introduction to Catastrophe Risk Modelling

*Spring Semester 2022*

# Project - River Flood Risk

Figure 1: The Ahr River in Insul, Germany, on July 15, 2021 after heavy rainfall. [1]

Hutter Valeria
Lombardi Matteo
Nasrazadani Hossein

May 10, 2022

# Table of contents

# 1 Introduction

## 1.1 Generalities on Peril Considered

The objective of this project is to develop a catastrophe risk model. To do this, it was decided to choose a natural, hydrological hazard, namely a river flood.

This is a natural hazard that affects us very closely and and as of late came up frequently, such as the flooding following the severe weather last summer in various regions of Germany and Belgium.
Heavy and prolonged rainfall, as well as in extreme cases snow or melting ice, can lead to high runoff from a reservoir into a river, which can result in flow over the riverbanks and subsequent flooding causing damage.

Floods are constrained by a river system associated with a specific catchment area. This consists of the area of land where precipitation is collected and flows towards an outlet, such as a river. The main parameters characterising floods are the topography of the basin (land gradient, flow distance, soil characteristics) and the amount of rainfall (quantity and duration of rainfall).

For this reason, it was decided to consider a relatively small regional area for the project. In this case, the region of the city of Chur in the canton of Grisons in Switzerland was chosen, where the main data for creating the model are available. This location is crossed by two rivers, the larger Rhine and the smaller Plessur, which then joins the Rhine, as can be seen in figure 2. In any case, the model that is created focuses on the Rhine, which flows in a north-eastern direction.

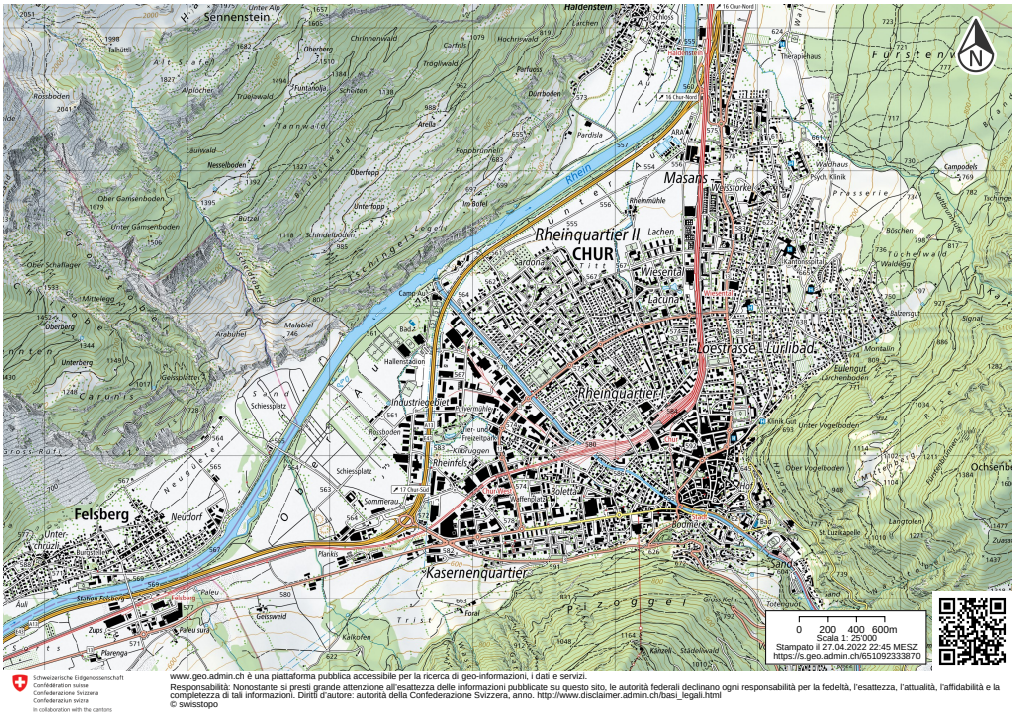



Figure 2: City of Chur with the two main rivers. [4]

The size and intensity of a flood are described by the inundation depth $h$, which depends on the discharge $Q$ and thus also on the velocity $v$ and the morphology of the location.
The modeling of river floods is based on the equations derived from the conservation of mass and linear momentum, also called Navier-Stokes equations. As these are complex, flood modelling is approximated by a cellular automata. Other simplifications and assumptions during the various steps of the model creation will be made and are reported in section 4.1. This is both for reasons of simplicity, since the model can always be improved at a later stage, but also for reasons of time limitation.

# 2 Methods & Data

## 2.1 Methodology

To assess the risks of flooding in the area of study, a simulation methodology was developed. The methodology is composed of two models, namely flood and exposure, which will be described in more detail in the following sections. The methodology starts with user-defining a hazard scenario by specifying its return period. The methodology proceeds with calculating the corresponding discharge value of water inflow and having the flood model simulate the temporal evolution of water flow using a cellular automata approach. Next, using the entire set of inundation values for each time step, the highest water depth is considered as the hazard footprint for that specific scenario. The hazard footprints of all hazard scenarios are then aggregated based on their probability of occurrence and the hazard map and hazard curves are obtained. The hazard curve essentially shows the exceedance probability of having a certain hazard intensity. Lastly, the methodology calculates the degree of damage to each asset using the hazard footprints; this is achieved through the developed exposure model. The methodology concludes with calculating the risks and amount of loss per return period, developing the exceedance probability (EP) curve, and the calculating the average annual loss (AAL).

To facilitate running computational experiments, the methodology was coded in python; for convenience, the entire code including all the developed models is attached as an appendix to this report. The code was divided into three large parts in order to structure and understand it better. The first part, called "main", as the name implies, deals with initialisation, importing of source data, defining the context of the study region, as well as the context of the simulation and scenarios of interest, etc. The "main" module works as an orchestrating algorithm and calling up other models to execute their functions, communicate data, and produce their outputs in a consistent manner. The second part concerns all the modelling of the flood, while the last one contains all that is related to the exposure and vulnerability of the buildings to the peril.

For the spatial analysis, QGIS was used. This is an open-source GIS program which also runs on python.

## 2.2 Topography Data

For modelling the underlying topography, a gridded $16x16\,m$ digital elevation model is being used. The digital elevation data is available from swisstopo under the name "swissALTI3D" [5] at multiple resolutions. The resolution of 16x16 $m$ was chosen to reduce the calculation time, as the model area covers roughly $25\,km^2$. While an increase in resolution might improve the results, the results from a coarse resolution grid are deemed sufficiently accurate to draw relevant conclusions for the project. Figure 3 gives an overview of the model area using the digital elevation data from [4].

## 2.3 Runoff Data

As introduced in section 1.1, river flooding from the Rhine in the city of Chur is being modelled. The underlying assumptions regarding return periods of discharges and associated simplifications are presented in the following paragraphs.

Event size in flooding events is determined by the associated peak discharge $HQ$, thus every discharge has an associated return period. This information is available on the Federal Office of Environment's (FOEN) website for hydrological data and forecasts. The information from the Domat-Ems gauging station was used, since it is only a few kilometers upstream of Chur and there are no relevant tributaries between the gauging station and the model area. [4]

The statistical analysis on the return periods for the various discharges was provided by the hydrodata-portal of FOEN [2] and the graph is presented in figure 4.

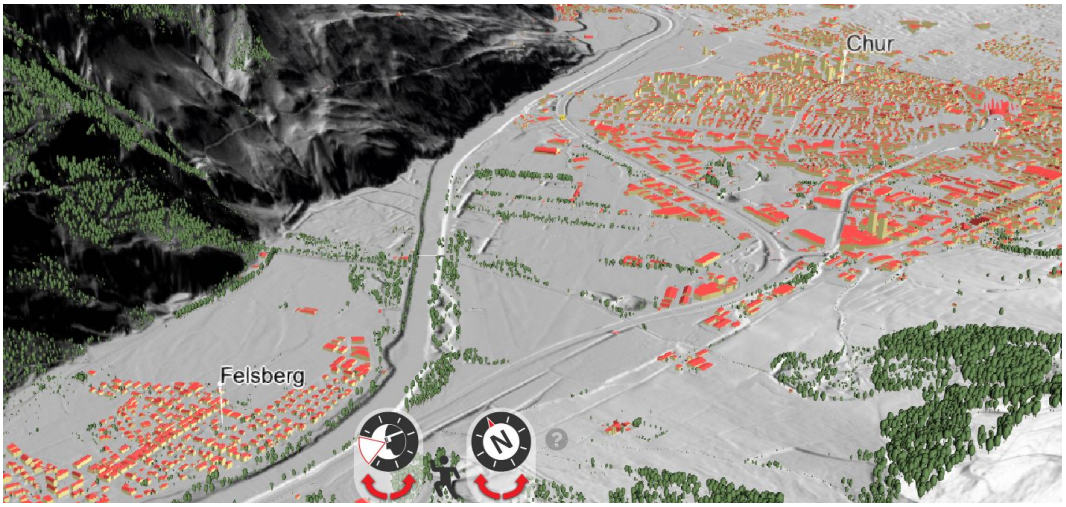



Figure 3: Bird's eye view of swissALTI3D data looking north-east over the model area with Chur in the centre. [4]

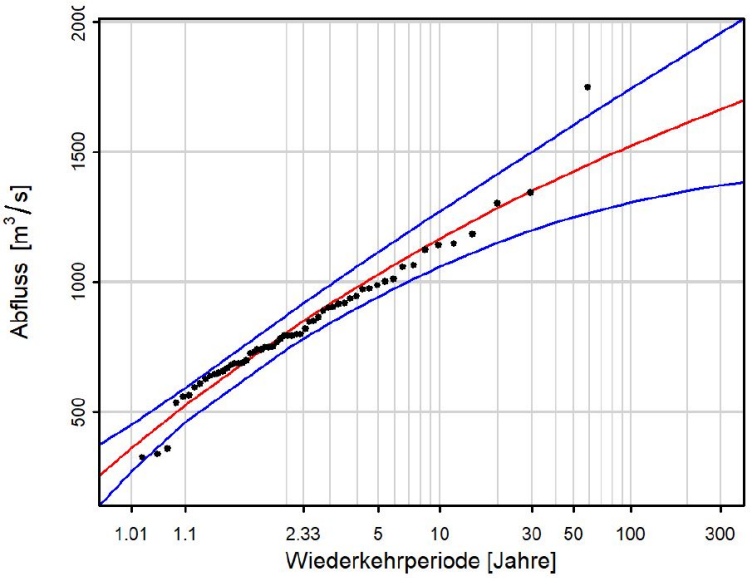



Figure 4: Annual maximum discharge statistics for the Domat-Ems gauging station. The x-axis describes the return period in years, the y-axis describes the discharge in $m^3/s$. [2]

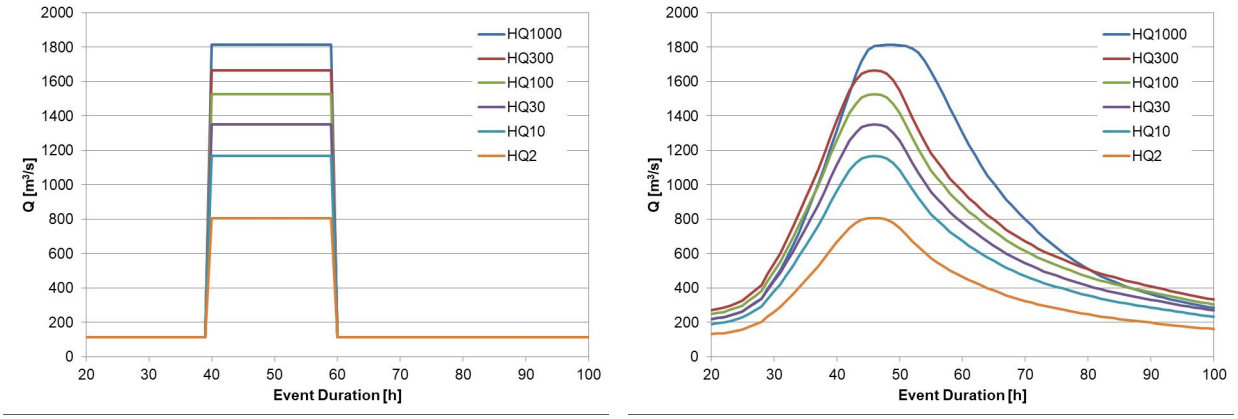
For the various scenarios, the return periods of 2, 10, 30, 100, 300, and 1000 years (1000 years = extreme event) were used. For the runoff-curves, simplifications were made for the model. Instead of using time-variable runoff-curves, a block runoff was assumed, as illustrated in figure 5. The underlying assumption is, that any event with the same peak runoff will cause the same extent of inundation.

Additionally, the required model input is the water elevation above the digital elevation model. For the river area, the digital elevation model uses the normal water surface. To calculate the relevant runoff for the event we therefore have to subtract the base discharge (LQ) from the peak discharge (HQ). The LQ is assumed to be 115 $m^3/s$ based on the information provided from [2].

Since the velocity is not relevant in a cellular automata model, the water depth per event and calculation cycle is determined as:

$$h = \frac{Q_i \cdot \Delta t}{A_{cell}} \quad [m] \tag{1}$$

With $A_{cell}$ as the model resolution and $\Delta t$ as the calculation time step.




(a) Cellular Automata Model input

(b) Hydraulic Model input

Figure 5: Simplification of the runoff-curves for the model compared to the hydraulic model curves for the Rhine at Domat-Ems.

## 2.4 Flood Model

A cellular automata model was used to simulate the flow of water and generate the inundation maps. The model follows a number of steps and determines how water is distributed from one cell to cells in its von Neumann neighbourhood. These steps are:

- The water elevation is calculated for each cell as the sum of its topographic elevation and current water depth.
- For each cell, the water gradient between that cell and its four neighboring cells are calculated.
- The calculated gradients are then normalized to find the relative gradient for each cell.
- For each cell, the amount of water that can be distributed is calculated. This equals half of the maximum gradient between that cell and its neighbours. Note that the amount of water that can be distributed cannot exceed the current depth of water in each cell.
- The calculated amount of water for distribution is then distributed to neighboring cells based on their relative gradient.
- This process is repeated for each time step.

Figure 6 shows a snippet of the implemented code that demonstrates the above steps.

```
waterelevation = self.topography + self.surfaceWater

gradient_left = (waterelevation[:, :-1] - waterelevation[:, 1:])
gradient_right = (waterelevation[:, 1:] - waterelevation[:, :-1])
gradient_top = (waterelevation[:-1, :] - waterelevation[1:, :])
gradient_bottom = (waterelevation[1:, :] - waterelevation[:-1, :])

gradient_left_pad = np.pad(gradient_left, [(0, 0), (1, 0)], mode="constant", constant_values=0)
gradient_right_pad = np.pad(gradient_right, [(0, 0), (0, 1)], mode="constant", constant_values=0)
gradient_top_pad = np.pad(gradient_top, [(1, 0), (0, 0)], mode="constant", constant_values=0)
gradient_bottom_pad = np.pad(gradient_bottom, [(0, 1), (0, 0)], mode="constant", constant_values=0)

gradients = np.stack((gradient_left_pad, gradient_right_pad, gradient_top_pad, gradient_bottom_pad))
gradients[gradients > 0] = 0
total_gradients = np.sum(gradients, axis=0)
total_gradients[total_gradients == 0] = 1E14

relative_gradients = (gradients / total_gradients) + 0.0

max_flow_gradients = gradients * 0.5
max_water_distribution = np.amax(np.abs(max_flow_gradients), axis=0)
water_to_distribute = np.minimum(self.surfaceWater,max_water_distribution)
distribution = (relative_gradients * water_to_distribute) + 0.0
```

Figure 6: Python code for cellular automata flood model.

In this model, the model time step is 1 h, but a finer calculation time step of 1 min was used to improve the performance of the model. At each calculation time step, water is added at the entrance cells to the region based on the corresponding peak discharge values of events, and then, through numerous time steps, is moved to the next cells until the water exits the region.

It is worth noting that this model is concerned with mass conservation, which is relevant for flooding, but does not account for momentum conservation, which is a second aspect of flood modelling. This is taken up in section 4.1 in the model limitations. In comparison to a numerical model, the cellular automata assumes some simplifications, as mentioned in section 1.1.

## 2.5 Damage Model

An initial key input for exposure is the base area of buildings $A_i$ $[m^2]$ affected by the flood.
This is known and determined from the open source data of the region available in QGIS.

Other necessary information is the number of floors of the various structures, as well as - if present, the number of underground floors of the buildings such as cellars, underground parking, basements, etc.

However, these attributes are not available. For this reason, after investigating the geographical area of interest by means of swisstopo and Google Street View, it was decided to consider a single underground floor for each building as a reasonable assumption. At the same time, the information on the number of floors of the structure is not taken into account, since for an inundation only the ground floor and the basement are of interest.

The damage costs of each building are calculated using the following formula [3]:

$$D_i = V_{m,i} \cdot E_{c,i} \tag{2}$$

Where $V_m$ is the volume of each structure exposed to flooding. It is expressed as follows:

$$V_m = 0.25 \cdot V_b + 1.00 \cdot V_1 = A_i \cdot 2.5 \cdot 1.25 \quad [m^3] \tag{3}$$

This means that the entire volume of the underground spaces $V_b$ of the buildings is taken into account and only a quarter of the volume of the ground floor $V_1$, since a flood would not develop by occupying the entire ground floor. In addition, as a simplification, an average height of the levels of 2.50 $m$ is assumed, thus obtaining an expression that is only a function of the base area of the buildings $A_i$.

The $E_c$ value is the monetary exposure of the building. Since not all buildings have the same importance and sensitivity during a flood, they are divided into two categories. The first category contains all the highly sensitive and critical buildings, such as hospitals, museums, stations, power plants, churches, etc. The

second category contains all non-critical buildings such as residential buildings, as well as all industrial buildings with no particular dangerous or high risk activities.

$E_c$ is expressed as a function of flood depth $h_i$. As a unit cost of the damage caused per cubic metre, CHF 900 is allocated for critical structures and CHF 400 for non-critical structures. These values are normalised to a water flood depth of 25 $cm$, as shown in the following equations:

$$E_{c,crit.} = 900 \cdot \frac{\sqrt{h}}{\sqrt{0.25}} \quad [CHF] \qquad for\ critical\ buildings \tag{4}$$

$$E_{c,non\ crit.} = 400 \cdot \frac{\sqrt{h}}{\sqrt{0.25}} \quad [CHF] \qquad for\ non\ critical\ buildings \tag{5}$$

The damage calculation in this model is therefore done by considering buildings only, other types of structures were ignored (bridges, road infrastructures, agricultural land, etc.). Moreover, the age of the structure is neglected for the buildings taken into consideration to further simplify the damage model.

# 3 Results

In the following section, the various results obtained from the model are reported and briefly illustrated. The various outputs refer to the different return periods taken into consideration, i.e. 2, 10, 30, 100, 300 and 1000 years. In order not to overfill the report with all the outcomes of the various return periods, only a couple of representative results per type of output are included.

## 3.1 Event Table

The event table (table 1) summarizes the model runs. Due to the time-intensive calculations, the six events included in the table were used for the hazard calculations, as will be illustrated in the following sections.

For each return period (and therefore for each peak discharge) one event was modelled and recorded in the event table. Since there is only one source of hazard, the model is expected to look the same for any event with the same return period.

Table 1: Event table for the modelled flood events of the Rhine in Chur. The discharge in the table is only the surplus flow above the base flow and therefore differs from the values in section 2.3.

| Event $X_i$ | Source | Discharge ($m^3/s$) | Return Period (years) |
|---|---|---|---|
| 1 | Rhine | 691 | 2 |
| 2 | Rhine | 1051 | 10 |
| 3 | Rhine | 1235 | 30 |
| 4 | Rhine | 1410 | 100 |
| 5 | Rhine | 1550 | 300 |
| 6 | Rhine | 1700 | 1000 |

## 3.2 Hazard Footprint

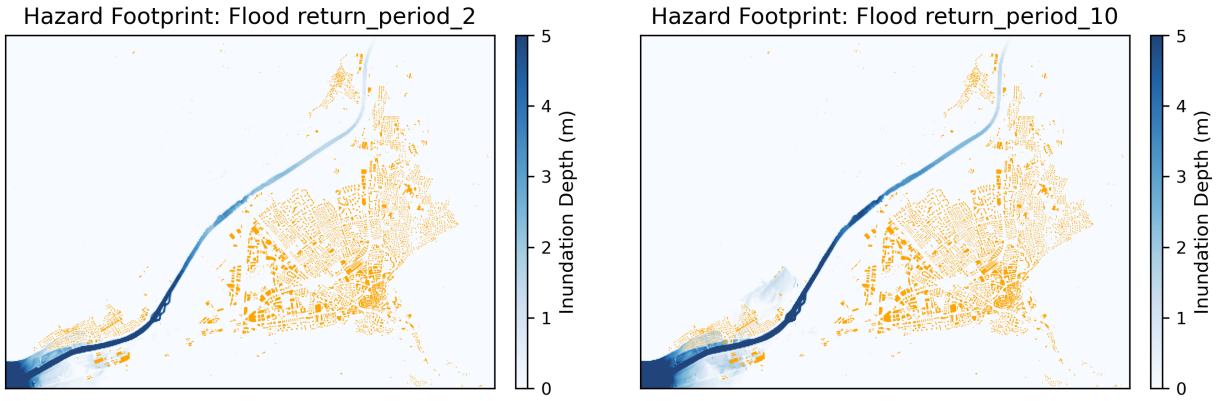
Flooding events usually have a singular point or line source - an outlet or a river. In the case of the Rhine in Chur, the water source is the upstream river. For modelling purposes, a simplified inflow as described in section 2.3 was used.

Figures 7(a) through 7(f) show the hazard footprints for the modelled return periods of 2, 10, 30, 100, 300, and 1000 years. The Rhine flows from the bottom left corner to the top right corner in the figures. The initialization cell of the model is the bottom left corner, thus the model results in that area are not applicable to reality.

A difference is visible between the footprints of different return periods. With increasing return period - and thus increasing runoff - flooding propagates downstream towards the city of Chur and affects larger areas and more buildings. This observation is supported by the hazard metrics in section 3.5 that show increasing losses for more severe events.

The flooding is mostly constrained to the river, additionally, there is shallow inundation in the western part of the city of Chur both on the left and right banks of the river.

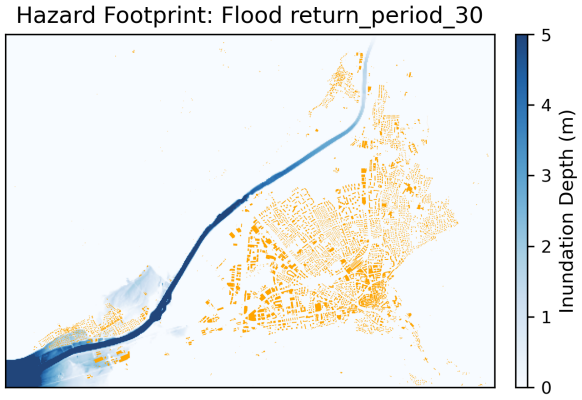
The observed flooding from the hazard footprints is measured in meters of water depth and shows the maximum inundation depth and extent during the event. Due to the topography, the flooding is constrained to the bottom of the Rhine valley. Referring to historical maps, this is also the area where historically flooding was most likely to occur, which is even in the name of the area “Ober Au” (“upper floodplain”). [4]



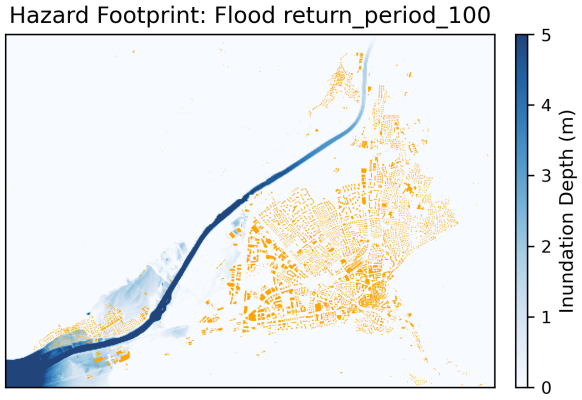

(a) T = 2 years



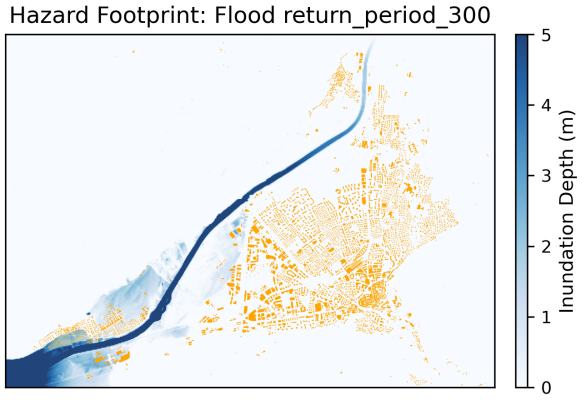

(b) T = 10 years



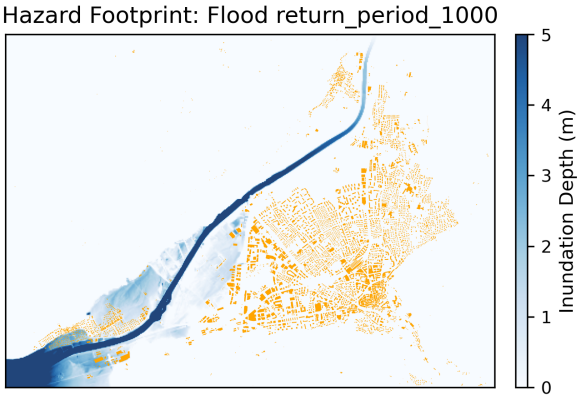

(c) T = 30 years




(d) T = 100 years




(e) T = 300 years




(f) T = 1000 years

Figure 7: Hazard footprint catalogue for the modelled return periods.

### 3.3 Hazard Map and Hazard Curve

To create the hazard curves, the hazard footprint for each return period is determined, multiplied by their respective probabilities and then added to generate the hazard curve as a whole. This process is illustrated in figure 8. For the application of this study, since there is only one event per each of the return periods, the hazard footprint function is a reverse step function for each location. For instance, if the inundation depth at a certain location for the event of 300yr flood is four meters, this means that there is the probability of one that the inundation depth is less than four meters and zero afterwards. Therefore, the hazard curve, which is created based on weighted aggregation of hazard footprint function,

is a step-wise function, as shown in the rightmost graph in 8.

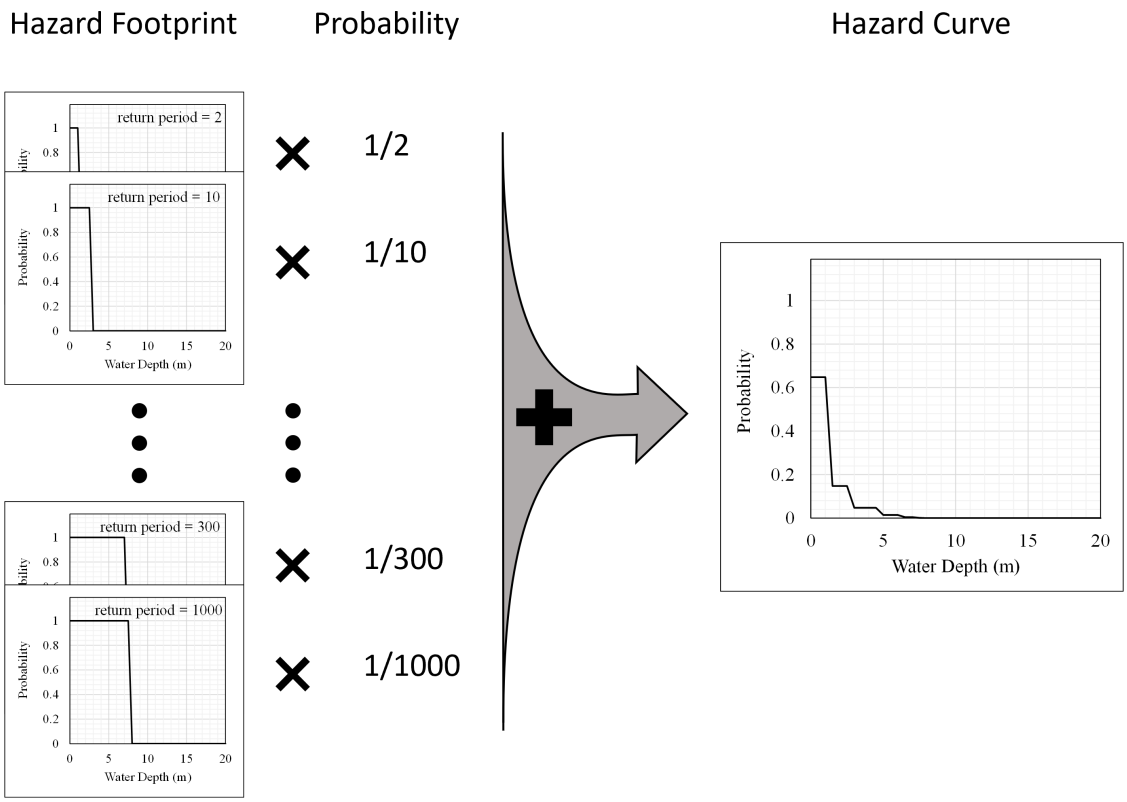
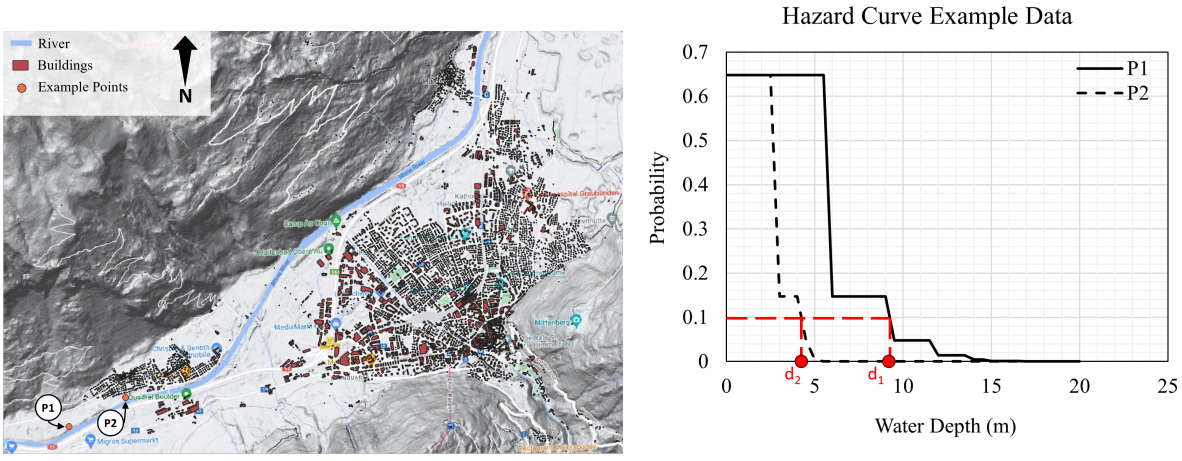



Figure 8: Process of developing hazard curves.

The resulting hazard curves for two example locations, as shown in figure 9(a), are displayed in figure 9(b). Each location has its own hazard curve, which in essence, summarizes the results of all hazard footprints into one single curve for that location. In comparison to the discharge-return period curve in figure 4 which describes the expected return period in the river at the gauging station, the hazard curve varies depending on the location.



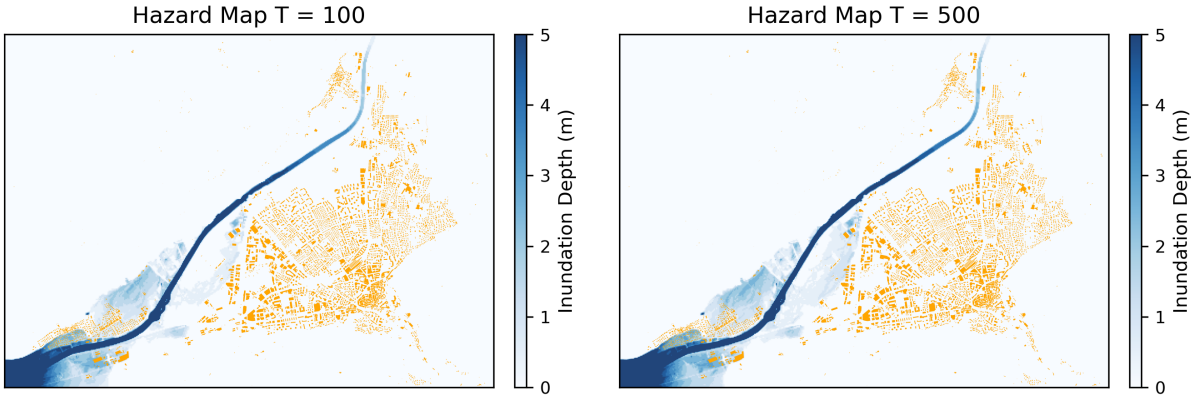

(a) Map overview of sample points

(b) Hazard curve for two sample points

Figure 9: Hazard curve for river flooding of the Rhine in Chur.

After generating the hazard curves for all locations, the hazard maps can be obtained. For that aim, the

return period of of the event of interest needs to be determined, e.g., 10 years. Next, the corresponding probability with that event needs to be found, e.g, 1/10 for the 10yr flood events. This probability is then used in conjunction with the hazard curves to find the hazard intensity, here water depth, whose corresponding exceedance probability matches with that of the event of interest. This process is illustrated in figure 9(b); here, $d_1$ and $d_2$ are the values of the hazard map for P1 and P2 for an event with 0.1 exceedance probability, i.e., with return period of 10 years. Figure 10 presents two example hazard maps for events of 100yr and 500yr return period.




(a) Hazard map for T = 100 years

(b) Hazard map for T = 500 years

Figure 10: Hazard map for two example return periods.

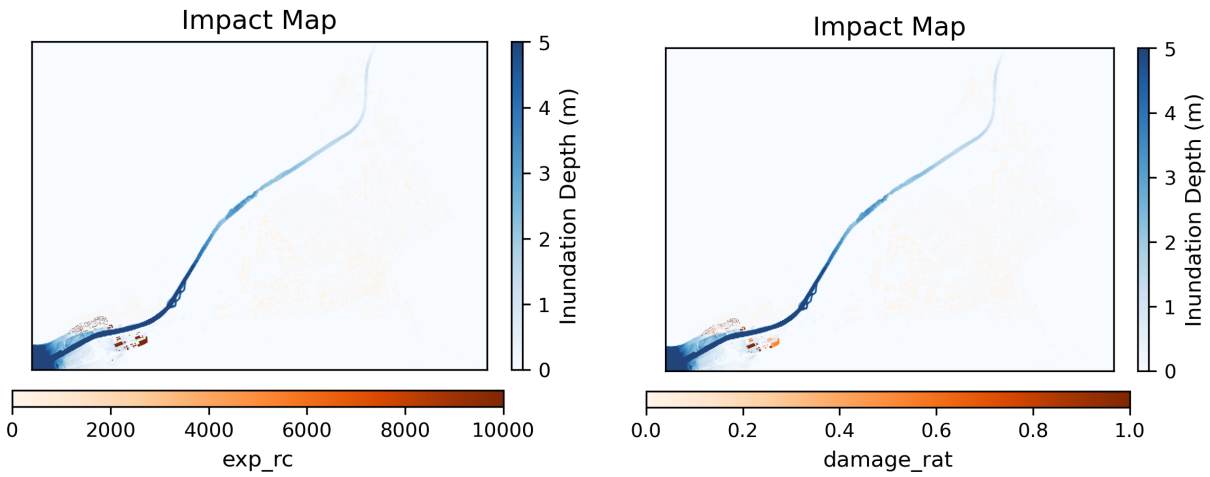
### 3.4 Damage/Loss Footprint Catalogue

Resulting from the information in the hazard footprints and the damage model, the damages and expected repair costs were calculated. These calculations in the model are included in appendix C line 329ff and follow:

$$Damage\,Ratio = \frac{min(rebuilding\,cost, damages)}{rebuilding\,cost} \tag{6}$$

Where the rebuilding cost describes the theoretical cost of reconstructing a building. This is calculated using the area of the building and cost constants for different building types.
The damages use the damage model in section 2.5, and differentiate between critical and non-critical buildings. The model code for these calculations is included in appendix C in line 329ff.

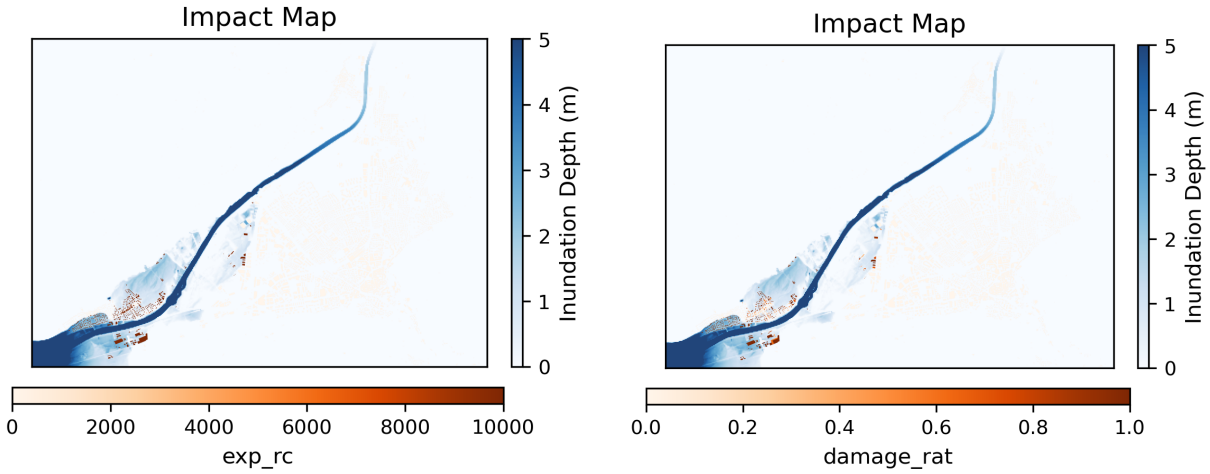
The expected repair cost in figures 11 and 12 are the damages incurred from inundation and are calculated using equation 2 with the respective number of affected buildings and inundation depth for the two presented return periods.




(a) Expected repair costs

(b) Damage Ratio

Figure 11: Impact map with repair cost and damage ratio for a return period T = 2 years.




(a) Expected repair costs

(b) Damage ratio

Figure 12: Impact map with repair cost and damage ratio for a return period T = 300 years.

### 3.5 Loss Table, AAL, EP Curve

In table 2, the event loss table for monetary loss is represented. It summarises in the various columns the type of event, the exceeding probability of the event, the associated monetary loss, the exceeding probability, as well as the expected loss for a given event, per year.

Table 2: Event loss table (ELT) for the calculation of the EP curve and expected loss.

| **Event** $X_i$ | Probability $p_i$ | Loss $L_i$ (CHF) (CHF) | EP | $\mathbb{E}[L] = p_i \cdot L_i$ (CHF) |
|---|---|---|---|---|
| 1 | 0.3934693 | 77,681,004.4 | 0.5356933 | 30,565,093.5 |
| 2 | 0.0951626 | 135,043,126.4 | 0.1422239 | 12,851,052.6 |
| 3 | 0.0327839 | 160,774,474.1 | 0.0470614 | 5,270,814.2 |
| 4 | 0.0099502 | 194,718,126.2 | 0.0142775 | 1,937,477.7 |
| 5 | 0.0033278 | 221,707,526.9 | 0.0043273 | 737,794.7 |
| 6 | 0.0009995 | 242,790,023.4 | 0.0009995 | 242,668.7 |

The loss information in table 2 provides the basis for hazard metrics calculations for the average annual loss (AAL) and the exceedance probability (EP). The AAL describes the expected annual losses incurred due to the flooding events. It is calculated as follows:

$$AAL = \sum (p_i \cdot L_i) \tag{7}$$

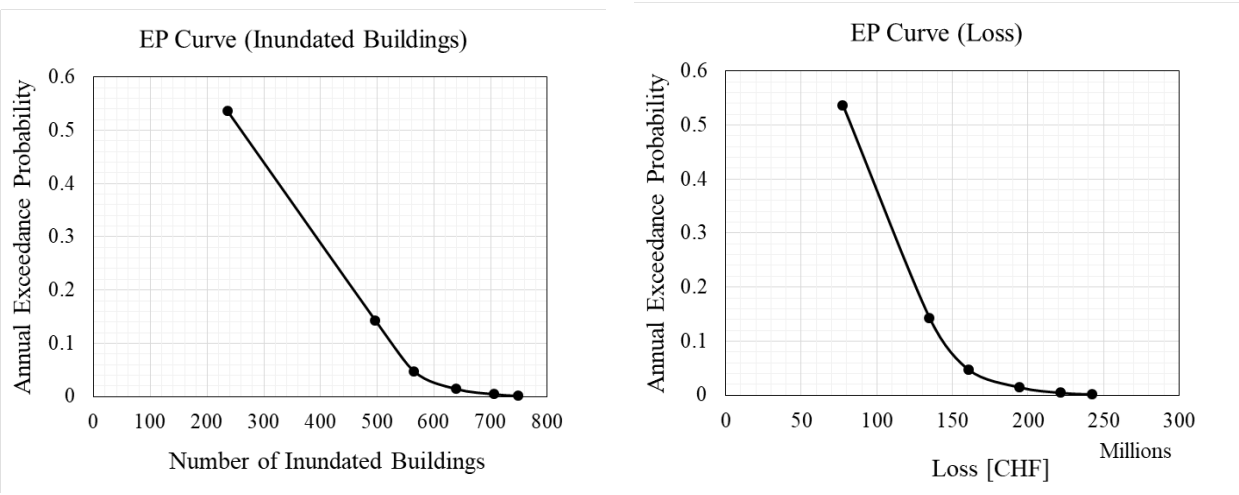
Where $p_i$ is the exceedance probability of event i and $L_i$ is the Loss because of event i.

The financial AAL is included in the first line in table 3. Looking at the additional loss parameters *inundated buildings* and *inundated area*, the results for these AAL are calculated from the model results and are as follows:

Table 3: AAL for Rhine flooding in Chur for the three impact parameters.

| Parameter | AAL value |
|---|---|
| Loss (CHF) | 51,604,901.48 |
| Inundated buildings (Number) | 168.14 |
| Inundated area ($m^2$) | 35,691.42 |

The annual exceedance probability curves are calculated based on the modelled events and are presented in figure 13.




(a) Affected buildings

(b) Losses (financial)

Figure 13: Annual exceedance probability for the number of affected buildings and for the losses in millions of CHF.

# 4 Conclusions

## 4.1 Model Limitations

There are multiple sources for model limitations for this particular case. These include the simplification assumptions on the input side (number of rivers, discharge, building vulnerabilities, data resolution), and on the calculation approach (cellular automata).

As introduced in the first section, only the Rhine was modelled due to capacity constraints. For a complete hazard map for the city of Chur however, other rivers and surface runoff channels should be taken into consideration. In particular the Plessur, which passes through the City and the old town, presents a potentially significant threat to the city. The chosen resolution of the topography data was insufficient to model the river though, and increasing the resolution would have exponentially increased the calculation duration.

The model focuses on inundation depth only, thus neglecting the flow velocity. This is an important parameter in the determination of danger from flooding. In reality, flood intensity maps also account for flow velocity when determining the hazard level as a shallow, yet fast flowing flood is potentially more hazardous than a static flood. Additionally, the flood model does not work properly at the inflow area (bottom left corner of the area) due to the fact that the inflow rate of water to the area, based on the peak discharge values per each event, are higher than the ability of water to distribute the water to the next cells with the same rate. This results in observing high inundation values around this area.

## 4.2 Recommendations for Improvements

The CAT risk model that has been developed, although simple with its various simplifications and assumptions explained above, allows to have a working model that simulates a flood in the chosen study area and allows to obtain realistic results.

Clearly, in order to develop even more realistic models, it is necessary to improve the basic model shown in the report, taking into account the various complexities of the flooding process and finding a way to reduce and replace these simplifications by models and equations that are increasingly closer to reality.

More concretely, the following improvements can be made:

- To better represent the actual situation in the model area, make sure to include all potential sources of hazards, here all rivers.
- Instead of using the simplified discharge curve through the blocks, use the complete curve in function of time.
- To improve the limitation of cellular automata model, it is good to use physics-based models with this hazard type.
- The peak discharge values used are based on historical data. However, climate change is expected to increase the intensity and frequency of rainfall, and therefore also of flooding. For this reason, it is good to use a future projection of peak flow values to be able to have a better risk assessment on future floods.
- Use probabilistic damage models that are able to capture the uncertainty in the level of damage per hazard intensity.
- Use other infrastructure systems besides buildings to better model damage and loss. Otherwise use better databases that have more detailed information about the structures in the study area, e.g. year of construction, occupancy class, elevation, etc.

## References

[1] CNN. *The Ahr River in Insul, Germany, on July 15, 2021 after heavy rainfall.* URL: https://edition.cnn.com/2021/08/23/europe/germany-floods-belgium-climate-change-intl/index.html.

[2] FOEN. *Hydrological Data: Rhein Domat-Ems.* 2022. URL: https://www.hydrodaten.admin.ch/en/2602.html.

[3] Arrighi C. Mazzanti B. Pistone F. et al. *Empirical flash flood vulnerability functions for residential buildings.* SN Applied Sciences 2, 904. 2020.

[4] swisstopo. *Maps of Switzerland.* 2022. URL: https://www.map.geo.admin.ch.

[5] swisstopo. *swissALTI3D.* 2022. URL: https://www.swisstopo.admin.ch/de/geodata/height/alti3d.html.

# Appendix

## A Python Code - Main

```
import sys

import numpy as np
import pandas as pd
import csv
import pickle
import math
import os
import json
import uuid
import shutil

from scipy.stats import norm
from time import strftime

from osgeo.gdalconst import *
from osgeo import osr, ogr, gdal
import geopandas as gpd
import subprocess
import matplotlib.pyplot as plt
import matplotlib.cm as cm

import matplotlib as mpl

from fiona import collection
from shapely.geometry import shape, LineString

from flood import FloodSimulator, TwoDFloodModel
from exposure import *

class Engine:
    def __init__(self):

        self.damageSimulationContext = SimulationContext(self)

        self.currentSimulationContext = self.damageSimulationContext

        configPath = os.path.dirname(os.path.realpath(__file__)) + "\CAT_Model.conf"

        configFile = open(configPath)
        data = json.load(configFile)

        regionConfig = data["RegionContext"]
        dataConfig = data["DataContext"]
        self.regionContext = RegionContext(self, width=regionConfig["width"], height
            =regionConfig["height"],
                                           west=regionConfig["west"], north
                                          =regionConfig["north"],
                                           resolution=regionConfig["resolution"],
                                          epsg=regionConfig["epsg"])
        self.dataContext = DataContext(self, sourceDataDirectory=str
            (dataConfig["sourceDataDirectory"]),
                                       preprocessedDataDirectory=str
                                         (dataConfig["preprocessedDataDirectory"]),
                                       simulationDataDirectory=str
                                         (dataConfig["simulationDataDirectory"]))

        self.utilities = Utilities(self)


        pass

    def preprocess(self):
        self.exposure.preprocess()



    def simulateHazard(self):
        print(self.utilities.getCurrentTime() + "------------ engine: simulating
            hazard  ------------")

        for s_i in range(0, self.currentSimulationContext.numberTimeSteps):
            print(self.utilities.getCurrentTime() + "------------ simulation time step
                : " + str(self.currentSimulationContext.currentTimeStep) + "
                ------------")

            self.floodSimulator.run()
            self.floodSimulator.writeResults()

            self.currentSimulationContext.currentTimeStep += 1

    def createHazardFootprint(self):
        print(self.utilities.getCurrentTime() + "------------ engine: creating hazard
            map ------------")
        self.floodSimulator.createHazardFootprint()
        self.floodSimulator.writeHazardFootprint()

```

```
    def computeImpact(self):
        print(self.utilities.getCurrentTime() + "------------ engine: computing impact
            ------------")
        self.exposureDamageSimulator.run()
        self.exposure.writeResults()

    def setUpDirectories(self, baseDir, dirs=["Flood", "Exposure"]):
        print(self.utilities.getCurrentTime() + "-- engine: creating output
            directories at " + str(baseDir) + " --")


        for dir in dirs:
            pathToCreate = baseDir + '/' + dir + '/'

            # this is a coarse and incomplete check that prevents to accidently clean
                root, i.e. "/", or any other base directories (hopefully).
            if not any(c.isalpha() for c in pathToCreate) and not (pathToCreate == "/"
                ):
                print ("WARNING: DIRECTORY TO BE CREATED DOES NOT CONTAIN ANY
                    ALPHABETIC LETTER. FOR SECURITY REASONS THE DIRECTORY WILL NEITHER
                    BE CLEANED NOR CREATED.")
                continue

            if not os.path.exists(pathToCreate):
                os.makedirs(pathToCreate)
                print("creating: " + str(pathToCreate))

        pass

def main(argv):
    if len(argv) == 0:
        print('-- no arguments given: quitting --')

        return

    # load config file
    engine = Engine()

    if argv[0] == "preprocess":
        engine.exposure = Exposure(engine)
        engine.preprocess()

    if argv[0] == "hazard":
        return_period = float(argv[1])

        engine.damageSimulationContext.currentRun = 'return_period_' + argv[1]
        engine.damageSimulationContext.return_period = return_period
        engine.damageSimulationContext.currentTimeStep = 0
        engine.damageSimulationContext.numberTimeSteps = 2000

        engine.setUpDirectories(engine.dataContext.getImpactDirectory(),["Flood"])

        engine.floodSimulator = FloodSimulator(engine, TwoDFloodModel(engine))

        # initialize simulation modules
        engine.floodSimulator.initialize()

        # start simulation
        engine.currentSimulationContext = engine.damageSimulationContext
        engine.simulateHazard()
        engine.createHazardFootprint()

    if argv[0] == "hazardCurve":
        engine.floodSimulator = FloodSimulator(engine, TwoDFloodModel(engine))
        engine.floodSimulator.initialize()

        sim_dir = engine.dataContext.simulationDataDirectory
        sub_folders = [name for name in os.listdir(sim_dir) if os.path.isdir(os.path
            .join(sim_dir, name))]
        all_hazard_footprints = {}
        for folder in sub_folders:
            return_period = int(folder.split('_')[2])
            hazardFootprintDir = sim_dir + '/' + folder + '/hazard_map_' + folder + '
                .tif'
            hazardFottprint = engine.utilities.load_raster(hazardFootprintDir)

            all_hazard_footprints[return_period] = hazardFottprint

        depth_range = np.arange(0, 50.5, 0.5)
        probability_range_map = [np.zeros_like(hazardFottprint) for _ in range(len
            (depth_range))]
        weigted_depth_map = np.ones_like(probability_range_map)

        for i in range(len(depth_range)):
            d =  depth_range[i]
```

```
            for key in all_hazard_footprints:
                return_period = key
                prob = 1/return_period
                probability_range_map[i] += prob * (1-np.heaviside(d -
                    all_hazard_footprints[key], 1))

            weigted_depth_map[i] = d * probability_range_map[i]

        probability_range_map_stacked = np.stack(probability_range_map)
        prob_list = probability_range_map_stacked.reshape(len(depth_range),-1).T
            .reshape(336,466,len(depth_range))
        np.save(sim_dir + "/hazardCurves.npy", prob_list)

        x1 = 321
        y1 = 15

        x2 = 298
        y2 = 60

        df_example_probs = pd.DataFrame(columns=['water depth', 'point 1', 'point 2'])
        probs1 = np.zeros_like(depth_range)
        probs2 = np.zeros_like(depth_range)

        for i in range(len(depth_range)):
            d1 =  probability_range_map[i][x1][y1]
            d2 =  probability_range_map[i][x2][y2]

            probs1[i] = d1
            probs2[i] = d2

        df_example_probs['water depth'] = depth_range
        df_example_probs['probability point 1'] = probs1
        df_example_probs['probability point 2'] = probs2

        sample_dir = sim_dir + '/sample_prob.csv'
        df_example_probs.to_csv(sample_dir)

    if argv[0] == "hazardMap":
        return_period = float(argv[1])
        engine.floodSimulator = FloodSimulator(engine, TwoDFloodModel(engine))
        engine.floodSimulator.initialize()
        scenario_prob = 1 / return_period
        sim_dir = engine.dataContext.simulationDataDirectory

        hazard_map = np.zeros([engine.regionContext.height, engine.regionContext
            .width])
        prob_list = np.load(sim_dir + "/hazardCurves.npy")
        depth_range = np.arange(0, 50.5, 0.5)

        for i in range(prob_list.shape[0]):
            for j in range(prob_list.shape[1]):
                probL = prob_list[i][j]
                for k in range(len(probL)):
                    p = probL[k]
                    if p < scenario_prob:
                        break

                hazard_map[i,j] = depth_range[k]


        hazard_map_dir = sim_dir + '/hazard_map_' + argv[1] + '.tif'
        hazard_map_image_dir = sim_dir + '/hazard_map_'+ argv[1]  + '.png'

        engine.utilities.save_raster(hazard_map_dir, hazard_map)

        shapefile_path = engine.dataContext.sourceDataDirectory + "/Buildings
            /chur_buildings.shp"
        engine.floodSimulator.geodataToImage(shapefile_path, hazard_map_dir,
            hazard_map_image_dir,
                                             "Hazard Map T = " + argv[1])



    if argv[0] == "impact":
        return_period = float(argv[1])

        engine.damageSimulationContext.currentRun = 'return_period_' + argv[1]
        engine.damageSimulationContext.return_period = return_period

        engine.setUpDirectories(engine.dataContext.getImpactDirectory(),["Impact"])

        engine.floodSimulator = FloodSimulator(engine, TwoDFloodModel(engine))
        engine.exposure = Exposure(engine)
        engine.exposureDamageSimulator = ExposureDamageSimulator(engine, engine
            .exposure)
```

```
        # initialize simulation modules
        engine.floodSimulator.initialize()
        engine.exposure.initialize()
        engine.exposureDamageSimulator.initialize()

        # start simulation
        engine.currentSimulationContext = engine.damageSimulationContext
        engine.computeImpact()

    if argv[0] == 'aggregateIndicators':
        df = pd.DataFrame(columns=["return period", "sum exp_rc", "count inundated",
            "area inundated"])
        sim_dir = engine.dataContext.simulationDataDirectory
        sub_folders = [name for name in os.listdir(sim_dir) if os.path.isdir(os.path
            .join(sim_dir, name))]
        print(sub_folders)

        for folder in sub_folders:
            indicatorFile = sim_dir + '/' + folder + '/Impact/buildingIndicators.json'
            if os.path.exists(indicatorFile):
                with open(indicatorFile) as json_file:
                    data = json.load(json_file)

                    return_period = int(folder.split('_')[2])
                    sum_exp_rc = data['buildingAggregators']['sum exp_rc']
                    count_inundated = data['inundationHazardAggregators']['count
                        inundated']
                    area_inundated = data['inundationHazardAggregators']['area
                        inundated']
                    df.loc[len(df.index)] = [return_period, sum_exp_rc,
                        count_inundated, area_inundated]

                    # print(data)


        df.to_csv(sim_dir + 'ResultsSummary.csv')

    pass


class RegionContext:
    def __init__(self, engine, width=1600, height=1200, west=741200, north=199596,
        resolution=16, epsg=21781):
        self.engine = engine

        self.width = width
        self.height = height
        self.west = west
        self.north = north
        self.resolution = resolution
        self.epsg = epsg

    pass


class DataContext:
    def __init__(self, engine, sourceDataDirectory="",
                 preprocessedDataDirectory="",
                 simulationDataDirectory="",):

        self.engine = engine

        self.sourceDataDirectory = sourceDataDirectory
        self.preprocessedDataDirectory = preprocessedDataDirectory
        self.simulationDataDirectory = simulationDataDirectory
        pass

    def getSimulationRunDataDirectory(self):
        return self.simulationDataDirectory + "/" + str(self.engine
            .currentSimulationContext.currentRun) + "/"

    def getImpactDirectory(self, subfolder=''):
        return self.simulationDataDirectory + self.engine.damageSimulationContext
            .currentRun + "/" + subfolder

class SimulationContext:
    def __init__(self, engine, timeStep=3600, numberTimeSteps=1, currentRun=1,
        currentTimeStep=0, return_period = 500,
                 name="damage_simulation"):
        self.engine = engine

        self.timeStep = timeStep  # in seconds; 900 equals 15 minutes
        self.numberTimeSteps = numberTimeSteps  # in seconds
        self.currentRun = currentRun
        self.currentTimeStep = currentTimeStep
        self.return_period = return_period


        self.name = name

```

```
317     pass
318
319 class Utilities:
320     def __init__(self, engine):
321         self.engine = engine
322
323     def load_raster(self, path_to_file):
324         # Load Raster Data START
325         dataset = gdal.Open(path_to_file, GA_ReadOnly)
326         input_grid = np.array(dataset.GetRasterBand(1).ReadAsArray(), dtype=float)
327         dataset = None
328         return (input_grid)
329
330     def save_raster(self, path_to_file, output_grid):
331         # get parameters
332         driver = gdal.GetDriverByName('GTiff')
333         ds = driver.Create(path_to_file, self.engine.regionContext.width, self.engine
                .regionContext.height, 1,
334                            GDT_Float32)
335
336         band = ds.GetRasterBand(1)
337         band.WriteArray(output_grid, 0, 0)
338         band.FlushCache()
339         band.SetNoDataValue(-99)
340         band.GetStatistics(0, 1)
341
342         simpleGeoTransform = (
343             self.engine.regionContext.west, self.engine.regionContext.resolution, 0.0,
                    self.engine.regionContext.north,0.0,-1 * self.engine.regionContext
                    .resolution)
344         ds.SetGeoTransform(simpleGeoTransform)
345
346         target_crs = osr.SpatialReference()
347         target_crs.ImportFromEPSG(self.engine.regionContext.epsg)
348
349         ds.SetProjection(target_crs.ExportToWkt())
350         ds = None
351
352         pass
353
354     def getCurrentTime(self):
355         return str(strftime("%Y-%m-%d %H:%M:%S"))
356         pass
357
358     def geodataToImage(self, shapefile_path, range_max, flood_path = '', attribute
        ='exp_rc'):
359             plt.clf()
360             # plt.ion()
361
362             # Set figure size and title size of plots
363             mpl.rcParams['figure.figsize'] = (
364             self.engine.regionContext.width / 100 + 2, self.engine.regionContext
                    .height / 100)
365             mpl.rcParams['axes.titlesize'] = 12
366             mpl.rcParams['axes.labelsize'] = 10
367             mpl.rcParams['xtick.labelsize'] = 9
368             mpl.rcParams['ytick.labelsize'] = 9
369
370             # Open fire boundary data with geopandas
371             shapefile_boundary = gpd.read_file(shapefile_path)
372
373             # print(shapefile_boundary.crs)
374
375             west_extent = self.engine.regionContext.west
376             east_extent = self.engine.regionContext.west + self.engine.regionContext
                    .width * self.engine.regionContext.resolution
377             north_extent = self.engine.regionContext.north
378             south_extent = self.engine.regionContext.north - self.engine.regionContext
                    .height * self.engine.regionContext.resolution
379             plot_extent = (west_extent, east_extent, south_extent, north_extent)
380
381
382             # Plot uncropped array
383             figure, ax = plt.subplots()
384
385             shapefile_boundary.plot(ax=ax, cmap='Oranges',
386                                     column=attribute,  # categorical=True,
387
388                                     vmin=0.0, vmax=range_max,
389                                     legend=True,
390                                     legend_kwds={'label': attribute, 'orientation'
                                        : 'horizontal', 'anchor': (0.6, 1.0),
391                                                  'pad': 0.05,'shrink': 0.6,
                                        'aspect': 30},
392                                     )
393
394             flood_data = self.engine.utilities.load_raster(flood_path)
395
```

```
396             # ax.imshow(filteredFlood_data, cmap='Blues', alpha = 0.9, extent =
                    plot_extent)
397             plt.imshow(flood_data, cmap='Blues', extent=plot_extent, vmin=0, vmax=5,
                    alpha=0.9)
398             plt.colorbar(ax=ax, label='Inundation Depth (m)', orientation='vertical',
                    location='right', pad=0.03, shrink=1,
399                          aspect=30)
400
401
402             ax.set_title("Impact Map")
403             ax.axes.xaxis.set_visible(False)
404             ax.axes.yaxis.set_visible(False)
405             # ax.axis('off')
406
407             image_dir = self.engine.dataContext.getImpactDirectory()
408
409             outputLocation = image_dir + 'impact_' + attribute + '.png'
410
411             # plt.show()
412
413             plt.savefig(outputLocation, dpi=300, bbox_inches='tight')
414             plt.close()
415
416     class RasterIndexAssigner:
417         def __init__(self, input_shapefile, input_raster):
418             self.input_shapefile = input_shapefile
419             self.input_raster = input_raster
420
421             self.src_ds = gdal.Open(self.input_raster)
422             self.gt = self.src_ds.GetGeoTransform()
423             self.rb = self.src_ds.GetRasterBand(1)
424
425             pass
426
427         def fetchRasterDataByPoint(self, x, y, return_Index=False):
428
429             px = int((x - self.gt[0]) / self.gt[1])  # x pixel
430             py = int((self.gt[3] - y) / -self.gt[5])  # y pixel
431
432             if return_Index == True:
433                 return (px, py)
434
435             data = self.rb.ReadAsArray(px, py, 1, 1)
436             return float(data[0, 0])
437
438 if __name__ == "__main__":
439     main(sys.argv[1:])
440
```

## B Python Code - Flood

```
from __future__ import division

import scipy
import json
from scipy.optimize import bracket, brentq, minimize
from scipy.interpolate import griddata

import matplotlib.pyplot as plt
import matplotlib as mpl
from mpl_toolkits.axes_grid1 import make_axes_locatable

import geopandas as gpd
import numpy as np

import math

import os

class FloodSimulator:
    def __init__(self, engine, generator):
        self.engine = engine
        self.generator = generator

        self.surfaceWater = np.zeros([engine.regionContext.height, engine
            .regionContext.width])
        self.lastSurfaceWater = np.zeros([engine.regionContext.height, engine
            .regionContext.width])
        self.maxSurfaceWater = np.zeros([engine.regionContext.height, engine
            .regionContext.width])

        self.writeImage = False

        pass

    def updateSurfaceWater(self, pattern):
        self.lastSurfaceWater = self.surfaceWater.copy()
        # print "lastSurfaceWater"
        # print np.sum(self.lastSurfaceWater)

        self.surfaceWater = pattern.copy()
        # print "surfaceWater"
        # print np.sum(self.surfaceWater)

    def createHazardFootprint(self):

        for s_i in range(0, self.engine.currentSimulationContext.numberTimeSteps):
            surfaceWater_dir = self.engine.dataContext.getImpactDirectory("Flood/") +
                str(s_i) + ".tif"

            tempSurfaceWater = self.engine.utilities.load_raster(surfaceWater_dir)
            self.maxSurfaceWater = np.maximum(self.maxSurfaceWater, tempSurfaceWater)


    def writeHazardFootprint(self):

        hazardMap_dir = self.engine.dataContext.getImpactDirectory(
            "/hazard_map_") + self.engine.currentSimulationContext.currentRun
        self.engine.utilities.save_raster(hazardMap_dir + ".tif", self.maxSurfaceWater
            )

        shapefile_path = self.engine.dataContext.sourceDataDirectory + "/Buildings
            /chur_buildings.shp"
        self.geodataToImage(shapefile_path, hazardMap_dir + ".tif", hazardMap_dir + '
            .png',
                                'Hazard Footprint: Flood ' + self.engine
                                    .currentSimulationContext.currentRun)

        outputLocation = self.engine.dataContext.getImpactDirectory("Flood/") + str(
            self.engine.currentSimulationContext.currentTimeStep)

        self.engine.utilities.save_raster(outputLocation + ".tif", self.surfaceWater)

        outputDict = {}

        outputDict["flood_sum"] = np.sum(self.maxSurfaceWater)
        outputDict["flood_max"] = np.amax(self.maxSurfaceWater)
        outputDict["flood_min"] = np.amin(self.maxSurfaceWater)
        outputDict["flood_avg"] = np.average(self.maxSurfaceWater)
        outputDict["flood_median"] = np.median(self.maxSurfaceWater)

        outfile = open(hazardMap_dir + ".json", 'w')
        json.dump(outputDict, outfile)


    def preprocess(self):
        self.generator.preprocess()

        pass
```

```
81
82      def initialize(self):
83          self.generator.initialize()
84          pass
85
86      def run(self):
87          print(self.engine.utilities.getCurrentTime() + "-- flood: computing water
                depths --")
88          self.updateSurfaceWater(self.generator.generateSurfaceWater())
89
90          pass
91
92      def writeResults(self):
93          outputLocation = self.engine.dataContext.getImpactDirectory("Flood/") + str(
94              self.engine.currentSimulationContext.currentTimeStep)
95
96          self.engine.utilities.save_raster(outputLocation + ".tif", self.surfaceWater)
97
98          outputDict = {}
99
100         outputDict["flood_sum"] = np.sum(self.surfaceWater)
101         outputDict["flood_max"] = np.amax(self.surfaceWater)
102         outputDict["flood_min"] = np.amin(self.surfaceWater)
103         outputDict["flood_avg"] = np.average(self.surfaceWater)
104         outputDict["flood_median"] = np.median(self.surfaceWater)
105
106
107         outfile = open(outputLocation + ".json", 'w')
108         json.dump(outputDict, outfile)
109
110
111         if self.writeImage == True:
112             shapefile_path = self.engine.dataContext.sourceDataDirectory + "/Buildings
                    /chur_buildings.shp"
113             flood_path = self.engine.dataContext.getImpactDirectory("Flood/") + str
                    (self.engine.currentSimulationContext.currentTimeStep) + '.tif'
114             image_dir = self.engine.dataContext.getImpactDirectory('Flood/') + 'Images
                    /'
115
116             if not os.path.exists(image_dir):
117                 os.makedirs(image_dir)
118
119             outputLocation = image_dir + str(
120                 self.engine.currentSimulationContext.currentTimeStep) + '.png'
121
122             image_title = "Inudation Map at Time Step " + str(
123                 self.engine.currentSimulationContext.currentTimeStep)
124
125             self.geodataToImage(shapefile_path, flood_path,outputLocation,image_title)
126
127             pass
128
129
130         pass
131
132     def geodataToImage(self, shapefile_path, flood_path, outputLocation, image_title,
            attribute='OBJEKTART',  ):
133         plt.clf()
134         # plt.ion()
135
136         # Set figure size and title size of plots
137         mpl.rcParams['figure.figsize'] = (
138         self.engine.regionContext.width / 100 + 2, self.engine.regionContext.height /
                100)
139         mpl.rcParams['axes.titlesize'] = 12
140         mpl.rcParams['axes.labelsize'] = 10
141         mpl.rcParams['xtick.labelsize'] = 9
142         mpl.rcParams['ytick.labelsize'] = 9
143
144         # Open fire boundary data with geopandas
145         shapefile_boundary = gpd.read_file(shapefile_path)
146
147         # print(shapefile_boundary.crs)
148
149         west_extent = self.engine.regionContext.west
150         east_extent = self.engine.regionContext.west + self.engine.regionContext.width
                * self.engine.regionContext.resolution
151         north_extent = self.engine.regionContext.north
152         south_extent = self.engine.regionContext.north - self.engine.regionContext
                .height * self.engine.regionContext.resolution
153         plot_extent = (west_extent, east_extent, south_extent, north_extent)
154
155         dtm_data = self.generator.topography
156
157         flood_data = self.engine.utilities.load_raster(flood_path)
158
159         # Plot uncropped array
160         figure, ax = plt.subplots()
161
```

```
        # ax.imshow(dtm_data, cmap='Greys', extent=plot_extent)

        filteredShapefile_data = shapefile_boundary
        filteredShapefile_data.plot(ax=ax,
                                    color='orange',
                                    # column=attribute,  categorical=True,
                                    # linewidth=np.sqrt(normalized_data) * 2,
                                    # vmin=0.0, vmax=4000.0,
                                    # legend=True,
                                    # legend_kwds={'label': attribute, 'orientation':
                                      'horizontal', 'anchor': (0.25, 1.0),
                                    #              'pad': 0.05,
                                    #              'shrink': 0.6, 'aspect': 30},
                                    )

        flood_data[flood_data < 0] = 0

        # ax.imshow(filteredFlood_data, cmap='Blues', alpha = 0.9, extent =
            plot_extent)
        plt.imshow(flood_data, cmap='Blues', extent=plot_extent, vmin=0, vmax=5, alpha
            =0.9)
        plt.colorbar(ax=ax, label='Inundation Depth (m)', orientation='vertical',
            location='right', pad=0.03, shrink=1,
                     aspect=30)


        ax.set_title(image_title)
        ax.axes.xaxis.set_visible(False)
        ax.axes.yaxis.set_visible(False)
        # ax.axis('off')

        # plt.show()

        plt.savefig(outputLocation, dpi=300, bbox_inches='tight')
        plt.close()

    def postprocess(self):
        path = self.engine.dataContext.getConsequenceDirectory("Flood/")

        maxTimeStep = self.engine.damageSimulationContext.numberTimeSteps

        statsDict = {"flood_sum": [], "flood_max": [], "flood_min": [], "flood_avg":
            [], "flood_median": []}
        for ts in range(0, maxTimeStep):
            tspath = self.engine.dataContext.getConsequenceDirectory("Flood/") + str
                (ts) + ".json"

            with open(tspath) as data_file:
                data = json.load(data_file)
                statsDict["flood_sum"].append(data["flood_sum"])
                statsDict["flood_max"].append(data["flood_max"])
                statsDict["flood_min"].append(data["flood_min"])
                statsDict["flood_avg"].append(data["flood_avg"])
                statsDict["flood_median"].append(data["flood_median"])

        with open(path + "AggregateIndicators.json", 'w') as outJson:
            json.dump(statsDict, outJson, indent=4)

        pass

class TwoDFloodModel:
    def __init__(self,engine):
        self.engine = engine
        self.max_flow = 10
        self.return_period = self.engine.damageSimulationContext.return_period

    def initialize(self):

        # waterdepths should reflect the height of water at river
        self.surfaceWater = np.zeros([self.engine.regionContext.height, self.engine
            .regionContext.width])

        # flood discharge
        self.Q_flood = self.calculate_discahrge(self.return_period)


        # Inflow and outflow based on landcover
        landcover_shp_dir = self.engine.dataContext.sourceDataDirectory + '/Region
            /Landcover.shp'
        landcover_raster_dir = self.engine.dataContext.sourceDataDirectory + '/Region
            /Landcover.tif'

        dtm_dir =  self.engine.dataContext.sourceDataDirectory + '/dtm/dtm_final.tif'
        self.topography = self.engine.utilities.load_raster(dtm_dir)
```

```
        if not os.path.exists(landcover_raster_dir):
            self.engine.utilities.rasterizeShapefile(landcover_shp_dir,
                landcover_raster_dir, dtm_dir, setPixelValueFromAttributes = True,
                                attribute_name = 'type', all_touched = False)
        self.landcover = self.engine.utilities.load_raster(landcover_raster_dir)

        self.inflow_value_river_meterperhour = self.Q_flood / self.engine
            .regionContext.resolution**2 * 3600 / 100

        self.outflow_value_river_meterperhour = self.inflow_value_river_meterperhour

        # number of iterations to move water between two adjacent cells in one hour;
            it is assumed to be 60 times (everyminute)
        self.numIteration = 60
        self.adjustedTimeInterval = 1 / self.numIteration

        # inflow from upstream cell
        self.I = np.zeros([self.engine.regionContext.height, self.engine.regionContext
            .width])

        pass

    def calculate_discahrge(self, Z):
        # Q in cubic meter per second
        # Z being the return period: 2, 10, 30, 100, 300, 1000 years

        if Z == 2:
            Q = 691
        elif Z == 10:
            Q = 1051
        elif Z == 30:
            Q = 1235
        elif Z == 100:
            Q = 1410
        elif Z == 300:
            Q = 1550
        elif Z == 1000:
            Q = 1700
        else:
            Q = 0

        return Q

    def generateSurfaceWater(self):

        # It is assumed that a flood of 20hrs occurs at peak flow ==> the inflow
            discharge is set to zero after 20hrs
        if self.engine.currentSimulationContext.currentTimeStep > 20:
            self.inflow_value_river_meterperhour = 0

        for iter in range(self.numIteration):

            # Adding inflow to river
            self.surfaceWater = self.surfaceWater + (
                        self.landcover == 1) * self.inflow_value_river_meterperhour *
                            self.adjustedTimeInterval

            distribution = self.gradientMethod()

            left_flow = distribution[0, :, :]
            right_flow = distribution[1, :, :]
            top_flow = distribution[2, :, :]
            bottom_flow = distribution[3, :, :]

            drain = np.sum(distribution, axis=0)

            # self.Qs = drain

            left_flow_pad = np.pad(left_flow[:, 1:], [(0, 0), (0, 1)], mode="constant"
                , constant_values=0)
            right_flow_pad = np.pad(right_flow[:, :-1], [(0, 0), (1, 0)], mode
                ="constant", constant_values=0)
            top_flow_pad = np.pad(top_flow[1:, :], [(0, 1), (0, 0)], mode="constant",
                constant_values=0)
            bottom_flow_pad = np.pad(bottom_flow[:-1, :], [(1, 0), (0, 0)], mode
                ="constant", constant_values=0)

            # new inundation field
            self.I = left_flow_pad + right_flow_pad + top_flow_pad + bottom_flow_pad
            self.surfaceWater = self.surfaceWater + self.I - drain

            # Removing outflow to northeast outlet of river 2
            self.surfaceWater = self.surfaceWater - (
                        self.landcover == 2) * self.outflow_value_river_meterperhour *
                            self.adjustedTimeInterval

            self.surfaceWater[self.surfaceWater < 0] = 0
```

```
315         self.surfaceWater[self.surfaceWater < 0] = 0
316         return self.surfaceWater
317
318     def gradientMethod(self):
319         # max_water_distribution is defined as the maximum halved gradient
320         # there should never be more water be redistributed than
                max_water_distribution
321         # this ensures that the final inundation field converges to static values and
                thus avoids oscillation
322
323         waterelevation = self.topography + self.surfaceWater
324
325         gradient_left = (waterelevation[:, :-1] - waterelevation[:, 1:])
326         gradient_right = (waterelevation[:, 1:] - waterelevation[:, :-1])
327         gradient_top = (waterelevation[:-1, :] - waterelevation[1:, :])
328         gradient_bottom = (waterelevation[1:, :] - waterelevation[:-1, :])
329
330         gradient_left_pad = np.pad(gradient_left, [(0, 0), (1, 0)], mode="constant",
                constant_values=0)
331         gradient_right_pad = np.pad(gradient_right, [(0, 0), (0, 1)], mode="constant",
                constant_values=0)
332         gradient_top_pad = np.pad(gradient_top, [(1, 0), (0, 0)], mode="constant",
                constant_values=0)
333         gradient_bottom_pad = np.pad(gradient_bottom, [(0, 1), (0, 0)], mode
                ="constant", constant_values=0)
334
335         gradients = np.stack((gradient_left_pad, gradient_right_pad, gradient_top_pad,
                gradient_bottom_pad))
336         gradients[gradients > 0] = 0
337         total_gradients = np.sum(gradients, axis=0)
338         total_gradients[total_gradients == 0] = 1E14
339
340         relative_gradients = (gradients / total_gradients) + 0.0
341
342         max_flow_gradients = gradients * 0.5
343         max_water_distribution = np.amax(np.abs(max_flow_gradients), axis=0)
344         water_to_distribute = np.minimum(self.surfaceWater,max_water_distribution)
345         distribution = (relative_gradients * water_to_distribute) + 0.0
346
347         # free memory
348         del waterelevation
349         del gradient_left
350         del gradient_right
351         del gradient_top
352         del gradient_bottom
353
354         del gradient_left_pad
355         del gradient_right_pad
356         del gradient_top_pad
357         del gradient_bottom_pad
358
359         del gradients
360         del max_flow_gradients
361         del max_water_distribution
362         del water_to_distribute
363         del relative_gradients
364         del total_gradients
365
366         return distribution
367
```

# C Python Code - Exposure

```python
import math
from scipy.stats import lognorm

import numpy as np

import json
import csv
import ast
import copy
import os

from fiona import collection, open as fiona_open
from fiona.crs import from_epsg
from shapely.geometry import mapping, shape, Point, LineString
from shapely import wkt

# import matplotlib
# matplotlib.use("Agg")
import matplotlib.pyplot as plt


class Exposure:
    def __init__(self, engine, ):
        self.engine = engine

        self.buildings = {}

        self.buildingsRequiringUpdate = set()

        pass

    def preprocess(self):
        buildings = self.engine.dataContext.sourceDataDirectory + "/Buildings/chur_buildings.shp"
        dtm = self.engine.dataContext.sourceDataDirectory + "/dtm/" + "dtm_final.tif"

        buildingsCSV = self.engine.dataContext.preprocessedDataDirectory + "/buildings.csv"
        cell2featCSV = self.engine.dataContext.preprocessedDataDirectory + "/cell2feat.csv"
        feat2edgeCSV = self.engine.dataContext.preprocessedDataDirectory + "/feat2bldg.csv"

        ria = self.engine.utilities.RasterIndexAssigner(buildings, dtm)

        cellIndicesToFeatureIDsMap = {}
        featureIDsToEdgeIDsMap = {}

        with open(buildingsCSV, 'w', newline='') as f:
            w = csv.DictWriter(f,
                               ["feat_id", "bldg_id", "type", "area", 'indices',
                                'geometry'])
            w.writeheader()
            with collection(buildings, "r") as features:
                for feature in features:
                    raster_Index_list = []

                    origType = feature['properties']["OBJEKTART"]

                    geometry = shape(feature['geometry'])
                    feat_id = feature["id"]
                    bldg_id = feature['properties']["UUID"]

                    # centroid point
                    i_point = geometry.centroid

                    indices = ria.fetchRasterDataByPoint(i_point.coords[0][0], i_point.coords[0][1], True)

                    # check if geometry is outside of extent, should not occur if properly prepared
                    if (indices[0] > self.engine.regionContext.width - 1 or indices[1] > self.engine.regionContext.height - 1):
                        indices[0] = -1
                        indices[1] = -1

                    raster_Index_list.append(indices)


                    area = geometry.area

                    w.writerow(
                        {"feat_id": feat_id, "bldg_id": bldg_id, "type": origType, "area": area,
                         'indices': sorted(list(set(raster_Index_list))), 'geometry': geometry})

                    if indices in cellIndicesToFeatureIDsMap:
                        cellIndicesToFeatureIDsMap[indices].append(feat_id)
```

```
                else:
                    cellIndicesToFeatureIDsMap[indices] = [feat_id]

                featureIDsToEdgeIDsMap[feat_id] = bldg_id

        # write Index files
        with open(cell2featCSV, 'w', newline='') as f:
            w = csv.DictWriter(f, ["indices", "feat_ids"])
            w.writeheader()
            for key, value in cellIndicesToFeatureIDsMap.items():
                w.writerow({"indices": key, "feat_ids": value})

        with open(feat2edgeCSV, 'w', newline='') as f:
            w = csv.DictWriter(f, ["feat_id", "bldg_id"])
            w.writeheader()
            for key, value in featureIDsToEdgeIDsMap.items():
                w.writerow({"feat_id": key, "bldg_id": value})

        pass

    def initialize(self):
        buildingsCSV = self.engine.dataContext.preprocessedDataDirectory + "/buildings
            .csv"
        cell2featCSV = self.engine.dataContext.preprocessedDataDirectory + "/cell2feat
            .csv"
        feat2bldgCSV = self.engine.dataContext.preprocessedDataDirectory + "/feat2bldg
            .csv"


        self.buildingsUnitCost = json.load(
            open(self.engine.dataContext.sourceDataDirectory + "/Buildings
                /buildingsUnitCost.json"))

        print(self.engine.utilities.getCurrentTime() + "-- buildings: loading
            cell2feat.csv --")
        self.cell2featMap = {}
        with open(cell2featCSV, 'r', newline='') as cell2featCSVFile:
            IndexReader = csv.DictReader(cell2featCSVFile, delimiter=',')
            for row in IndexReader:
                self.cell2featMap[ast.literal_eval(row["indices"])] = ast.literal_eval
                    (row["feat_ids"])

        print(self.engine.utilities.getCurrentTime() + "-- exposure: loading feat2bldg
            .csv --")
        self.feat2bldgMap = {}
        with open(feat2bldgCSV, 'r', newline='') as feat2bldgCSVFile:
            IndexReader = csv.DictReader(feat2bldgCSVFile, delimiter=',')
            for row in IndexReader:
                self.feat2bldgMap[ast.literal_eval(row["feat_id"])] = str
                    (row["bldg_id"])

        print(self.engine.utilities.getCurrentTime() + "-- infrastructure: loading
            buildings.csv --")

        with open(buildingsCSV, 'r', newline='') as buildingsCSVFile:
            featuresReader = csv.DictReader(buildingsCSVFile, delimiter=',')
            for row in featuresReader:
                bldg_id = str(row["bldg_id"])
                feat_id = int(row["feat_id"])
                area = float(row["area"])
                bldg_type = row["type"]
                geometry = row["geometry"]
                bldg_unit_cost = self.buildingsUnitCost[bldg_type]
                indices = ast.literal_eval(row["indices"])[0]

                building = Building(self.engine, feat_id, bldg_id, indices, area,
                    geometry, bldg_type, bldg_unit_cost)

                building.impactEvaluator = ImpactEvaluator(self.engine,building,)

                self.buildings[feat_id] = building

                pass

        pass

    def setIntensityMeasure(self, feat_id, measure):

        building = self.buildings[int(feat_id)]

        building.impactEvaluator.updateState(measure)

        self.buildingsRequiringUpdate.add(feat_id)

        pass

    def updateBuildings(self):
        for feat_id in self.buildingsRequiringUpdate:
            self.buildings[int(feat_id)].updateState()
```

```
160
161         pass
162
163     def resetUpdateLists(self):
164         self.buildingsRequiringUpdate.clear()
165
166         pass
167
168     def writeResults(self):
169
170         format = "ESRI Shapefile"
171         suffix = ".shp"
172
173         baseName = self.engine.dataContext.getImpactDirectory("Impact/")
174
175         print(self.engine.utilities.getCurrentTime() + "-- exposure: writing impact
                properties --")
176         self.exportToGeoFile(baseName + "impactProperties", suffix, format,
                ImpactEvaluator.getStateDescription(), lambda building: building
                .impactEvaluator.getState())
177         self.exportToCSVFile(baseName + "impactProperties.csv", ImpactEvaluator
                .getStateDescription(),
178                              lambda building: building.impactEvaluator.getState())
179
180         print(self.engine.utilities.getCurrentTime() + "-- exposure: writing building
                aggregation properties --")
181         self.exportBuildingIndicators(baseName + "buildingIndicators",
182                                       {"buildingAggregators": Building.getAggregators
                                            (),
183                                        "inundationHazardAggregators": ImpactEvaluator
                                            .getAggregators(),})
184
185
186         impactLocation = self.engine.dataContext.getImpactDirectory("Impact/")
                +"impactProperties.shp"
187         hazard_footprint_dir = self.engine.dataContext.getImpactDirectory("
                /hazard_map_") + \
188                                self.engine.damageSimulationContext.currentRun + '.tif'
189
190         self.engine.utilities.geodataToImage(impactLocation, 1, hazard_footprint_dir,
                attribute = 'damage_rat')
191         self.engine.utilities.geodataToImage(impactLocation, 10000,
                hazard_footprint_dir, attribute = 'exp_rc')
192
193
194     def exportBuildingIndicators(self, path, aggregators):
195
196         # initialize aggregation variables
197         aggregation_values = {}
198         for aggregator_group_id, aggregator_group in aggregators.items():
199             aggregation_values[aggregator_group_id] = {}
200             for aggregator_id, aggregator in aggregator_group.items():
201                 aggregation_values[aggregator_group_id][aggregator_id] = 0
202
203         # fill aggregation variables
204         for feat_id, building in self.buildings.items():
205             for aggregator_group_id, aggregator_group in aggregators.items():
206                 for aggregator_id, aggregator in aggregator_group.items():
207                     value = aggregator(building,
                            aggregation_values[aggregator_group_id][aggregator_id])
208                     aggregation_values[aggregator_group_id][aggregator_id] = value
209                     # print "aggregator_group_id: " + str(aggregator_group_id) + "
                        aggregator_id: " + str(aggregator_id) + " value: " + str(value
                        )
210
211         with open(path + ".json", 'w') as s:
212             json.dump(aggregation_values, s)
213
214         pass
215
216     def exportToGeoFile(self, location, suffix, format, properties, func):
217         # append IDs to dictionary
218         schema = {
219             'geometry': 'Polygon',
220             'properties': dict(list(properties.items()) + list({"feat_id": "int",
                    "bldg_id": "str"}.items()))
221         }
222
223         nonValue = -999
224
225         with fiona_open(location + suffix, 'w', format, schema, crs=from_epsg(self
                .engine.regionContext.epsg)) as c:
226             for feat_id, building in self.buildings.items():
227                 ids = {"feat_id": feat_id, "bldg_id": building.id}
228
229                 state = func(building)
230                 if state == None:
231                     state = {key: nonValue for key in properties.keys()}
232                 propertiesDict = dict(list(ids.items()) + list(state.items()))
```

```
                mappedGeometry = mapping(wkt.loads(building.geometry))

                # print "intersection " + str(set(schemaSections["properties"].keys())
                    ^ set(propertiesDict.keys()))

                c.write({"geometry": mappedGeometry,
                         "properties": propertiesDict})

        pass

    def exportToCSVFile(self, location, properties, func):
        # combDict = dict(properties.items() + {"feat_id" : "int", "edge_id" : "str",
            "geometry" : "wkt"}.items())
        combDict = dict(list(properties.items()) + list({"feat_id": "int", "bldg_id":
            "str"}.items()))
        nonValue = -999

        # dict.keys() and dict.values() will directly return in the same order (http
            ://stackoverflow.com/questions/835092/python-dictionary-are-keys-and
            -values-always-the-same-order)
        with open(location, 'w', newline='') as fp:
            a = csv.writer(fp, delimiter=",")
            a.writerow([key for key in sorted(combDict)])

            for feat_id, building in self.buildings.items():
                ids = {"feat_id": feat_id, "bldg_id": building.id}

                state = func(building)
                if state == None:
                    state = {key: nonValue for key in properties.keys()}

                # propertiesDict = dict(ids.items() + state.items() + {"geometry" :
                    mapping(wkt.loads(section.geometry))}.items())
                propertiesDict = dict(list(ids.items()) + list(state.items()))

                a.writerow([propertiesDict[key] for key in sorted(propertiesDict)])

        pass

class Building():
    @staticmethod
    def getStateDescription():
        return {"damage_ratio" : "float", "exp_rc" : "float" }
        pass

    @staticmethod
    def getAggregators():
        return {
                "sum exp_rc" : lambda building, aggregate: aggregate + max(0, building
                    .exp_rc),
                }

    def __init__(self, engine, feat_id, bldg_id,  indices, area, geometry, bldg_type,
        bldg_unit_cost):
        self.engine = engine
        self.id = bldg_id
        self.fid = feat_id
        self.indices = indices
        self.area = area
        self.bldg_type = bldg_type
        self.bldg_unit_cost = bldg_unit_cost
        self.geometry = geometry

        self.initialize()

        pass


    def initialize(self):
        # provided by damage models

        self.damage_ratio = 0
        self.exp_rc = 0

        pass


    def updateState(self):

        self.damage_ratio = self.impactEvaluator.damage_ratio
        self.exp_rc = self.impactEvaluator.exp_rc

        pass
```

```
    def getState(self):

        return {"damage_ratio" : self.damage_ratio, "exp_rc" : self.exp_rc  }

        pass


class ImpactEvaluator():

    @staticmethod
    def getAggregators():
        return {
            "count inundated": lambda building,
                                      aggregate: aggregate + 1 if building
                                          .impactEvaluator.inundationDepth > 0 else
                                          aggregate,
            "area inundated": lambda building, aggregate: aggregate + building.area if
                building.impactEvaluator.inundationDepth > 0 else aggregate,
        }
        pass

    @staticmethod
    def getStateDescription():
        return {"inundation": "float", "damage_ratio": "float", "exp_rc": "float"}
        pass

    def __init__(self, engine, building):
        self.name = "InundationImpactEvaluator"
        self.engine = engine
        self.building = building
        self.initialize()

        self.critical_building = [""]
        self.non_critical_building = ["",""]

        pass

    def initialize(self):
        self.inundationDepth = 0

        # impact measures
        self.damage_ratio = 0
        self.exp_rc = 0

        pass

    def updateState(self, measure):

        self.inundationDepth = measure
        height = (1 + 1 / 4) * 2.5  # with the assumption of having one floor upper
            ground (1/4 height) and one basement (1 height)

        # New inundation damage model
        if self.building.bldg_type in self.critical_building:
            unit_temp_exp_rc = 900 * np.sqrt(self.inundationDepth / 0.25)
            temp_exp_rc = unit_temp_exp_rc * self.building.area * (height)
            pass

        else:  # if the building is NOT critical
            unit_temp_exp_rc = 400 * np.sqrt(self.inundationDepth/0.25)
            temp_exp_rc = unit_temp_exp_rc * self.building.area * (height)
            pass

        # measures are in CHF/km
        rebuilding_cost = (self.building.bldg_unit_cost * self.building.area)
        self.exp_rc = min(temp_exp_rc, rebuilding_cost)
        self.damage_ratio = self.exp_rc / rebuilding_cost

        pass

    def getState(self):
        return {"inundation": self.inundationDepth, "damage_ratio": self.damage_ratio,
            "exp_rc": self.exp_rc, }
        pass

    def restore(self):
        self.initialize()
        pass

class ExposureDamageSimulator:
    def __init__(self, engine, exposure):
        self.engine = engine
        self.exposure = exposure

        pass

    def preprocess(self):
        pass

```

```
395     def initialize(self):
396         pass
397 
398     def getUpdatedInnundatedCellIndices(self):
399         initial_surfaceWater = np.zeros([self.engine.regionContext.height, self.engine
                .regionContext.width])
400         pathToSurfaceWater = self.engine.dataContext.getImpactDirectory("/hazard_map_"
                ) + self.engine.currentSimulationContext.currentRun + '.tif'
401         surfaceWater = self.engine.utilities.load_raster(pathToSurfaceWater)
402         surfaceWater[surfaceWater < 0] = 0
403 
404         self.engine.floodSimulator.surfaceWater = surfaceWater
405 
406         cellIndicesSurfaceWater = np.argwhere(surfaceWater != initial_surfaceWater)
407         return cellIndicesSurfaceWater
408 
409     def run(self):
410 
411         cellIndicesSurfaceWater = self.getUpdatedInnundatedCellIndices()
412 
413         print(self.engine.utilities.getCurrentTime() + "-- exposure: updating damage
                states for buildings due to flooding --")
414         for indices in cellIndicesSurfaceWater:
415             if (indices[1], indices[0]) not in self.exposure.cell2featMap:
416                 continue
417 
418             im = self.engine.floodSimulator.surfaceWater[indices[0], indices[1]]
419 
420             #find the id of the features that are in cells where there is inundation
421             if im > 0:
422                 ids = self.exposure.cell2featMap[(indices[1], indices[0])]
423                 for id in ids:
424                     self.exposure.setIntensityMeasure(id, im)
425 
426         print(self.engine.utilities.getCurrentTime() + "-- exposure: updating states
                of buildings --")
427         self.exposure.updateBuildings()
428         self.exposure.resetUpdateLists()
429 
430         return
431         pass
432 
```